Improving Software Maintenance Ticket Resolution Using Process Mining

by

Monika Gupta

Under the supervision of

Prof. Pankaj Jalote

Dr. Alexander Serebrenik

Indraprastha Institute of Information Technology Delhi

December, 2019

This thesis is dedicated to my parents, brother, and sister, who have always been very supportive and encouraged me unconditionally. I can never forget the sacrifices they have made because of which I could pursue my passion for regular PhD and go for internship to various places.

Improving Software Maintenance Ticket Resolution Using Process Mining

by

Monika Gupta

Submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

to the
Indraprastha Institute of Information Technology Delhi
December, 2019

Certificate

This is to certify that the thesis titled "**Improving Software Maintenance Ticket Resolution Using Process Mining**" being submitted by **Monika Gupta** to the Indraprastha Institute of Information Technology Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by her under our supervision. It is to be noted that from August 2012 to August 2014 research was carried out under the supervision of Dr. Ashish Sureka. In our opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

December, 2019
Prof. Pankaj Jalote

December, 2019
Dr. Alexander Serebrenik

Indraprastha Institute of Information Technology Delhi
New Delhi 110020

Acknowledgment

Here is the nontechnical section of the thesis with no page limit, which makes a lot of relevance when I recollect the list of people who have directly or indirectly made this PhD journey worth it. After all, the thesis has some page limit and therefore I will try to be precise and mention a few names here.

My advisors, Prof. Pankaj Jalote and Dr. Alexander Serebrenik have played a key role in my PhD journey. I cannot thank them enough for their constant support and encouragement. I have learned many technical and non-technical things from a visionary like Prof. Jalote which will go a long way. I still wonder how he managed multiple roles and responsibilities so well. His feedback gave a different and more meaningful perspective which helped in shaping this thesis. Having Dr. Alexander Serebrenik as an advisor is one of the best things that has happened in my life - not an overstatement. Our collaboration started off with an internship that got converted to an advisor-advisee relationship. He is one of the most meticulous person when it comes to research. The amount of time and attention he devoted never made me feel that I am working remotely with him. I do not remember an instance when he declined the meeting request. I would have never reached this stage without his mentorship.

I am very thankful to Dr. Ashish Sureka who laid the foundation of my research career. He guided me during the first 2 years of PhD and invested a lot of his invaluable time in technical discussions. He helped me to gradually accommodate with the grad school environment, directly after finishing my undergrad. His hard work and commitment toward his students were inspirational. His teachings made an everlasting impact, which will continue to help me in my professional career.

Dr. Srinivas Padmanabhuni, my industry mentor, shared the industry research perspective and got me access to practitioners for surveys/feedback, which made a huge difference in my PhD training. His passion for trying out new things and being an entrepreneur is commendable. I will retain the motivational discussions I had with him.

It has been a great experience to collaborate with Allahbaksh Asadullah from Infosys. His strong industry experience and willingness to share knowledge made my internship at

Infosys a success. I acknowledge my IBM Research internship mentors, Atri Mandal and Dr. Gargi Dasgupta for giving me an opportunity to work on an interesting real project. Thanks for all the brainstorming that made the project a great learning experience. I would like to thank my current manager, Sampath Dechu for giving me a chance to contribute in my PhD area and explore new directions around the same. I am thankful to my co-authors, Prerna and Tarun for their help with the research project. I am grateful to all the survey respondents and data annotators for their contribution.

Thanks a lot to Prime Minister's Fellowship, Department of Science and Technology, Confederation of Indian Industry and Infosys, the industry partner, for funding my PhD. The Prime Minister's Fellowship enabled industry-academia collaboration which positively influenced my research. Special thanks to Miss Shalini, Miss Neha, and Mr. S. S. Kohli for facilitating this fellowship and organizing many useful research events. I would like to thank Microsoft, SIGSOFT, and IARCS for the conference travel grants.

I appreciate my academic siblings Ayushi, Damodaran, and Senthil for their feedback and fun time at conferences. I will miss the light-hearted chit-chats with Ayushi; hopefully one day we will co-author some of our crazy ideas. Thank you very much Anush and Tejas for having all the technical and philosophical discussions, which provided me more clarity about various concepts specifically machine learning, and helped me stay strong during PhD crisis phase (inevitable part of the journey). This PhD journey would have been less memorable without my friends - Samarth, Monalisa, Sweta, Anupama, Jayprakash, Dhriti, Surabhi, Venkatesh, Rohit, Parikshit, Garvita, Chandani, Amani, Denzil, Swati, Pooja, and Ankita. Thanks a lot Trasha for motivating me to join PhD at IIIT Delhi. Nancy and Sakshat, thanks for bearing with a PhD friend. Their phone calls were always a pleasant surprise and reminder that things were different outside grad school. I acknowledge Mr. Vinod and Miss Priti from non-academic staff for their timely responses and coordinating formal requirements of PhD.

I had an opportunity to intern at various places, including Eindhoven University, Infosys, and IBM Research during my PhD where I met many people with different interests and learned different things from each of them. I am thankful for their kind hospitality and all the love that made my stay comfortable in different states and different countries.

None of this would have been possible without the support of my loving and caring family. My grandmother who barely understands PhD but still continuously asked me about my PhD updates, which helped me stay focused. Maa Papa, thanks will not do justice to all the prayers and sacrifices you have made to see me growing in my career. They have always stood by my side, which kept me going and helped me embrace failures as much as success. Home-made snacks that my mother never missed to pack whenever I traveled felt like a warm expression of her love and care. For the efforts and attention that my father invested, he deserves this degree more than I do. My brother Lokesh played multiple roles in this journey - he has been a counselor, a patient listener, an informal advisor for handling PhD/non-PhD situations, and an optimist. I looked forward to him for so many things and he was always there for me. I am very thankful to my sister Ayushi who performed family responsibilities on my behalf whenever I was unavailable. I will really miss stealing her clothes in the name of no shopping time due to hectic PhD life. I am blessed with the best Chachu (uncle) who has been my teacher also. The way he taught me the concepts of mathematics during my school days got me interested into the subject and inculcated the problem-solving ability, a crucial requirement for PhD. My Chachi (aunt) did her bit by appreciating and wishing for every milestone in my personal and professional life. Love to Shikha and Shivam for the resolute cheerfulness. Special thanks to my parents-in-law for showing interest and regularly following up on my PhD timeline. I am very grateful to my family for everything they have done to make this adventurous PhD journey full of memories. I express my gratitude to Rambilas Uncle, Rakesh Sibbal Uncle, and Seema Sibbal Aunty for praising my little steps toward the career.

This thesis and acknowledgment would be incomplete without the mention of my understanding and loving husband, Ankush. He happily supported me with my PhD deadlines in all possible ways when a usual couple would have been dating/going out. His true-hearted presence made the whole world difference to the last leap of my PhD journey.

Improving Software Maintenance Ticket Resolution Using Process Mining

by
Monika Gupta

Abstract

Software maintenance refers to the modification of software product after delivery and is required to correct faults, to improve the performance or other attributes, or to adapt the product to a modified environment. It is a crucial activity in the software industry and consumes a major portion of the expenditure on software. It is known that the performance of an organization can be improved by improving the process. Therefore, given the importance and cost involved, there is need to continuously improve the software maintenance process.

This thesis focuses on analyzing and improving the software maintenance ticket resolution process by exploring novel applications of process mining. We decided to study the ticket resolution process as it is an important part of a software maintenance process. Process mining consists of mining event logs generated from business process execution supported by information systems.

To identify the potential opportunities for improvement in software process management by mining data repositories, we first conducted qualitative interviews and surveys of more than 40 managers in a large global IT company. The survey provided us with a list of more than 10 maintenance process challenges encountered by practitioners, and benefits that might accrue by addressing them. This thesis addressed a few of the identified challenges pertaining to the software maintenance ticket resolution process. We studied different types of software maintenance tickets, that is, software bug tickets and IT support tickets. As identified from the survey, there is a need to analyze the data generated during the ticket resolution process to capture process reality and identify the process inefficiencies. Hence, we proposed a framework to analyze the data for ticket resolution process from diverse perspectives, by applying process mining techniques. Using the proposed framework, we discovered the process model that captured the control flow, timing and frequency information about events. We then studied inefficiencies such as self-loops, back-forth, ticket reopen, timing issues, delay due to user input requests, and effort consumption. We also analyzed the degree of conformance between the designed and the runtime (discovered) process model. The data-driven insights helped to make process improvement decisions. For example, using the proposed framework on IT support ticket data for a large global IT company we found that around 57% of the tickets had user input requests in the life cycle, causing significant delays in user-experienced resolution time. Therefore, we proposed a machine learning based-system that preempts a user at the time of ticket submission with an average accuracy of around 94% to provide additional information that the analyst is likely to ask, thus mitigating delays due to later user input requests.

Also, we explored unstructured data generated during process execution to derive insights that could not be obtained solely from structured data (event logs). To achieve this, we extracted topical phrases (keyphrases) from the unstructured data using an unsupervised graph-based approach. The keyphrases were then integrated into the event log, which got reflected in the discovered process model.

To resolve a ticket, some code changes were made, which led to anomalies such as regression bugs. We aimed to detect whether ticket resolution caused some anomalous behavior so as to reduce the post-release bugs, one of the important challenges identified from the survey. To achieve this, we proposed an approach to discover an execution behavior model for the deployed and the new version using the execution logs that is, runtime print statements. Differences between the two models were then identified, which allowed programmers to detect anomalous behavior changes, that is, not consistent with code changes thereby identifying potential bugs that might have been introduced during code change. We applied the proposed framework and solution approaches on a series of case studies on data sets of commercial and open source projects.

Through the aforementioned contributions, we explored the potential of applying process mining using various data sources to improve various aspects of the software maintenance ticket resolution process. Such analysis usually focuses on identifying the inefficiencies, but as we observed in the thesis, it can also lead to automation opportunities to make the ticket resolution process more efficient.

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

Abbreviations

AUI Awaiting User Inputs.

BNR bottleneck ratio.

CMM Capability Maturity Model.

CMMI Capability Maturity Model Integration.

DIG Document Index Graph.

DM Delivery Manager.

EB Expected Behavior.

EBM Execution Behavior Model.

FLOSS free-libre/open-source software.

FM Fitness metric.

FN False Negative.

FP False Positive.

FSA Finite State Automata.

HL Hamming Loss.

ITIS IT Infrastructure Support.

ITS Issue Tracking System.

MSR Mining Software Repositories.

NIM Net Importance Metric.

OSS open source software.

PCA principal component analysis.

PCR peer code review.

PM Project Manager.

RBF radial basis function.

RDF random decision forest.

RHEL Red Hat Enterprise Linux.

ROC receiver operating characteristic.

S2R Steps to Reproduce.

SDLC software development life cycle.

SLA service-level agreement.

SLRT service-level resolution time.

SPI Software Process Improvement.

SPM Senior Project Manager.

SVM support vector machine.

SWEBOK Software Engineering Body of Knowledge.

TL Team Lead.

TN	True Negative.
TP	True Positive.
URT	user-experienced resolution time.
VCS	version control system.

Research Dissemination

Journals

1. **Monika Gupta**, Srinivas Padmanabhuni, Allahbaksh Mohhamedali Asadullah, and Alexander Serebrenik. "Reducing User Input Requests to Improve IT Support Ticket Resolution Process." *Empirical Software Engineering Journal*, 2017.
   Impact Factor - 3.275.

   Presented in Journal First at 2017 *IEEE International Conference on Software Maintenance and Evolution.*

Peer Reviewed Conference Publications

1. **Monika Gupta**. "Nirikshan: Process Mining Software Repositories to Identify Inefficiencies, Imperfections, and Enhance Existing Process Capabilities." *In Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 658-661. ACM, 2014.

2. **Monika Gupta**, and Ashish Sureka. "Nirikshan: Mining Bug Report History for Discovering Process Maps, Inefficiencies and Inconsistencies." *In Proceedings of the 7th India Software Engineering Conference.* pp. 1-10, ACM, 2014.

3. **Monika Gupta**, Ashish Sureka, and Srinivas Padmanabhuni. "Process Mining Multiple Repositories for Software Defect Resolution from Control and Organizational Perspective." *In Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 122-131. ACM, 2014.

4. **Monika Gupta**, and Ashish Sureka. “Process Cube for Software Defect Resolution.” *In Proceedings of the 21st Asia-Pacific Software Engineering Conference*, 2014, vol. 1, pp. 239-246. IEEE, 2014.

5. **Monika Gupta**, Ashish Sureka, Srinivas Padmanabhuni, and Allahbaksh Mohammedali Asadullah. “Identifying Software Process Management Challenges: Survey of Practitioners in a Large Global IT Company.” *In Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 346-356. IEEE Press, 2015.

6. **Monika Gupta**. “Improving Software Maintenance Using Process Mining and Predictive Analytics.” *In Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, pp. 681-686. IEEE, 2017.

7. **Monika Gupta**, Atri Mandal, Gargi Dasgupta, and Alexander Serebrenik. “Runtime Monitoring in Continuous Deployment by Differencing Execution Behavior Model.” *In International Conference on Service-Oriented Computing*, pp. 812-827. Springer, Cham, 2018.

8. **Monika Gupta**, Prerna Agarwal, Tarun Tater, Sampath Dechu, and Alexander Serebrenik. “Analyzing Unstructured Data to Discover Underlying Business Process Interactions.” (Under Submission).

Posters

1. **Monika Gupta**. “Improving Software Maintenance Using Process Mining and Predictive Analytics.” *International Conference on Software Maintenance and Evolution 2017.*

2. **Monika Gupta**, Atri Mandal, Gargi Dasgupta, and Alexander Serebrenik. “Detecting Anomalies in the DevOps Ecosystem by Mining Multiversion Execution and Code Flow Graphs.” *IBM Research 2017.*

3. **Monika Gupta**, and Alexander Serebrenik. “Reducing Ticket Resolution Time for

IT Support Process Using Prognostic Diagnostic Model." *Grace Hopper Conference India 2016* (received **Best Poster Award**).

4. **Monika Gupta**, Ashish Sureka, and Srinivas Padmanabhuni "Process Mining Multiple Repositories for Software Defect Resolution from Control and Organizational Perspective." *Grace Hopper Conference India 2014*.

5. **Monika Gupta**, and Ashish Sureka "Nirikshan: Process Mining Software Repositories to Identify Inefficiencies, Imperfections, and Enhance Existing Process Capabilities." *International Conference of Software Engineering 2014*.

Chapter 1

Introduction and Related Work

Software development is a collective, complex, and creative effort [1]. A software product goes through the following stages during its life cycle: requirement analysis and specification, design, implementation, integration and testing, deployment, operation, and maintenance [2]. Each aforementioned stage is quite complex and involves various processes. Three critical dimensions that organizations typically focus on to improve their businesses are as follows: people, procedures and methods, and tools and equipment [3]. While people and technology are important, processes bind these dimensions together. Managers must make continuous efforts to assess and improve the software processes because the quality of a process is directly related to the quality of the developed software and the productivity of an organization [4][1] [5]. This motivated various software process research initiatives, such as better ways to model the developer organization processes (*process modeling*) and better ways of assessing and improving the processes of the organization (*process assessment and improvement*) [1][6]. Various Software Process Improvement (SPI) frameworks [7] exist to assess and improve the software development processes, such as Capability Maturity Model Integration (CMMI) [8] and ISO/IEC 15504 (SPICE) [9][10], and quality improvement paradigm [11].

Software maintenance is a crucial activity in the software industry and consumes a major portion of the expenditure on software [12][13]. Software maintenance refers to the modification of a software product after delivery and is required to correct faults, improve the performance or other attributes, or adapt the product to a modified environment. Based on study by Lientz and Swanson [14], there are four categories of maintenance:

- Adaptive - to keep a software product usable in the changing environment
- Perfective - to implement changed or new user requirements
- Corrective - to fix discovered faults
- Preventive - to prevent problems in the future

The primary focus of the aforementioned SPI initiatives was on the overall project management, and not on the issues specific to software maintenance. The SWEBOK initiative identified a number of activities and practices specific to software maintenance, such as service-level agreement negotiation and handling of tickets [15]. Therefore, models, such as software maintenance model, SM^{mm} (complement to CMMI version 1.1 [16]), are developed to assess and improve the quality of software maintenance [17]. However, application of these initiatives requires professional judgment to evaluate how an organization benchmarks against the reference model that is not scalable. Ticket reporting and management is an important part of software maintenance. It is supported by various information systems such as ticketing system, peer code review system, and version control system (for source code base). A ticket once reported gets assigned to a developer (analyst), a person responsible for servicing the ticket. Ticket resolution can involve code changes thus, code patches are submitted to peer code review system before committing to the final code base. Tracking of software maintenance tickets is critical for efficient improvement of software quality.

Software development and maintenance is supported by various tools, which generate a huge amount of data archived in software repositories. The mining software repositories field analyzes the rich data available in software repositories to uncover interesting and actionable information about software projects; thus, it relies less on the practitioner's intuition and experience [18] [19] [20] [21] [22] [23]. Software repositories can be broadly divided into the following three categories [22]:

- *Historical repositories*, such as source control repositories, bug repositories, and archived communications, record a lot of information about the evolution and progress of a project.

Table 1.1: Software engineering data, mining algorithms, and software engineering tasks

Software Engineering Data	**Mining Algorithms**	**Tasks**
Sequences: Execution/ Static traces, co-changes, and so on	Association rule mining, frequent item set/subseq/partial-order mining, seq matching/clustering/classification, and so on	programming, maintenance, bug detection, debugging, and so on
Graphs: Dynamic/static call graphs, program dependence graphs, and so on	Frequent subgraph mining, graph matching/clustering/classification, and so on	bug detection, debugging, and so on
Text: Bug reports, emails, code comments, documentations, and so on	text matching/clustering/classification, and so on	maintenance, bug detection, debugging, and so on

- *Run time repositories*, such as deployment logs, contain information about the execution and the usage of an application at a single or multiple deployment sites.

- *Code repositories*, such as GitHub, SourceForge.net, and Google code, contain the source code of various applications developed by several developers.

The repositories are analyzed by applying various data mining techniques, such as classification, association rule mining, and clustering, to support different software engineering tasks such as programming, defect detection, testing, debugging, and maintenance [22]. Table 1.1 [21] presents example software engineering data being mined (the first column) and example software engineering tasks (the last column) assisted by applying various mining algorithms (the middle column) on each type of software engineering data listed in the first column. A comprehensive literature survey on the approaches for mining software repositories was presented by Kagdi *et al.* [23].

Ever-changing customer needs and rapid technical progress highlight the need to continuously improve software maintenance processes to make it more efficient. Given the complexity and cost for software maintenance ticket resolution [12], it is important to improve the efficiency of ticket resolution process. The goal of this thesis is to help improve the ticket resolution process by analyzing the data stored in software repositories during ticket resolution.

Software maintenance ticket resolution has been analyzed by mining software repositories for multiple tasks, such as duplicate bug detection [24][25], bug triaging [26][27][28][29], component assignment [30][31], resolution time prediction [32][33], reopen prediction [34][35][36], reassignment prediction [37], and bug prediction [38][39][40]. The existing studies [21][23] facilitate a variety of tasks by applying different data mining techniques; however, they do not focus on end-to-end process analysis. This highlights the need for techniques to objectively capture the integrated view of the ticket resolution process to efficiently improve it. Most research appears to begin with a particular problem, and then develops the investigation from that problem. For instance, existing techniques can facilitate efficient bug (ticket) assignment [26][27][28], duplicate detection [24][25], and other specific tasks during the ticket resolution process. However, at the outset, we need to analyze and understand the actual ticket resolution process from different perspectives to objectively identify its inefficiencies and improvement possibilities. Accordingly, we can decide whether it needs improvement in duplicate detection or bug assignment or other to enhance the overall process quality. Previous studies have indicated that this can be achieved through process mining. Process mining consists of mining event logs generated from business process execution supported by information systems. The application of process mining for end-to-end ticket resolution process analysis is challenging due to the unstructuredness of the software ticket resolution process, distribution of data over multiple data sources of different kinds (software repositories, e.g., a version control system or an issue tracker), and data format not being directly usable for existing process mining techniques. In this thesis, we contribute methods that can provide a promising lens to study the software ticket resolution process with a holistic perspective. The aim of the thesis is *to analyze and improve the software maintenance ticket resolution process by exploiting novel applications of process mining.*

In the next section we discuss related work in the area of ticket resolution in software maintenance. In Section 1.2, we discuss related work in process mining. Having provided a brief background on the two key aspects of the thesis, that is, ticket resolution and process mining, we then provide an overview of the thesis in Section 1.3.

1.1 Ticket Resolution in Software Maintenance

Ticket reporting and resolution is a key aspect of software maintenance. Tickets help track and coordinate software maintenance efforts. This is supported by a ticketing system (e.g., Bugzilla issue tracking system) where tickets are reported by developers and users. Tickets usually have specific elements, including structured fields such as reporter, reporting time, priority and agent, and unstructured fields such as description, comments and attachments. Analysts (such as developers) responsible for servicing the ticket may depend on other repositories such as code base and revision history, which also need to be referred to for fixing the ticket. A ticket goes through various stages during its lifecycle (starting from ticket reporting to ticket resolution), which are discussed in the following section.

1.1.1 Ticket Lifecycle

A ticket goes through various states such as ticket assignment to a suitable agent (developer), component assignment (a component of software to which the issue pertains), need info (asking of information from the reporter), resolution, and closure during the resolution lifecycle. The states can be broadly grouped into the following major phases [41]:

- **Ticket understanding:** A ticket once reported needs to be understood such that it can be summarized, filtered as duplicate, and assigned value for attributes such as priority and severity. A good understanding of the ticket plays a key role in the overall lifecycle of ticket resolution.

- **Ticket assignment:** Based on the initial understanding of the reported ticket, a ticket needs to be assigned to a suitable analyst (developer). A developer is responsible for servicing the ticket. If a ticket is assigned to an inappropriate developer, it needs to be reassigned to another developer. These reassignments degrade the quality of ticket resolution because the probability of a ticket being fixed decreases with an increased number of reassignments [26].

- **Ticket fixing:** An agent (developer) assigned to a ticket needs to take suitable actions to resolve (fix) an issue. This may need discussion with the reporter for various reasons such as asking for information; these interactions are usually captured as comments. The fixing of some issues may need code changes. Hence, a developer generates a patch and commits it to the final code base.

There are many existing studies corresponding to various activities in each of these stages. We discuss some of the existing studies in the next section.

1.1.2 Studies for Improving Ticket Resolution

A large number of tickets are submitted, which can lead to the queuing of work and delay the ticket resolution. Several approaches have been proposed to automate various ticket resolution activities from the aforementioned stages of the ticket lifecycle.

Ticket understanding: We briefly discuss studies on ticket (bug) summarization, duplicate detection, priority prediction, severity prediction, and reopen prediction. These are some of the main activities performed as part of this stage of ticket lifecycle.

- **Bug Summarization:** Automatic summarization helps to reduce the size of bug reports by selecting a subset of existing sentences, that is, extraction approach. Various supervised and unsupervised learning approaches have been used to predict whether a sentence from the existing report should be a part of the summary or not. In supervised learning, a prediction model is trained on the labeled summary data to predict whether a sentence belongs to the summary of a new bug report [42][43]. Unlike supervised approaches, unsupervised bug summarization approaches [44][45] do not require labeled data. The summary is generated using various unsupervised approaches, such as centrality, and a hypothetical model.

- **Duplicate Detection:** If a ticket similar to a ticket already submitted is reported, it is referred to as duplicate of the existing ticket. Duplicate detection is important to avoid redundant efforts towards the resolution. However, it is a tedious task given a large number of tickets, and it is not straightforward to determine whether two tickets indeed correspond to the same issue. There are existing studies for automatic duplicate detection which can be grouped as textual information analysis based detection and hybrid information analysis based detection. Textual information analysis based approaches [46][47] rely on textual contents, such as summaries, descriptions, and comments, of bug reports to detect textual similarity between a new bug report and historical bug reports. Hybrid information analysis based detection approaches [48][49][50] combine non textual information, such as severity, product, and execution information, with textual similarity to detect the duplicates more accurately.

- **Priority Prediction:** Priority is a structured field associated with tickets, which has categorical values such as high/medium/low or P1/P2/P3/P4/P5 (highest to lowest priority). It helps developers prioritize tickets, that is, tickets with the highest priority needs to be given more attention. The manual assignment of priority based on the understanding from the initially reported ticket is time-consuming. The existing automatic priority prediction approaches [51][52] make use of textual content and other ticket attributes to train a machine learning model. Using a trained model, a new ticket is classified to an appropriate priority level.

- **Severity Prediction:** Severity is another predefined field of bugs (tickets) that indicates the degree of impact on the functionality. It can have levels such as blocking, critical, major, normal, minor, and trivial. It needs the expertise to gauge the impact by the manual inspection of a reported ticket. Using various ticket attributes, the existing automatic severity prediction approaches predict the severity level for a new ticket at different levels of granularity. For example, some studies [53] proposed solutions to predict the severity level as severe or non-severe. However, other approaches [54][55] can predict at a more granular level and assign a specific severity level to a new ticket.

- **Reopen Prediction:** A ticket can be reopened for multiple reasons, such as availability of additional information for a better understanding of the issue, wrongly fixed bug, or regression bugs. The reopening of tickets increases maintenance efforts and leads to rework [36], and hence it is undesirable. Various machine learning algorithms using different sets of features have been proposed to predict ticket reopening [35][56].

Ticket triaging (assignment): Assignment of a ticket to a suitable developer for fixing the ticket is an important activity. The automatic triaging of tickets can reduce the waiting time for deciding the appropriate developer and also reduce the probability of reassignments.

The existing approaches for recommending an appropriate developer (analyst) employ different techniques, which are based on machine learning [27][57][28], expertise model [58][59], tossing graph [60], social network [61], or topic model [62][63]. The performance of machine learning based approaches for developer recommendation can be further improved using a feature selection technique [64] and a composite model [65].

Ticket fixing: Assignees investigate the reported issue and find out whether it needs a code change. We discuss the studies for automatic bug localization and patch generation, which can reduce the overall fixing time.

- **Bug localization:** The fixing of some bugs may require code changes. It needs to be localized by the assigned developer. It requires a good understanding of the bug and the code base for bug localization. Hence, it is time consuming. IR-based approaches provide a ranked list of candidate source code files where the bug may appear [66][67][68]. To generate this list, a bug report is treated as a query, and the source code files correspond to the document corpus. The accuracy of automatic bug localization can be improved by considering additional information such as structured information [69] or by using compositional vector space models [70]. However, this enhancement increases the running cost and algorithm complexity.

- **Patch generation:** Developers need to write code patches to fix the bugs. Various automatic patch generation approaches have been proposed for efficient patch generation. Genetic programming-based solutions [71][72] have been proposed wherein program variants are generated by introducing mutations. The variant that passes all test cases is regarded as a successful patch. However, this requires test cases for patch evaluation and thus identifies the correct patch. Bug report analysis-based approaches are proposed to overcome this dependency. Liu *et al.* [73] proposed an approach to produce the patches based on bug reports, which do not require test cases. This approach generated more correct patches and reduced the patch generation time.

As discussed earlier, studies for automating various activities from the ticket resolution process aimed to optimize the overall ticket resolution process by automating a specific activity. They did not focus on analyzing the reality of ticket resolution process using data generated during the resolution process. It is important to discover the process reality and identify inefficiencies, thus making informed process improvement decisions. This thesis aimed to analyze the ticket resolution process by discovering process reality from logs generated during process execution, thus supporting effective process improvement decisions.

1.2 Process Mining

Process mining consists of mining event logs generated from business process execution supported by information systems to capture business processes [74][75]. Process mining is aimed at providing fact-based insights about business processes and hence support efficient process improvements [76]. Process mining includes process discovery, process performance analysis, conformance verification, case prediction, history-based recommendations, and organizational analysis [76]. It has already been applied to analyze business processes from multiple domains [75]. Many process mining framework and tools, such as ProM[1] (open source) and Disco[2] (commercial), are used to derive process model and analyze data from different perspectives.

1.2.1 Types of Process Mining

As shown in Figure 1-1, business processes are supported by software systems that collect, organize, and store data (such as messages and transactions). Event log is extracted from the stored data for process mining, including: (a) process discovery, (b) conformance, and (c) enhancement [76]. Event log consists of cases where each case is an instance of a process.

As part of the process mining, a discovery technique takes an event log and produces a model capturing the behavior recorded in the log. Multiple process discovery algorithms exist. For example, the *alpha*-algorithm can depict the behavior recorded in the log as a Petri net [77]. It represents the control flow of the process. The quality of the discovered

[1]http://www.processmining.org/prom/start
[2]http://www.fluxicon.com/

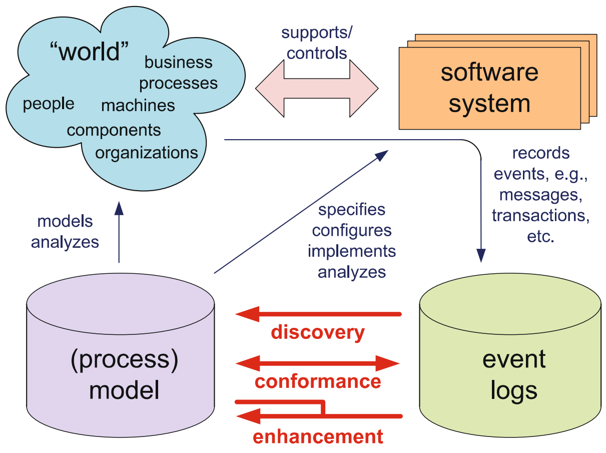


Figure 1-1: Process mining: discovery, conformance, and enhancement [76].

model is assessed on multiple dimensions, with few are follows [76]:

- *Fitness:* The discovered model should allow for the behavior seen in the event log.
- *Precision:* The discovered model should not allow for behavior completely unrelated to what was seen in the event log.
- *Generalization:* The discovered model should generalize the example behavior seen in the event log.
- *Simplicity:* The discovered model should be as simple as possible.

There should be a balance between the four criteria as there exists a trade-off between them [76].

The second type of process mining is **conformance**. Conformance checking can be used to check whether reality, as recorded in the log, conforms to the model and vice versa. The model may have been constructed by hand or may have been discovered. Conformance checking relates events in the event log to activities in the process model and compares both. The goal is to find commonalities and discrepancies between the modeled behavior and the observed behavior. Conformance checking is relevant for: (a) business alignment and auditing, that is, when cases deviate from the defined model and corrective actions are needed, and (b) measuring the performance of process discovery algorithms thus, repair models that are not aligned well with reality. Conformance checking can be achieved by replaying tokens or comparing footprints.

The third type of process mining is **enhancement**. The idea is to extend or improve a defined process model using information about the actual process discovered from the event log. For example, one can extend the analysis to identify the bottlenecks using the time stamps recorded for each event. Thus, appropriate measures can be taken to mitigate the delays.

Different process mining perspectives can be identified as follows:

- *Control-flow perspective:* It focuses on the ordering of activities. The goal of mining this perspective is to find a good characterization of all possible process paths that can be expressed in terms of a Petri net or some other notations (e.g., EPCs, BPMN, and UML ADs).

- *Organizational perspective:* It focuses on information about resources hidden in the log, that is, which actors (e.g., people, systems, roles, and departments) are involved and how they are related. The goal is to either structure the organization by classifying people in terms of roles and organizational units or show the social network.

- *Case perspective:* It focuses on properties of cases. Obviously, a case can be characterized by its path in the process or by the originators working on it. However, cases can also be characterized by the values of the corresponding data elements. For example, if a ticket gets reopened, it is interesting to investigate its characteristics.

- *Time perspective:* It is concerned with the timing and frequency of events. When events bear time stamps, it is possible to discover bottlenecks, measure service levels, monitor the utilization of resources, and predict the remaining processing time of running cases.

1.2.2 Event Log

Event log generation and extraction are crucial parts of process mining. As shown in Figure 1-1, event log is the starting point for process mining. It is taken as input for process discovery and further process analysis. Typically, the structure of event log is defined as follows:

Ticket ID	Priority	Activity	Time stamp	Actor
6672	High	Create	1/20/2009 12:14	metalsi...@gmail.com
6672	High	Open	5/14/2009 23:49	j...@chromium.org
6672	High	WontFix	9/17/2009 21:40	suna...@chromium.org
6672	High	Closed	9/17/2009 21:48	NULL
7187	Low	Create	1/29/2009 14:30	igi...@gmail.com
7187	Low	Fixed	5/14/2009 23:52	j...@chromium.org
7187	Low	Closed	5/14/2009 23:52	j...@chromium.org

Table 1.2: Sample Event Log with Ticket ID and Priority as Case-Specific Attributes, and Activity, Time Stamp and Actor as Event-Specific Attributes.

- Event log consists of cases where each case is an instance of a process. For example, every ticket (issue) reported in the Issue Tracking System (ITS) is an instance of the ticket resolution process.
- Case consists of events such that each event relates to precisely one case, that is, each event is associated with a unique case ID.
- Events can have attributes. Examples of typical attribute names are activity, time, costs, and resource.

Therefore, as illustrated in Table 1.2, every entry in the event log is an event referring to a case, activity, time stamp, and additional attributes such as actor (resource), associated cost, and duration. We can classify the attributes in each entry as follows:

- **Case-Specific Attributes**: These are attributes associated with a case that remain constant for all the events pertaining to the same case. For instance, in the given example Table 1.2, *Ticket ID* and *Priority* are case attributes.
- **Event-Specific Attributes**: These are attributes characterizing each event that are different for the same case across its life cycle, such as *Activity, Time Stamp*, and *Actor* (refer to Table 1.2).

Every attribute has its significance and contributes differently to the analysis such as:

- Case ID: It uniquely identifies the case. It helps to visualize the life cycle of each case in discovered process models. For example, here *Ticket ID* can be selected as case ID to capture the complete flow starting from *Create* till the last activity.

- Activity: Every event is related to some activity embarking the progress of case life cycle, for example, *Assigned, Resolved,* and *Closed*.

- Time stamp: All events have an associated time stamp, a datetime attribute. It enables ordering of activities on the basis of execution time and allows analysis such as bottleneck identification.

- Resource: It is basically the information about the person performing the activity, that is, actor. Organizational analysis can be performed if this information is available. For example, email id of the ticket reporter.

- Other attributes: Additional attributes can be useful for more interesting and diverse analysis. However, they are not necessary for basic types of process mining.

Also, we can say that an event log is a *multiset of traces* (the same trace may appear multiple times) where the trace is a sequence of activity names. We represent trace with activities ordered by time stamp. For example, in Table 1.2, there are two unique traces exist, each with the frequency of occurrence as one each and can be represented as $< Create, Open, WontFix, Closed >^1$ and $< Create, Fixed, Closed >^1$ where superscript corresponds to the frequency of the trace.

1.2.3 Process Model Representations

Various conventional process models, such as Petri nets [77], BPMN [78], WF nets [79], EPCs [80], YAWL [81], and UML activity diagrams, have problems such as internal inconsistency (deadlocks, livelocks, and so on) and inability to represent the underlying process well [82]. Causal nets are a representation tailored toward process mining and address the limitations of conventional languages in the context of process mining [76]. A causal net is a graph where nodes represent activities and arcs represent causal dependencies [82]. In a causal net, there is one start activity and one end activity. Each activity has a set of possible input bindings and a set of possible output bindings. Output bindings create obligations, whereas input bindings remove obligations. A valid binding sequence models an execution path starting with start activity and ending with end activity while removing all obligations created during execution. The behavior of a causal net is restricted to valid binding sequences. Hence, the routing logic is solely represented by the possible input and output bindings. Causal nets are particularly suitable for process mining given their declarative nature and expressiveness without introducing all kinds of additional model elements (places, conditions, events, gateways, and so on). Algorithms such as heuristic mining [74], and fuzzy mining [83] use representations similar to causal nets.

1.2.4 Algorithms for Process Discovery

Many process discovery techniques, based on algorithmic, machine learning, and probabilistic approaches, have been conceived in literature. Cook *et al.* proposed three different methods for process discovery in the context of software engineering [84]. The RNet, Ktail and Markov methods adopt statistical, algorithmic, and hybrid approaches respectively [84]. Agrawal *et al.* and Datta *et al.* presented approaches to construct process models from logs of workflow management systems [85][86]. Weerdt *et al.* presented an extensive overview of available process discovery techniques and evaluated the performance on real-life event logs [87].

The α-algorithm can be considered as one of the most substantial techniques in the process mining field. It represents the behavior recorded in a log as a Petri net [77]. The algorithm assumes event logs to be complete and without any noise. Therefore, the α-algorithm is sensitive to noise and incompleteness of event logs. Moreover, the original α-algorithm was incapable of discovering short loops or nonlocal, nonfree choice constructs. The original α-algorithm is improved to mine short loops ($\alpha+$-algorithm [88]) and detect nonfree choice constructs ($\alpha++$-algorithm [89]). Heuristic miner extends α-algorithm such that it applies frequency information with regard to three types of relationships between activities in an event log: direct dependency, concurrency, and not direct connectedness [74]. A Heuristic algorithm is prone to be noise resilient because of threshold parameter settings and therefore expected to be robust in a real-life context. While heuristics miner can discover short loops and non local dependencies, it lacks the capability of detecting duplicate activities.

Günther *et al.* [83] proposed a fuzzy miner, an adaptive simplification and visualization technique based on *significance* and *correlation* measures to visualize the behavior in event logs at various levels of abstraction. The main contribution of a fuzzy miner is that it can also be applied to less-structured, or unstructured processes, which is mostly the case for real-life environments. The Heuristics miner, which also employs heuristics to limit the set of precedence relations (not event), is closely related to a fuzzy miner. Fuzzy miner discovers the process more precisely as compared with heuristic miner even in case of less-structured process behavior [83]. The main limitation of a fuzzy model is that it cannot be translated to a formal Petri net that limits a comparative evaluation to other process discovery techniques. We present the fundamentals of a fuzzy miner in next chapter, which is used to discover the process model for our case studies.

1.2.5 Studies using Process Mining on Software Repositories

We present an overview of previous studies that applied process mining on software repositories. Samalikova *et al.* investigated CMMI from a process mining perspective and identified model components for which process mining techniques can be applied [90]. The results of a case study on the change control board process illustrated that process mining could provide CMMI assessors with the relevant information [90]. Kim *et al.* proposed a distributed workflow mining approach to discover the workflow process model, incrementally amalgamating a series of vertically or horizontally fragmented temporal work cases [91]. Sunindyo *et al.* proposed an observational framework that supported OSS project managers in observing project key indicators such as checking conformance between the designed and the actual process models [92]. They used a hypothesis testing approach to verify the design model with a runtime event log from bug history data. They obtained a process map for RHEL bug history data using a heuristic mining algorithm of the process mining tool ProM [92]. Knab *et al.* presented an approach of extracting general visual process patterns for effort estimation and analyzing problem resolution activities for ITS [93]. Ashish *et al.* presented a generic framework for software process intelligence involving mining and analysis of software processes [94].

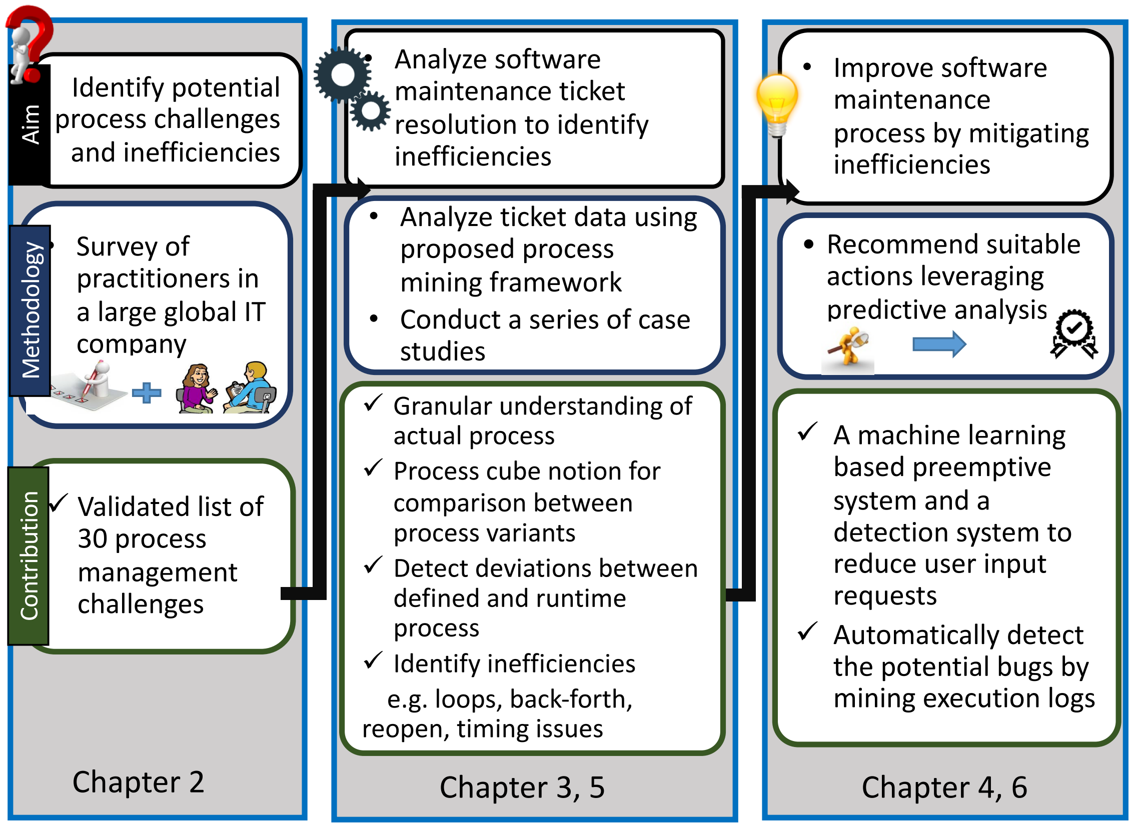
Process mining is applied to data integrated from multiple information systems. Poncin *et al.* presented a framework called as “FRASR” (FRamework for Analyzing Software Repositories) that facilitates combining and matching of events across multiple repositories [95].

They presented an approach to combine related events (belonging to the same case) spanning across multiple software repositories, such as mail archives, subversion and bug repositories, followed by the assignment of role to each developer [95]. Poncin *et al.* presented a methodology for assessing of the development process component for undergraduate software engineering projects [96]. Song *et al.* applied process mining technology to common event logs of information systems for behavior pattern mining [97]. They created one input data for behavior pattern mining from event logs of five different information systems [97].

From the discussed existing studies, we observed that the data from software repositories is explored for control flow analysis [91][92][97][93] and organizational perspective [95] [96], using process mining techniques. Also only few of them focused on the ticket resolution process [95][92][93]. Therefore, different software repositories from different perspectives need to be analyzed to capture an integrated view of the software maintenance ticket resolution process and support improvements. In this thesis, we focused on applying process mining to various software repositories, such as ticketing system, peer code review system, and version control system, to improve the software maintenance ticket resolution process.

1.3 Thesis Overview

To identify the potential opportunities for improvement in software process management, we first conducted qualitative interviews and surveys of more than 40 managers in a large global IT company. The survey provided us with a list of 30 software process management challenges encountered by practitioners out of which around 10 correspond to ticket resolution process. This thesis addressed a few of the identified challenges pertaining to the software maintenance ticket resolution process. We studied different types of software maintenance tickets, that is, software bug tickets and IT support tickets. As identified from the survey, there is a need to capture process reality and identify the process inefficiencies. Hence, we proposed a framework to analyze the data for ticket resolution process from diverse perspectives, by applying process mining techniques. Using the proposed framework, we discovered the process model that captured the control flow, timing and frequency information about activities performed during ticket resolution. From the discovered process model, we then



Aim
Identify potential process challenges and inefficiencies
Methodology
Survey of practitioners in a large global IT company
Contribution
✓ Validated list of 30 process management challenges
Chapter 2
Analyze software maintenance ticket resolution to identify inefficiencies
• Analyze ticket data using proposed process mining framework
• Conduct a series of case studies
✓ Granular understanding of actual process
✓ Process cube notion for comparison between process variants
✓ Detect deviations between defined and runtime process
✓ Identify inefficiencies e.g. loops, back-forth, reopen, timing issues
Chapter 3, 5
• Improve software maintenance process by mitigating inefficiencies
• Recommend suitable actions leveraging predictive analysis
✓ A machine learning based preemptive system and a detection system to reduce user input requests
✓ Automatically detect the potential bugs by mining execution logs
Chapter 4, 6

Figure 1-2: Thesis outline capturing the objective, methodology, and contribution.

studied inefficiencies such as self-loops, back-forth, ticket reopen, timing issues, delay due to user input requests, and effort consumption. We also analyzed the degree of conformance between the designed and the runtime (discovered) process model. The data-driven insights helped to make process improvement decisions. For example, using the proposed framework on IT support ticket data for a large global IT company we found that around 57% of the tickets had user input requests in the life cycle, causing significant delays in user-experienced resolution time. Therefore, we proposed a machine learning based-system that preempts a user at the time of ticket submission with an average accuracy of around 94% to provide additional information that the analyst is likely to ask, thus mitigating delays due to later user input requests.

One of the problems identified from the survey is in-depth understanding of actual process to effectively identify the opportunities for ticket resolution process improvement. We discovered the detailed process model for ticket resolution process, using the information present in the comments. To model the detailed process, we extracted topical phrases (keyphrases) from the unstructured data (comments) using an unsupervised graph-based approach. The keyphrases were then integrated into the event log, which got reflected in the discovered process model. This provided insights that could not be obtained solely from the structured data (i.e., activities), and these insights could be used to perform the ticket resolution process more efficiently.

Some code changes are made to resolve a ticket. This change can lead to anomalies, such as regression bugs. We aimed to detect whether ticket resolution caused some anomalous behavior so as to reduce the post-release bugs, one of the important challenges identified from the survey. To achieve this, we proposed an approach to discover an execution behavior model for the deployed and the new version using the execution logs that is, runtime print statements. Differences between the two models were then identified, which allowed programmers to detect anomalous behavior changes, that is, not consistent with code changes thereby identifying potential bugs that might have been introduced during code change.

We applied the proposed framework and solution approaches on a series of case studies on data sets of commercial and open source projects. Through the aforementioned contributions, we explored the potential of applying process mining using various data sources to improve various aspects of the software maintenance ticket resolution process. Such analysis usually focuses on identifying the inefficiencies, but as we observed in the thesis, it can also lead to automation opportunities to make the ticket resolution process more efficient.

This thesis focused on *improving maintenance ticket resolution process using process mining*. Figure 1-2 presents the overall approach for the thesis. Overview of the thesis is as follows:

- **Chapter 2: Identify opportunities for process improvement using process mining**

  In the first stage of the research, we conducted a survey and interview study with more than 40 managers to identify the software process-related challenges that the practitioners would like to be addressed using process mining. From the survey, we identified 30 challenges out of which around 10 pertained to software maintenance. We attempted to address a few of the identified challenges applicable to software maintenance ticket resolution process, an important part of software maintenance. This work is discussed in Chapter 2 of the thesis and published in the International Conference on Mining Software Repositories (MSR) conference.

- **Chapter 3: Analyzing ticket resolution using process mining**

  Ticket resolution is an important part of software maintenance process. As identified from the survey, analyzing the data generated during the ticket resolution process is necessary to capture process reality and identify the process inefficiencies. We studied different types of software maintenance tickets, that is, software bug tickets and IT support tickets. We proposed a multistep framework for analyzing software repositories for ticket resolution from diverse perspectives, by using process mining. Using a multistep framework, we discovered the process model that captured the control flow, timing, and frequency information about events. We then studied inefficiencies, such as self-loops, back-forth, ticket reopen, timing issues such as delays due to user input requests, and effort consumption. We also analyzed the degree of conformance between the designed and the runtime (discovered) process model. We conducted a series of case studies on the open-source Firefox browser, Core project, open-source Google Chromium project, and IT support ticket data for a large global IT company. The data on tickets were obtained from Issue Tracking System (ITS) for the project (e.g., Bugzilla). We also used repositories for the peer code review system and version control system, where available. The proposed multiperspective process mining framework and the case studies to evaluate the proposed approach are presented in Chapter 3 of the thesis, and is published in Innovations in Software Engineering Conference (ISEC), Asia-Pacific Software Engineering Conference (APSEC), and MSR.

- **Chapter 4: Reducing user input requests in ticket resolution process**

  In an industrial context, a ticket is required to be resolved in the defined service-level resolution time, measured using the service-level clock. Failure to meet this requirement leads to a penalty on the service provider. After a ticket is assigned to an analyst (a person responsible for servicing the tickets), they can ask for user inputs to resolve the ticket. When user input is requested, the service-level clock stops to prevent spurious penalty on the service provider. However, this waiting time adds to the user-experienced resolution time and degrades the user experience. By applying the proposed multiperspective process mining framework on the tickets of a large global IT company, we found that around 57% of the tickets had user input requests in the life cycle. The user input requests caused user-experienced resolution time to be almost twice as long as the measured service resolution time. We observed that the user input requests were broadly of two types: real, seeking information from the user to process the ticket; and tactical, when no information is asked but the user input request is raised merely to pause the service-level clock. We proposed a machine learning-based system that preempts a user at the time of ticket submission to provide additional information that the analyst is likely to ask, thus reducing real user input requests. We also proposed a rule-based detection system to identify tactical user input requests. A case study was performed on the same IT support ticket data to illustrate the usefulness of the proposed preemptive and detection model. This work is discussed in Chapter 4 and published in the Empirical Software Engineering journal.

- **Chapter 5: Analyzing Comments in Ticket Resolution Process to Capture Underlying Process Interactions**

  Process mining uses largely structured data, namely event logs, and does not leverage the rich information from unstructured data such as comments and emails. One of the problems identified during the survey was to facilitate granular understanding of the process to support improvement decisions. In this chapter, we extracted topical phrases (key phrases) from the unstructured data using an unsupervised graph-based approach. The key phrases were then integrated into the event log, which subsequently got reflected in the discovered process model. This provided insights that could not be obtained solely from structured data, which can be used to identify process improvement opportunities. To evaluate the usefulness of the approach, we conducted a case study on the ticket data of a large global IT company. This work is discussed in Chapter 5 and is under submission.

- **Chapter 6: Identifying changes in runtime behavior of a new release to**

facilitate anomaly detection

To resolve a ticket, some code changes are made that could lead to an anomaly such as regression bugs. In this chapter, we proposed an approach to discover execution behavior model for the deployed and the new version using the execution logs (which contained outputs of all the print statements along with related information such as time, thread ID, statement number, and so on). Differences between the two models were then identified and presented graphically as regions within the discovered behavior model. This allowed programmers to identify anomalous behavior changes that were not consistent with code changes, thereby identifying potential bugs. To evaluate the proposed approach, we conducted case studies on Nutch (open-source application), and an industrial application. This work is presented in Chapter 6 and is published at the International Conference on Service-Oriented Computing (ICSOC).

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Identifying Opportunities for Software Process Improvement Using Process Mining

We aimed at identifying the software process−related challenges that the community in practice would like to be addressed using novel applications of process mining. To achieve this, we conducted a two-phase survey and interviews with managers at Infosys Limited[1], a large, global, software company employing more than 170000 software professionals. It is a CMM 5 company with well−defined processes in place. We first conducted a survey to identify the software process management challenges, followed by a second survey to analyze the importance of the identified challenges.

Several studies aimed at understanding the needs and questions of developers by means of surveys and interviews. The study that is most closely related to the work presented in this chapter is that by Begel *et al.* which cataloged 145 questions grouped into 12 categories that software engineers would like data scientists to investigate [98]. The study also identified the most important and most unwise problems based on rating survey with 607 Microsoft engineers [98]. Phillips *et al.* interviewed 7 release managers at a large software company to identify the information required for integration decisions when releasing software, and organized around 10 key factors [99]. Fritz and Murphy interviewed 11 developers to identify

[1]http://www.infosys.com/

the questions they would like to ask but the support to integrate different kinds of project information was lacking in the study [100]. LaTozo *et al.* proposed 19 problems from their own experience as software developers and discussed them with other developers to determine the seriousness of each problem [101]. Sillito *et al.* categorized 44 different kinds of questions that developers ask about a code base during a change task [102].

From our survey, we identified 30 process management challenges spanning across different phases of the software development lifecycle (SDLC). Of the 30 process management challenges, more than 15 corresponded to software maintenance. In this thesis, we attempted to address a subset of the problems pertaining to software maintenance, specifically the ticket resolution process. This chapter describes the survey study and its findings. More details of this study are given in reference [103].

For reference and benchmarking, we have made surveys, collected responses and consolidated lists for both the phases publicly available[2].

2.1 Survey to Identify Process Management Challenges

In the first phase, we conducted an open-ended online survey and interviews to identify the process management challenges encountered by managers and the benefits of solving those challenges. The interviews were conducted only for the participants who volunteered for in-person discussion instead of filling the online survey. First we performed a pilot study to improve the survey, followed by final data collection. The collected responses were open card sorted to groups having similar responses, where each group was represented using a generic problem statement.

2.1.1 Pilot Study

We conducted a pilot study to improve the survey before sending it to the target audience. We included only the definition of process mining to quickly introduce process mining to the participants and set the context.

The initial question asked in the survey was as follows:

[2]https://github.com/Mining-multiple-repos-data/QualitativeStudy

Process Mining of software repositories involves extraction of useful information from event logs recorded by Information Systems (such as Bug Tracking System, Version Control System, SCMs, E-mails, Peer Code Review System) used during the SDLC. Suppose you are given an opportunity to benefit from the expertise of process mining specialists team, where the team can process mine the data stored in repositories during SDLC and solve problems related to software development and maintenance process.

Please list up to three problems or inefficiencies that you encounter during the software development process management and you would like the process mining team to solve for you. Also, mention the benefits you will have if the problem is solved.

The survey was given to three participants with management experience and we observed them while they filled the survey sheet. Two of them asked for some examples of applications of process mining as they found it difficult to understand the kind of problems that would meet the criteria. Although one participant directly filled in the survey mentioning the challenges such as high attrition rate and the difficulty in requirements gathering, these were not aligned with our objective. Therefore, we added the examples of process mining applications along with the definition to give the participants a better idea of the domain.

Based on the responses of the pilot study, we added the following examples to the initial survey question:

- **Performance (Time) Analysis:** Analyze inefficiencies such as most time−consuming activities (bottlenecks) in the process.

- **Process Compliance Checking:** Detect inconsistencies with the defined process and flag anomalies.

- **Control Flow Analysis:** Discover the actual process from the execution logs and analyze activity sequence to better understand the actual runtime process.

- **Organizational Analysis:** Analyze individuals, team coordination and interactions with the process.

Henceforth, the improved survey was sent to a sample of eligible professionals with project management experience. Our collaborator at Infosys explained that most professionals who were engaged in significant project management activities had titles such as Team Lead (TL), Project Manager (PM), Senior Project Manager (SPM), Group Project Manager (GPM), and Principal Technical Architect (PTA) etc. Therefore, we limited our attention to the people of this title at the Bangalore location where we were working (this work was done by the author as part of her internship at Infosys). Of approximately 500 eligible professionals, we randomly selected 300 and requested them to fill the survey (online or through an in-person discussion).

2.1.2 Participants and Data Collection

Participants for the survey were purposively sampled to ensure that they met the criterion of having project management experience. We specifically targeted participants from different software development and maintenance departments to capture the diverse perspectives of the practitioners involved in the management activities. Using the refined survey from the pilot study, we asked practitioners to solicit the challenges and inefficiencies that they would like the process mining team to analyze. Also, to check that our criteria for project management experience were being met, we inquired about their role, work experience and project management experience (in years).

Overall, 46 practitioners (out of 300 randomly selected participants) responded, of which 12 opted for an in-person discussion. The response rate was around 15%, which was comparable with the response rate of similar free-text surveys [98]. Of these 46 participants, around 26% were TLs, 33% were PMs, 28% were SPMs or Senior Technology Architect, and the remaining (approximately 10%) were others such as Delivery Managers, Group Leaders and Quality Managers. The total work experience of the respondents ranged from 8 to 21 years, and the project management experience ranged from 1 year to 14 years, with 25 participants having more than 5 years of project management experience. Four respondents chose not to mention their experiences.

Interview: We conducted a semi-structured interview with the 12 participants (out of 46) who opted for in-person discussion instead of filling the online survey. Using the survey questionnaire as the guide, the interviewer gave the interviewee an overview of process mining followed by the objective of the study. Most participants described a scenario along with the context that was noted by the interviewer. The survey responses from other participants also included long statements describing the scenario and benefits. Very few responses stated the inefficiencies directly. A major difference between the survey responses and interviews was in terms of context understanding. During the interview, the interviewer had the opportunity to probe and ask follow−up questions for better understanding. Conducting interviews and getting a detailed idea on perspectives discussed in the interview helped better interpret the other survey responses as well.

We gathered 130 items as a result of this survey and interview. In this survey, we focused on identifying the problems. For determining importance, we conducted a second survey about the importance of solving the generic problems formulated using 130 items. Inspired by the study by Begel *et al.* [98], we believed that having a separate survey for importance analysis could give a better indication for the importance (support) of each identified problem. This allowed keeping the first survey focused on identifying problems and the second one focused on analyzing the importance of the identified problems. We discarded 19 points from the survey responses as we believe that they could not be addressed by mining data from software repositories and hence were not aligned with the context of the study. For example, the following responses were filtered out:

- *"Lack of product culture in the services industry."*
- *"Clarity of nonfunctional requirements; migration projects do not have proper testing."*
- *"Impact elements are not easily defineable."*

A complete list of discarded responses is made publicly available[3]. Specifically, refer to *Phase-1.xlsx* in which the rows with value as "WI": for the column, *Problem Statement*, correspond to discarded responses.

[3]https://github.com/Mining-multiple-repos-data/QualitativeStudy

2.1.3 Analyzing Survey Data for Formulating Problem Statements

We observed that many points, though expressed differently, had essentially the similar underlying problem. On the basis of the similarity in the problems, we sorted the gathered items using open card sorting, that is, groups unknown at the outset [104][105]. Multiple iterations were performed for sorting until the author and her two colleagues were in agreement. As a result, valid 111 items were sorted into 30 groups in which each of them represented a different problem. We noticed that more than one problem was stated in some items, and thus included in multiple groups. A problem statement was formulated for each group, leading to 30 problem statements. The formulated problem statement was an abstract representation of the original grouped items because it did not include specific details of each item grouped together. For example,

For formulated problem statement: *During issue resolution, detection and analysis of PING-PONG patterns due to bug tossing between developers to reduce resolution time*, the following three responses (as stated by respondents) belonged to the group:

- *"Quite often an issue is reported as a defect. However, developers do not consider it to be a defect or state that it belongs to some other component. A lot of efforts go waste in this tossing around. Can we improve the process to reduce such cases?"*
- *"Unnecessary delays and blame game. Reporting of defects often leads to delay and blame game. Understand the patterns with cause to avoid such situations."*
- *"Ping-Pong patterns between various teams when an issue is reported."*

Similarly, we abstracted other groups using brief problem statements. Each statement had **task** (or **actual problem**) as the first component followed by **cause and benefit** as the second component. We intentionally structured the problem statements like this considering the second survey of the study because we wanted the problem to act as a trigger. For instance, in the aforementioned example, *detection and analysis of Ping-Pong patterns* is the **problem** and *due to bug tossing between developers to reduce resolution time* is **cause and benefit**. The original detailed items for every group are not presented here due to limited space, but are included in Appendix A. Also, we classified the formulated problem statements into various process mining categories and identified the possible benefits of addressing challenges from each category, which are reported in reference [103].

Next, we conducted a follow-up survey in which the objective was to identify the importance of the 30 generic problems identified through the first survey. Since the first survey focused on the variety of the identified challenges, this survey helped establish the importance (an indicator of support) and assess whether the identified problems were worth solving.

2.2 Survey to Determine the Importance of the Identified Challenges

The survey identified 30 problems with 5 options as follows:
[*Essential* | *Worthwhile* | *Unimportant* | *Unwise* | *I don't understand*],
Where the question asked was:

> *We have identified some process—related problems encountered by practitioners while managing IT projects. In your opinion, please indicate how important it is to have a process mining team solve this problem where the significance of each option is:*

- *Essential:* The problem poses many challenges and should be dealt with on a high priority.
- *Worthwhile:* The problem is important to solve and will help the managers.
- *Unimportant:* It is not worth solving the problem.
- *Unwise:* Team is discouraged to solve the problem.
- *I don't understand:* Problem statement is not clear or difficult to understand.

> *Mention any other process—related problems that you encounter and are not listed in the questionnaire.*

Inspired by a similar study [98] for identifying the importance of answering questions identified from the first survey, we decided to opt for the aforementioned options. These options met the need for both positive (*Essential* and *Worthwhile*) and negative (*Unimportant* and *Unwise*) scale with variation in intensity. As the problems were succinct, the participant might find it difficult to understand them without a detailed context. Therefore, the fifth option allowed expression of a problem not understood. As an introduction, we included only the definition of process mining in this survey as the focus was to validate the identified problems. We mentioned the significance of each option to avoid any differences in understanding. To ensure that the participants could mention any other challenges, if missing, we added a free text question in the end. Also, we asked the following basic details: *current role, total work experience and total project management experience.*

2.2.1 Participants and Data Collection

A consolidated list of 30 problems with 5 options was sent to 160 distinct randomly selected practitioners with the same characteristics of having project management experience and having handled teams. We sent the survey to distinct participants (i.e., from the remaining 200 professionals out of the initial set of 500), excluding the participants of the first survey, to mitigate the confirmation bias. To reduce the likelihood of confirmation bias, we did not mention that the problems were identified by surveying their colleagues. Also, we ensured that the question asked in the survey did not favor any problem statement. We received responses from 43 people out of 160 with a response rate of around 27%, which was almost double the response rate of the previous survey. One of the major reasons for the increased response rate was that the second survey was closed type where the respondent had to select one out of the given options. While answering all the questions was preferred, a respondent could answer as many as possible due to the large number of questions. We got 1262 responses instead of 1290 (43×30), indicating that only 28 (2%) were unanswered. The participants covered a broad spectrum in terms of role, type of projects, and experience. We had participants with total work experience ranging from 6 years up to 25 years, with a median of 12.5 years. Similarly, the project management experience ranged from 1 year to 15 years with a median project experience of 6 years, thus covering a wide spectrum.

Out of 1262 responses, very few (only 42) were *I don't understand*. Two reasons accounted for this: either the respondent did not have experience with that kind of project and therefore, found it difficult to understand it, or our summarization was not clear. For only one problem statement, nine respondents selected not understandable, which essentially signaled that the formulation required more clarity. For other statements, we noticed that at most three respondents selected the option *I don't understand*, which could be attributed to diversity in roles, experiences, and domain. *Essential and Worthwhile* were referred to as positive ($+ve$) responses, and *Unimportant and Unwise* were referred to as negative ($-ve$) responses. We observed that most of the responses (1088 out of remaining 1220) were positive and believed that the confirmation bias was minimal because we did not inform participants about the second survey that the problems were identified by conducting a survey/an interview with their colleagues [104] and asked the survey question such that no problem statement was favored. Some of the possible reasons for the negative response to a particular problem were as follows:

- The problem was already solved to a satisfactory level and was not worth spending resources in solving it further. It was based on a remark, *"We already have a solution for this,"* mentioned by a participant as a comment along with a question where *Unwise* was selected as an option.

- The participant had never faced that problem and felt it was not really a problem. For example, we got similar comments such as *"Some of the listed problems are project and domain specific,"* from a couple of respondents.

The problem about identifying the group of ACTIVE VS INACTIVE CONTRIBUTORS, GENERALIST VS SPECIALIST by analyzing the performance of individuals participating in the process received maximum (that is, 17) negative responses. We do not have specific comments from the respondents to explain a high number of negative responses to this particular problem. One possible explanation may be that the respondents believed that such performance-based classification could increase work pressure and competition within the team.

We received a few comments in the free text question. No new problem emerged as a

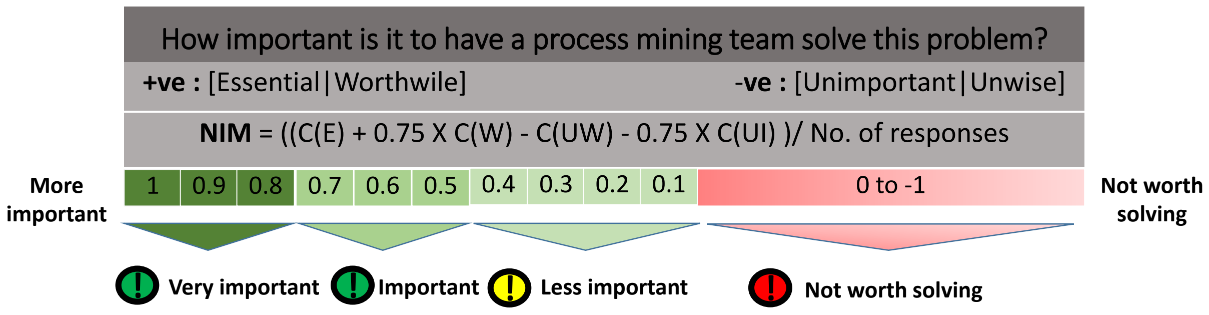

How important is it to have a process mining team solve this problem?
+ve : [Essential|Worthwile]
-ve : [Unimportant|Unwise]
NIM = ((C(E) + 0.75 X C(W) - C(UW) - 0.75 X C(UI))/ No. of responses
More important
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0 to -1
Not worth solving
Very important
Important
Less important
Not worth solving

Figure 2-1: Net Importance Analysis

result of those comments. Most of the comments were general remarks on the listed problems where the most common were the following:

- *"Need to identify the data required for solving these problems, thus ensuring we have a system in place to automatically fetch the required details."*
- *"Most of the identified problems are associated with the maintenance phase; can we apply process mining to improve the requirements phase, which is a perennial problem?"*

The second one helped us notice that only a few problems involved the requirement-gathering phase. One possible explanation could be that it was difficult to find event logs because requirements gathering was a communication-intensive phase resulting in the Software Requirement Specification document. Thus, the participants were refrained from soliciting requirement-phase challenges.

Also, we proposed the *Net Importance Metric* to measure the net importance of solving each problem using the count of positive/negative responses as discussed below.

2.2.2 Net Importance Analysis

We proposed a *Net Importance Metric* (NIM) to objectively measure the importance of solving each problem in which the counts of positive and negative responses were taken as input parameters (refer to Figure 2-1), that is, *NIM,*

$$\boxed{NIM = \frac{C(E) + 0.75 \times C(W) - C(UW) - 0.75 \times C(UI)}{C(E) + C(W) + C(UW) + C(UI) + C(DU)}}$$

where

$C(E)$ = Number of responses as *Essential*

$C(W)$ = Number of responses as *Worthwhile*

$C(UW)$ = Number of responses as *Unwise*

$C(UI)$ = Number of responses as *Unimportant*

$C(DU)$ = Number of responses as *I don't understand.*

Metric NIM measures the net importance of solving a problem based on responses from the participants. We used this metric to determine the joint effect of positive and negative responses. *Essential* expresses a stronger need to solve a problem as compared with *Worthwhile*. *Unwise* strongly discourages the team from spending efforts in solving a problem, while *Unimportant* reflects the same with a comparatively weaker intensity. Therefore, we used multiplier factors of $1, 0.75$, and $-1, -0.75$ to scale the intensity of positive and negative responses, respectively. The difference between $(+ve)$ and $(-ve)$ responses was normalized for comparison because all the problems did not have the same number of ratings.

As depicted in Figure 2-1, the value of NIM lies between -1 and 1. 1 corresponds to the *most important*, and 0 corresponds to the *less important*, that is, the relative importance of solving a problem increases with the increasing value of NIM. It can be attributed to the fact that some problems have more impact and need to be dealt with on a high priority. Although some problems exist, they do not require immediate attention. The value of NIM is less than 0 (indicated in red) for problems with more negative responses. Thus, solving these problems is not recommended from practitioners' points of view. In Table 2.1, the problems are arranged in the decreasing order of NIM. We observed that all the listed problems had NIM greater than 0. Thus, all were considered worth solving, although some were more important than others.

Effectively, all the listed problems were validated by experienced professionals. Practitioners believed that solving some problems was more important compared with others. Therefore, as a process mining specialist, one can make an informed choice on which problems to focus on and solve first.

2.3 Survey Results

Table 2.1 presents a list of formulated 30 process management challenges, each corresponding to a group identified from the survey and interviews. As determined from the follow−up importance survey, a count of both $+ve$ and $-ve$ responses for each problem statement is also presented in the table. The negative responses were significantly less, indicating none of the identified problems was absolutely irrelevant. The challenges were arranged in decreasing order of importance (as determined by the NIM metric). Any statement with one or more items pertaining to software maintenance is highlighted with $[M]$, indicating the problem statement belonged to the maintenance phase. We noticed that around 20 identified challenges were for the software maintenance process, of which 12 referred to the ticket resolution process in maintenance. In the rest of the thesis, we attempt to address some of the ticket resolution process challenges (as indicated by ♣ in Table 2.1) using process mining techniques. Some of the related problems which were about analysing ticket resolution process (such as $P1$, $P5$, $P15$, $P23$, $P24$ and $P27$) were grouped together and explored in chapter 3.

Table 2.1: List of problem statements formulated for each group along with the count of positive ($+ve$) and negative ($-ve$) responses. Statements with $[M]$ belong to the maintenance phase; the *italicized* ones are for the ticket resolution process; and ♣ are the ones we attempted to address.

ID	**Formulated Problem Statement**	**+ve**	**−ve**	**NIM**
$[M]$ ♣P1	*Identify BOTTLENECKS and inefficiencies causing a delay in the ticket resolution process to take remedial actions and have better estimation in future. [Chapter 3]*	42	0	0.92
$[M]$ ♣P2	*Enable early detection and PREVENTION OF DEFECTS instead of fixing them during the later stage by understanding patterns of escaped defects. [Chapter 6]*	41	0	0.91
P3	Avoid putting efforts on LESS SIGNIFICANT ACTIVITIES by identifying redundant or unnecessary steps of process.	41	1	0.89

P4	Automatic ADAPTATION OF PROCESS according to different project specifications, that is, design process based on knowledge of similar successful projects instead of selecting process only on the basis of experience.	43	0	0.85
[M] ♣P5	*Inspect REOPENED issues to identify the root cause and recommend verification for future issues based on learning from issues reopened in the past. [Chapter 3]*	41	1	0.84
[M] P6	Need for efficient TASK ALLOCATION mechanism by considering individuals' skills, interests, and expertise as well as team compatibility for better utilization of resources.	40	2	0.83
P7	Various approvals (such as managers' approval) are part of software development life cyle (SDLC) and need better management. Design a process for seamless approvals to reduce delays.	40	3	0.79
P8	Mechanism for CONTINUOUS PROCESS EVOLUTION based on best practices of individuals who exercise the process. Therefore, improve process by encouraging on-the-job learnings of people rather than dependence on process designers.	39	2	0.76
[M] P9	Improve effectiveness of CODE REVIEW PROCESS AND STANDARDIZATION by redesigning check list and updating code analyzers based on the defects reported during testing.	39	2	0.74
P10	Facilitate BETTER INTEGRATION between different silos by reconstructing the process, thus reducing rework happening due to differences in understanding.	36	3	0.71
[M] P11	Handle CHANGING TEAMS seamlessly by analyzing interaction pattern between team members and team dynamics.	38	2	0.70
P12	Design a technique to TRACE ADHERENCE WITH REQUIREMENTS and adapt process automatically with changing requirements.	36	3	0.69

P13	PEOPLE VS PROCESS: Identify which factor contributed to what extent toward the success and failure of the project.	39	4	0.69
[M] P14	Simplify tracking of the whole CODE REVIEW PROCESS to identify inefficiencies quickly.	37	4	0.67
[M] ♣P15	*During issue resolution, detection and analysis of PING-PONG patterns due to bug tossing between developers to reduce resolution time. [Chapter 3]*	36	3	0.67
P16	Improve PROJECT PLANNING AND ESTIMATION by complimenting it with the insights derived from event log mining of similar projects done in the past.	38	5	0.66
[M] ♣P17	*Investigate the LEAD TIME for issue resolution by analyzing issue resolution process from TIME PERSPECTIVE and thus increase timely resolutions. [Chapter 4]*	37	5	0.64
P18	Design more meaningful QUALITY METRICS by understanding run time process practices to precisely identify the scope of improvement.	35	5	0.63
[M] P19	Equip novices with the KNOWLEDGE OF EXPERIENCED PRACTITIONERS by associating the efficiency of adopted process with the experience of practitioners.	36	4	0.63
[M] ♣P20	*Facilitate in-depth understanding of point where things went wrong by deriving and understanding actual process at a MORE GRANULAR LEVEL. [Chapter 5]*	36	5	0.63
P21	Continuous check on SCHEDULE ADHERENCE is a complex task. Design an automated way to track and preempt if any deviations.	36	6	0.60
[M] P22	*Relate bugs with the ACTUAL STAGE OF INCEPTION by understanding issue resolution life cycle along with other relevant attributes.*	34	5	0.60

[M] ♣P23	*Uncover DEVIATIONS between the actual process followed by the team and the defined process, as well as their cause and impact on overall outcome and identify the set of people exhibiting more deviations. [Chapter 3]*	36	6	0.59
[M] ♣P24	*INTEGRATE MULTIPLE STANDALONE SYSTEMS used during SDLC to solve data and process redundancy challenges, and obtain a holistic view. [Chapter 3]*	34	5	0.57
[M] P25	Analyze code review life cycle to identify developers who are not reviewing their code properly before they submit it for external review and the deviations from defined checklist. It will help take corrective actions and reduce defects during testing phase.	33	7	0.53
[M] P26	Mechanism to manage and keep track of SVN check-ins process, that is, activity sequence for merging and branching as it is very important and can help take informed decisions.	27	5	0.49
[M] ♣P27	*Capture the ACTUAL STATUS (reality) of project or any task by discovering run–time process from event logs instead of current manual practice. [Chapter 3, and 4]*	31	8	0.48
[M] ♣P28	*Trace the complete flow and understand WHICH ISSUE LEADS TO WHICH CODE CHANGE by analyzing event logs for issue resolution in combination with the code modified in VCS. [Chapter 3]*	31	9	0.45
[M] ♣P29	*Perform COMPARATIVE ANALYSIS OF TICKETS along dimensions such as component, owner (analyst), reporter, type such as performance, regression and security, final resolution such as duplicate, invalid and fixed, and turnaround time to derive useful insights for improvement [106].*	32	10	0.44

[M] P30	Identify the group of ACTIVE VS INACTIVE CONTRIBUTORS, GENERALIST VS SPECIALIST by analyzing performance of individuals participating in the process.	24	17	0.14

Some of the identified problems have already been studied by researchers, such as $P3$ [107], $P5$ [36][56], and $P6$ [108]. There are many problems that have not been addressed sufficiently. Based on the responses from the survey, we believe that addressing the remaining problems will be useful for practitioners.

2.4 Threats to Validity

In this study, we surveyed and interviewed participants performing managerial roles for diverse project types such as development projects, maintenance projects, and support. Nevertheless, they worked in the same organization using similar processes and guidelines. Though having diverse participants from the same organization enabled deeply exploring experiences from several perspectives, organizational culture might create a bias because both the surveys were conducted in the same organization. Although the organization was huge and CMM 5, the problems encountered by its managers might not be as important to managers of other companies with different organizational cultures. As the questions identified covered a vast spectrum of process mining types, we expected them to reflect problems encountered by managers, which the team of process mining specialists could solve.

We included examples to illustrate process mining applications in the first survey. They shaped the thoughts of respondents and made them think in the direction of similar challenges. However, we preferred to include examples to ensure that relevant problems were mentioned by practitioners. This decision was inspired by the pilot study, which was conducted without examples and resulted in responses that were not aligned with our objective of identifying process managements challenges, which could be addressed by analyzing data. Not all challenges belonged to the categories included as example in the study.

Some problems were directly stated by respondents, while others were inferred from the long statements received in the responses. Informal discussion during the interview complimented the process of generic problem formulation, but the interview was conducted with a small set of respondents. During the second survey, importance was influenced by individual differences among the participants themselves.

We observed that most of the ratings for the second survey were positive, which might be due to confirmation bias. However, we tried to reduce the likelihood of confirmation bias by not telling the participants of the second survey that the problems were identified by conducting a survey/an interview with their colleagues. Also, no particular problem statement was favoured in the designed survey. Further, from the ratings of participants, we observed response distribution across various importance ratings and high negative responses for specific problems indicating a minimal confirmatory bias.

2.5 Summary

By conducting an online survey and interview study, we identified 30 different software process management challenges encountered by managers. In addition to this, we conducted a follow-up survey with distinct participants to validate the importance of solving identified challenges. To the best of our knowledge this is the most comprehensive catalog of challenges published to date that can be addressed by process mining of software repositories. Effectively, the result of the survey, that is, the list of identified challenges, was a contribution in itself that could provide an important input to researchers for selecting a problem. While process mining can be one of the approaches, the challenges may also be addressed using different techniques. The scope of the rest of the thesis is to use process mining by attempting to address a few of the identified challenges. We hope that other researchers will also address the other challenges.

In the remainder of the thesis, we focus on a few challenges related to the software maintenance phase. Many challenges (more than 15) were identified for this. Specifically, we analyzed the ticket resolution process as an important part of a software maintenance process, and 12 out of all identified problems referred to the ticket resolution process. We attempted to address a subset of these problems (around 10 as indicated by *clubsuit* in Table 2.1) by applying process mining techniques on logged data, which is the focus of the remaining thesis. We selected this subset because many problems were identified for the ticket resolution process and the data were available to illustrate the following:

- The problems identified in the survey ($P1$, $P5$, $P15$, $P23$, $P24$, $P27$, $P28$, $P29$, and $P30$) highlighted the need to analyze the data generated during the ticket resolution process to capture process reality and identify the process inefficiencies. We chose to address this issue cutting across many identified problems. In Chapter 3, we propose a framework to analyze the ticket resolution process from diverse perspectives by mining the logs from one or more information systems (software repositories). This includes the discovery of process model capturing control flow, timing and frequency information of events, and further study inefficiencies such as self-loops, back-forth (ping-pong), ticket reopen, timing issues, conformance of the discovered model against the designed model, and comparative analysis of the process along different dimensions.

- In $P17$, the practitioners highlight the need to investigate lead time for issue (ticket) resolution and thus reduce the delays in ticket resolution. In Chapter 4, we analyze the ticket resolution process using the proposed framework and observe that there are many user input requests causing significant delays in the resolution time. Hence, we propose a machine learning-based system that preempts the user to provide required additional information at the time of ticket submission and thus mitigate delays due to later user input requests.

- Problem $P20$ highlights the need for understanding the process reality at a more granular level. Therefore, we explore unstructured data generated during process execution to derive insights that cannot be obtained solely from the structured data (as discussed in Chapter 5).

- Some code changes are made to resolve a ticket. This change can lead to anomalies, such as regression bugs. One of the top five problems identified in the survey is detecting bugs at an early stage, that is, $P2$. In Chapter 6, we investigate the usefulness of process mining to automatically detect bugs and inconsistencies in fast-evolving applications before the application was released. Similar to the ticket resolution, bug detection is one of the software maintenance processes. Bug detection is based on the run-time system logs as opposed to the information obtained from systems supporting software development (e.g., ticket tracker or version control system).

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Analyzing Ticket Resolution Using Process Mining

Ticket resolution is an important part of software maintenance process. Issue tracking systems (ITSs), such as Bugzilla[1] and Mantis[2], are applications where tickets are reported. ITS, peer code review (PCR) systems (such as Gerrit[3] and Rietveld[4]), and version control systems (VCS, such as SVN[5] and Mercurial[6]), are work flow management systems that jointly support the ticket reporting and resolution process in software maintenance. The free-libre/open-source software (FLOSS) projects, such as Google Android and Chromium, follow a process consisting of ticket reporting in ITS, patch submission for review in PCR systems, and committing source code change using VCS. The ticket reporting and resolution process spanning across multiple systems generates process and event log data that are archived in these software repositories.

Several projects have defined the ticket resolution process; however, the actual process being executed may be different from the defined ones. For improvement of the process, it is essential to identify the actual process being executed, which needs data generated during ticket resolution process execution. With a suitable representation using process

[1] http://www.bugzilla.org/
[2] http://www.mantisbt.org/
[3] http://code.google.com/p/gerrit/
[4] http://code.google.com/p/rietveld/
[5] http://subversion.apache.org/
[6] http://mercurial.selenic.com/

execution data, various types of analyses can be performed to identify process improvement opportunities.

The goals of this work were twofold:

- To integrate and transform the data available in various software repositories so that the actual ticket resolution process can be discovered using process mining and modeled as a control flow graph, annotated with execution information, that is, frequency and timing.

- To identify potential inefficiencies by analyzing the discovered ticket resolution process model along various perspectives. The analysis is based on the problems identified earlier in the survey (given in Chapter 2).

These two goals are discussed in the next sections. To illustrate the utility of the approach, we performed the case studies on large, long-lived open-source projects, such as Mozilla Firefox, Core, and Google Chromium, and ticket data of a large global IT company.

For the analysis, we focused specifically on exploring the following problems identified from the survey (refer to Table 2.1). These were chosen because they are related to the ticket resolution process and can be analyzed from the process execution logs. Research contribution toward addressing a problem is mentioned (in italics).

1. Problem 24 from Table 2.1: "INTEGRATE MULTIPLE STANDALONE SYSTEMS used during SDLC to solve data and process redundancy challenges, and obtain a holistic view."

   Presented a multistep framework to analyze single and multiple software repositories simultaneously for the ticket resolution process from diverse perspectives.

2. Problem 27 from Table 2.1: "Capture the ACTUAL STATUS (reality) of project or any task by discovering the runtime process from event logs instead of current manual practice."

   Investigated the application of process mining platforms, such as Disco and state-of-the-art algorithms, for discovering process maps from event logs and thus understanding the actual runtime process (reality).

3. Problem 1, 5, and 15 from Table 2.1: "Identify BOTTLENECKS and inefficiencies causing delay in the ticket resolution process to take remedial actions and have better estimation in future," "Inspect REOPENED issues to identify the root cause and recommend verification for future issues based on learning from issues reopened in the past," "During issue resolution, detection and analysis of PING-PONG patterns due to bug tossing between developers to reduce resolution time".

   Multiperspective analysis of the discovered process model includes bottleneck identification, reopen analysis, loop and back-forth analysis, and anti-patterns.

4. Problem 23 from Table 2.1: "Uncover DEVIATIONS between the actual process followed by the team and the defined process, their cause, and impact on the overall outcome, and identify the set of people exhibiting more deviations."

   Proposed a generic algorithm for quantitatively measuring the compliance between the design time and the runtime process model for the ticket resolution process.

5. Problem 30 from Table 2.1: "Identify the group of ACTIVE VS INACTIVE CONTRIBUTORS, GENERALIST VS SPECIALIST by analyzing the performance of individuals participating in the process."

   Analyzed from an organizational perspective to extract team-based interaction patterns, visualization, and metrics such as handover of work, subcontracting, working together (joint cases), and joint activities.

3.1 Discovering Ticket Resolution Process

The proposed approach (Fig. 3-1) for analyzing the ticket resolution process included mainly these steps: (1) data extraction, integration, and data transformation, and (2) process discovery, followed by multiperspective process analysis. The analysis of discovered process model from multiple perspectives (as discussed in Section 3.2), such as time (bottle-neck identification, delay due to user input requests), control flow (such as loop, back-forth, anti-patterns), conformance, and case (reopen analysis), is referred to as 'multi' perspective.

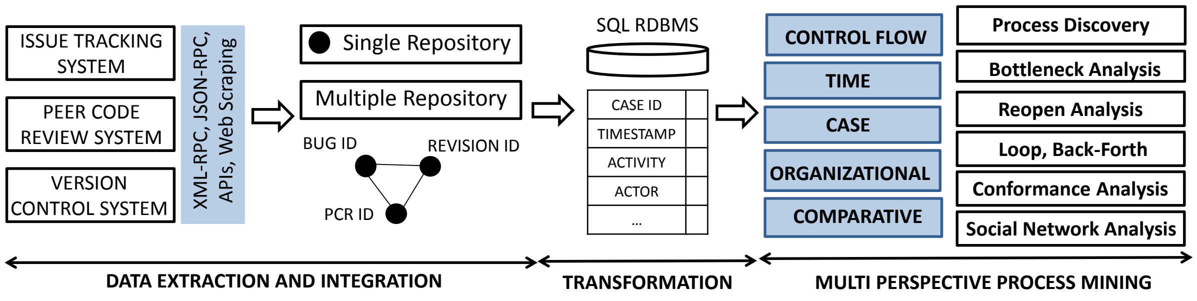


Figure 3-1: Framework for discovering the ticket resolution process using process mining and analyzing it from multiple perspectives.

3.1.1 Data Extraction, Integration and Transformation

We extracted data from software repositories, such as ITS, PCR system, and VCS, using APIs or web scraping. However, all the data were not available in a single well-structured data source. The event data were scattered over multiple sources, and efforts were needed to integrate multiple information systems for end-to-end process analysis. For example, multiple information systems (e.g., ITS, PCR system, and VCS) were not linked to each other explicitly, and data formats were different. Also, software repositories do not typically store data in a process-oriented manner, making the extraction of relevant data more difficult. Therefore, data extraction is driven by questions rather than the availability of lots of data [95].

One of the major challenges in software process mining is producing a log conforming to the input format of process mining tool [95]. Therefore, prior to using process mining for process model discovery, an event log should be generated using the data from repositories. An event log is generated with the following attributes: case identity document (ID), activity, timestamp, and resource (actor) and contextual information (based on the analysis to be performed). The following activities are performed during transformation:

- *Case ID selection:* Case ID associates all activities pertaining to the same case (*ticket id*) to analyze the ticket life cycle. However, integrating multiple information system is not trivial [95] because no common ID exists to explicitly link them. For instance, ticket in the ITS is identified by a ticket ID, patch for the ticket in the PCR by a patch ID, and commit in the VCS by a revision ID. Therefore, we need to identify all the patches in the PCR for a ticket and the corresponding commits in the VCS, and then represent all the events spanning across different information systems using a consistent case ID, say a ticket ID, and thus capture a ticket's end-to-end flow.

- *Activity identification and selection:* To generate an event log, we need to identify a list of important activities capturing the progression of a case (e.g., ticket resolution) during its life cycle. Many activities are captured in the information system (such as ITS, PCR, and VCS), but they may not be relevant for the analysis to be performed. The selection of the activities is driven by the analysis to be performed. Activities not relevant for the analysis need to be removed from the event log to avoid unnecessary complexity in the discovered process model. The list of selected activities can be confirmed with the manager(s) to ensure that we do not miss any activity that can have an impact on the analysis to be performed. For example, if one is not interested in analyzing the impact of priority or severity on the process life cycle, then it need not be captured as an activity in the discovered process model.

- *Extract implicit activities:* Some activities are not stored explicitly and need to be derived by parsing action comments or by linking multiple information systems.

- *Handle missing data:* Logged data may be incorrect or incomplete. Missing data are handled by imputation, that is, missing data are replaced with substituted values. For example, if time stamp is missing for some events, it is replaced with imputation.

- *Resolving time conflicts:* Sometimes data are captured in different time zones for a global organization and thus need to be handled carefully. Time stamps are converted to a consistent time zone on the basis of the geographical location (captured for the ticket) where the ticket is submitted and resolved.

3.1.2 Process Discovery

The event log obtained after transformation for all the tickets is then used for process mining. Many process mining tools are available, such as ProM (open source) [109] and Disco (commercial) [110], to discover process models. A discovered process model represents the control flow information of the activities, timing, and frequency of the involved activities, as recorded in the logs. The transformed event log derived from ticket resolution data is imported into Disco[7] to discover a process model depicting the process runtime behavior. The Disco miner is based on Fuzzy miner, a process mining algorithm that can be applied to less-structured processes, mostly for real-life environments [83]. Two fundamental metrics are used in the fuzzy miner: (1) significance and (2) correlation. *Significance* can be determined for both event classes (activities) and binary precedence relations over them (edges). It measures the relative importance of behavior, that is, the level of interest in events, or their occurrence after one another. *Correlation*, on the contrary, is only relevant for precedence relations over events. It can be measured in different ways, such as determining the overlap of data attributes associated with two events. More closely correlated events are assumed to share a large amount of their data. Based on these two metrics, the process is simplified as follows:

- Highly significant behavior is preserved, that is, contained in the simplified model.
- Less significant but highly correlated behavior is aggregated, that is, hidden in a cluster within the simplified model.
- Less significant and lowly correlated behavior is abstracted from, that is, removed from the simplified model.

A ticket ID is selected as a case ID for process model discovery to associate all activities pertaining to the same ticket (issue). A discovered process model is represented as a graph consisting of nodes and directed edges, where each node corresponds to an activity and directed edges depict the ordering relationship between the activities. Timing and frequency information is also labeled on the discovered process model for every activity and transition. Process discovery is the first step for an effective process analysis; therefore, we discovered a process in all the presented case studies.

[7] Availed academic license

3.2 Analyzing the Discovered Process Model from Multiple Perspectives

With the extracted process model, represented as a control flow graph, we can do various types of analysis. In this chapter, we mentioned a few, which we performed on the process model discovered for the ticket resolution process.

3.2.1 Bottleneck Identification

We identified a specific part within the discovered runtime process model, which was relatively time-consuming and reduced the overall performance of the end-to-end process, that is, bottleneck. Different processes can be discovered each corresponding to specific process instances that is, referred to as process variants. For instance, process model discovered for tickets with priority as high and low are two different process variants. We classified the bottlenecks into the following two classes on the basis of pervasiveness across the process variants:

1. *General bottleneck:* Time-consuming transitions in a stand-alone process without comparing across the variants.

2. *Exclusive bottleneck:* Transitions that were inefficient in some process variants compared with others. Even if some transitions are present in all process variants, the average time (after removing outliers) for the same transition might still significantly vary across the variants. This implies that the same activity is performed more efficiently in some process variants, suggesting the possibilities of improvement in others. We proposed the bottleneck ratio (*BNR*) to identify such instances:

$$\boxed{BNR = \frac{|(P_t - P'_t)|}{min(P_t, P'_t)}}$$

where P_t is the average time for transition t in process variant 1,
and P'_t is the average time for transition t in process variant 2.

BNR measures the extent of difference with respect to minimum average time taken for the same transition. If *BNR* ≥ 1, then the time taken by a process variant is at least double of minimum average time for the transition. We suggest using *BNR* to compare the two processes. However, if the number of process variants is more than two, then *Adjusted Box Plot* can be used [111]. An outlier in the box plot corresponds to the bottleneck, indicating a need for improvement.

From the discovered process, we identified bottlenecks in Case Study I (Section 3.3).

3.2.2 Reopen Analysis

Reopened bugs[8] (bug is reported as a ticket) increase the maintenance costs, degrade the overall user-perceived quality of the software, and lead to unnecessary rework by practitioners [36]. If a fair number of resolved bugs are reopened, instability is indicated in the software system [35]. Reopening of bugs has significant importance in open-source and commercial software maintenance projects. Understanding bug reopening is of significant interest to the practitioner's community as part of characterizing the quality of the bug-fixing process [35]. Therefore, the analysis of reopened bugs is expected to help process owners take preventive actions to minimize the reopening of bugs.

Shihab *et al.* [36] showed that the last status of the closed bug was an important influencing factor for the reopening. Depending on this status, the reasons for bug reopening could be as follows [35] [36]:

- *Wontfix/Invalid/Incomplete/Worksforme:* Clear steps to reproduce and additional information become available after the bug closure, allowing one to better understand the bug and determine its root cause.

- *Duplicate:* A bug accidentally marked duplicate due to similar symptoms or title matching with the existing bug without a 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 a boundary case missed during testing.

[8]Bug was once resolved, but the resolution was deemed incorrect.

- *Verified:* Incorrectness of the resolved bug verification process realized later or extra information becomes available, triggering the reopening of bug.

If many tickets are reopened, they can be mitigated using the reopen prediction solution as suggested in the existing studies [56][36]. Xia *et al.* [56] proposed an automatic, highly accurate predictor for reopened tickets (bugs), ReopenPredictor. It combined classifiers trained for three different sets of features, that is, description, comments, and metadata. Shihab *et al.* [36] built decision trees for the reopen prediction solution using the factors along four dimensions: (a) the work habits; (b) the bug report; (c) the bug fix; and (d) the team. They performed the case study on three open-source projects and observed that the factors for the reopen prediction solution varied with the project. Using these solutions, ticket reopening could be prevented, and thus the efficiency of the ticket resolution process could be improved.

We illustrated the application of process mining for reopening analysis in Case Study I (Section 3.3).

3.2.3 Loop and Back-Forth Analysis

Loop is an edge that starts and ends at the same state (activity). Similarly, a transition from state A to another state B and then back to state A is defined as one back-forth loop. Studying recurrent loops and back-forth is important as they represent repetition of the same activity, indicating potential inefficiency [110]. Detecting patterns where a ticket is passed repeatedly without any progress is difficult [112]. We detected loops and the back-forth phenomenon between a pair of activities (also referred to as a ping-pong pattern in the literature) in the discovered process model.

We illustrated the detection of self-loops and back-forth patterns from the discovered process in Case Study I (Section 3.3).

3.2.4 Anti-patterns

Anti-patterns represent erroneous interdependencies between activities of process models. Eid-Sabbagh *et al.* defined basic, composite, and nesting anti-patterns [113]. Analyzing anti-patterns is important for a process analyst to detect and eliminate erroneous transitions, thereby improving the overall process quality. In the context of ticket resolution process, we studied composite anti-patterns involving triggering and information flow between two or more process states.

We identified composite anti-patterns (undesired transitions) and the cause of their existence for our process model in Case Study II (Section 3.4).

Algorithm 1: evaluateFitnessMetric

```
Require: Event log, Adjacency matrix $A$
Ensure: Fitness measure
1: while not at the end of Event log do
2:    $\forall$ entries with Case ID $i$
3:    if $ts_C > ts_B > ts_A > ts_{Reported}$ then
4:       Trace, $T_i$ = {Reported,A,B,C} where A, B, C and Reported are the activities
         ordered by time stamp (ts).
5:    end if
6: end while
7: Count frequency of each unique trace $UT_i$ as $F_i$
8: while $\forall \ddot{u}nUT_i$ do
9:    $m$ := number of activities in unique trace $i$
10:   $p := UT_i[1]$, $q := UT_i[2]$, $V_i = 1$
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:
                    $\boxed{FM = \frac{\sum_{i=1}^{N}(F_i * V_i)}{\sum_{i=1}^{N}(F_i)}}$
    where $N$=Number of unique traces.
22: if FM<1 then
23:    inconsistentDetector(Eventlog, 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
 3:    for $j = state[2] : state[f]$ do
 4:       count=0;
 5:       while not end of Event log do
 6:          if i $\rightarrow$ j in Event log then
 7:             count++;
 8:          end if
 9:          Transition Frequency, $TF[i, j]$=count;
10:       end while
11:    end for
12: end for
13: Inconsistent Transition Frequency matrix, ITF:
        $\boxed{= (TF - TF \circ A)}$
14: Total Inconsistent Transition:
        $\boxed{= \sum(TF - TF \circ A)}$
15: Highest frequency of inconsistent transition:
        $\boxed{= max(TF - TF \circ A)}$
16: Most frequent inconsistent transition:
        $\boxed{= argmax(TF - TF \circ A)}$
```

3.2.5 Conformance Analysis

Sunindyo *et al.* used the hypothesis testing approach to verify the design model with a runtime event log from the bug history data [92]. As the verification was done for a few dimensions by proving/disproving the hypothesis, it did not capture the end-to-end process conformance [92]. We defined metrics to measure fitness (i.e., how a well-observed process complies with the control flow defined in the design time process model) and the point of inconsistency.

Algorithm 1 was used to evaluate the *Fitness metric*. The event log and adjacency matrix (with the row as the source activity and the column as the destination activity) having 1 in the cell if the transition is permitted, otherwise 0, were given as input. The trace (an array with activities from the event log in sequential order of their occurrence) was obtained for each case ID in steps 1 to 6 of Algorithm 1. We identified the unique traces and count frequency for each of them to optimize processing for conformance of trace. This optimization was very useful because most of the cases had the same trace, and hence we need not process conformance for each case individually. Each trace was verified with the adjacency matrix A for conformance in steps 8 to 18 of Algorithm 1. If it had all permitted transitions, then the valid bit V_i for the trace was assigned value 1, else 0. The *Fitness metric* was evaluated in step 21. If the value was < 1, then traces existed with some deviation from the defined model. Therefore, to detect the cause of inconsistency, *inconsistentDetector()* (Algorithm 2) was called, which took event log and adjacency matrix A as input and gave value for inconsistency metrics. *State* is an array of all possible activities. The frequency for the transition between each activity pair was counted from the event log and stored in the transition frequency (TF) matrix (refer to steps 2 to 12). The log was traversed *f(f-1)* times, where f is the total number of states (e.g., activities) and is typically in the order of tens and hence not computationally expensive. In equation 13, we obtained the inconsistent transition frequency matrix (ITF) by subtracting the product of TF and A from TF. The total inconsistent transitions were evaluated by adding the elements of ITF. The most-frequent inconsistent transition with its frequency was identified in steps 15 and 16 of Algorithm 2.

We evaluated metrics using Algorithm 1 and Algorithm 2 for projects in Case Study I (Section 3.3).

3.2.6 Delay due to User Input Requests

In an industrial context, a ticket is required to be resolved within the defined service-level resolution time (SLRT), measured using the service-level clock. The service-level clock is used to measure the service-level resolution time for every ticket and can have two states: pause (stops measuring the time) and resume (continues measuring the time). Failure to meet this requirement leads to a penalty on the service provider. After a ticket is assigned to an analyst (person responsible for servicing the tickets), they can ask for user inputs to resolve the ticket. When user input is requested, the service-level clock stops to prevent spurious penalty on the service provider. However, this waiting time adds to the user-experienced resolution time (URT) and degrades the user experience.

Analysts might require user input for various reasons such as incomplete or unclear information provided by the user, input information requirements not being defined clearly and completely, resolution of some tickets requiring specific information that is not intuitive to the user, and analysts not interpreting the user inputs correctly [114].

Further, given the paramount importance of honoring service-level agreement, asking user inputs is used as a sneaky way to achieve the service-level target of resolution time [114][115]. In general, analysts are guided to request user inputs only if they genuinely need information for ticket resolution. However, previous studies have suggested that cases of non-information-seeking user input requests also exist, which are handled merely for the sake of pausing the service-level clock and thus degrading the user experience [115]. Therefore, analyzing and reducing the user input requests in the tickets' life cycle are important to mitigate the delays incurred while waiting for user inputs.

We investigated delay due to user input requests in Case Study III (Section 3.5).

3.2.7 Organizational Analysis

Social networks were built using the event log data based on relations such as handover and subcontracting of work, joint activities, and joint cases [108].

Joint Cases: Interaction patterns between various organizational members were studied by plotting degree distribution graph and evaluating metric $M1$ for the strength of the interaction. Individuals working together (on the same case) should have a stronger relationship than the people rarely working together [108]. We proposed metric $M1$ for each vertex with degree d as:

$$\boxed{M1 = \frac{\sum_{i=1}^{N}(w_i)}{d * N}}$$

where N = number of vertices with degree d, and
w_i = sum of the weight of all the edges incident on vertex i, that is, weighted degree, having degree d.
Metric $M1$ measures the average strength of interaction for vertices (actors) having degree d with its neighbors. It helps to identify more social and active contributors.

Also, we calculated relative working together metric [108] for performer *p1* with respect to performer *p2* using complete log L as:

$$\boxed{p1 \bowtie_L p2 = \frac{\text{Number of cases p1 and p2 worked together}}{\text{Number of cases p1 participated}}} \tag{3.1}$$

A high value for the aforementioned equation indicates that performer *p1* has often worked with performer *p2*. Therefore, we can recommend a group of people to work on the same case using relative working together metric as it gives information about the strength of working jointly between different people.

Joint Activities: An adjacency matrix is created where a row represents a list of actors, and a column indicates activities performed by the actors. The value in each cell corresponds to the frequency of an activity performed by a particular actor. We represented the relation between actors and the activities they perform using a social graph where actors and activities

Attribute	**Value**
First issue creation date	Jan 1, 2012
Last issue creation date	Dec 31, 2012
Date of extraction	July 14, 2013
Total issues in 2012 for the complete Mozilla project	111,234
Issues not authorized for access	15,638
Issues without history	3,149
Total issues extracted for Firefox	12,234
Total issues extracted for Core	24,253
Total activities for Firefox (including Reported)	40,233
Total activities for core (including Reported)	88,396

Table 3.1: Experimental dataset details (open-source Mozilla project)

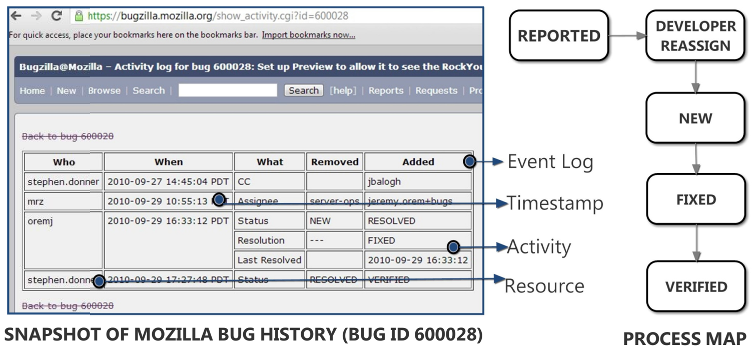
were represented by a node. If an actor performs an activity, then a weighted edge was drawn between the actor and the activity. It helps to identify a group of *generalists* and *specialists* for efficient task allocation.

Handover: Handover of work is based on the idea that two actors are related if a case is passed from one to another [108]. We consider only direct succession as handover; that is, an activity executed by actor $p1$ is consecutively followed by an activity executed by actor $p2$. Also, multiple transfers between the same actors (same instance) are counted and added to frequency. We identified and represented the handover dependency between different actors using a graph.

Subcontracting: If actor $p1$ performs an activity followed by $p2$ then again $p1$, it is considered as the subcontracting of work to $p2$ by $p1$ (directed edge from $p1$ to $p2$) [108]. We studied cases with only one activity in-between two activities performed by the same performer, that is, direct succession for the subcontract. We considered multiple occurrences of the same instance. Frequent subcontracting between different performers shows that they are more related to each other as work is subcontracted between them.

We used *Gephi*[9], an interactive visualization platform, to visualize the social network graphs. We illustrated social network analysis for Case Study II (Section 3.4).

[9]https://gephi.org/


https://bugzilla.mozilla.org/show_activity.cgi?id=600028

For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

Bugzilla@Mozilla – Activity log for bug 600028: Set up Preview to allow it to see the RockYou

Home | New | Browse | Search | Search [help] | Reports | Requests | Pro

~~Back to bug 600028~~

Who	When	What	Removed	Added
stephen.donner	2010-09-27 14:45:04 PDT	CC		jbalogh
mrz	2010-09-29 10:55:13 PDT	Assignee	server-ops	jeremy.orem+bugs
oremj	2010-09-29 16:33:12 PDT	Status	NEW	RESOLVED
		Resolution	---	FIXED
		Last Resolved		2010-09-29 16:33:12
stephen.donner	2010-09-29 17:27:48 PDT	Status	RESOLVED	VERIFIED

~~Back to bug 600028~~

Figure 3-2: Snapshot of the Mozilla bug report history (event log) and the corresponding process map.

3.3 Case Study I: Analyzing the Bugzilla Issue Tracking System

We conducted experiments on ITS data from Firefox and Core, two open-source subprojects of Mozilla, using *Bugzilla* ITS. Table 3.1 displays the experimental dataset details. We chose Firefox and Core for our analysis as these projects were large, complex, and long lived. Firefox is Mozilla's flagship software product, which is one of the most-used web browsers for Windows, MAC, and Linux. Core includes shared components used by Firefox and other Mozilla software, including handling of web content, such as Gecko, HTML, CSS, layout, DOM, scripts, images, and networking. The Bugzilla ITS ticket data for Firefox and Core projects are publicly available, and hence our results can be replicated and used for comparison by other researchers.

3.3.1 Data Extraction and Transformation

We extracted the bug report history (Fig. 3-2) using Bugzilla APIs (through XML-RPC or JSON-RPC interface). When the bug report history is transformed, it serves as the process event log generated by Bugzilla ITS.

We extracted *Status*, *Resolution* (only for closed), *Assignee*, *QA Contact*, and *Component* from the bug history to generate the event log. For open bugs, status can be *New*, *Unconfirmed*, *Assigned*, and *Reopened*, which are captured as *activities* in the event log. For closed bugs, if the ticket is verified, then the status is *Verified*, and the resolutions that can be *Fixed*, *Invalid*, *Wontfix*, *Duplicate*, *Worksforme*, and *Incomplete* are captured as activities[10]. *Assignee*, *QA Contact*, and *Component* assignments are recorded as *Dev-reassign*, *QA-reassign*, and *Comp-reassign* activities, respectively. The selection of these fields is driven by the analysis to be performed. We believe that in our case the aforementioned activities were sufficient for good characterization of bug's control flow, and performed the required analysis.

We obtained timestamp corresponding to an activity from the *when* field of bug history, as shown in Figure 3-2, to order the activities in the sequence of their actual execution (while generating process map via Disco). The performer of the activity was treated as a resource and captured from the *who* field of the bug history. We addressed issues such as missing data and same time conflicts as discussed in Subsection 3.1.1.

3.3.2 Process Discovery

We imported preprocessed data into Disco to discover the process model for Core and Firefox. Bug ID (ticket ID) was selected as the case ID to associate all activities pertaining to the same ticket, and hence we could visualize the life cycle of a ticket. We had 15 nodes, each corresponding to an activity in the process map for both the projects. A transition is a directed arrow between two nodes. The process map obtained was fairly complex with 160 and 156 unique transitions (including infrequent transitions) for Core and Firefox, respectively. For clarity, the process map with main transitions is shown in Figure 3-3 for Firefox. The main transitions were determined on the basis of significance and correlation [83], that is, the transitions with high significance and high correlation. The label of the edges and nodes indicated the absolute frequency of transition. The shade and thickness corresponded to the frequency, with more frequent being dark and less frequent being light.

Table 3.2 shows that a good percentage of cases had *Dev-reassign* (developer reassignment) and *Comp-reassign* (component reassignment) events in the life cycle. Also, the *QA-reassign* occurred quite often. Specifically, the *Dev-reassign* was much higher for Core than for Firefox. Therefore, efforts were needed to minimize reassignments [116]. A bug

[10]https://bugzilla.mozilla.org/page.cgi?id=fields.html#status

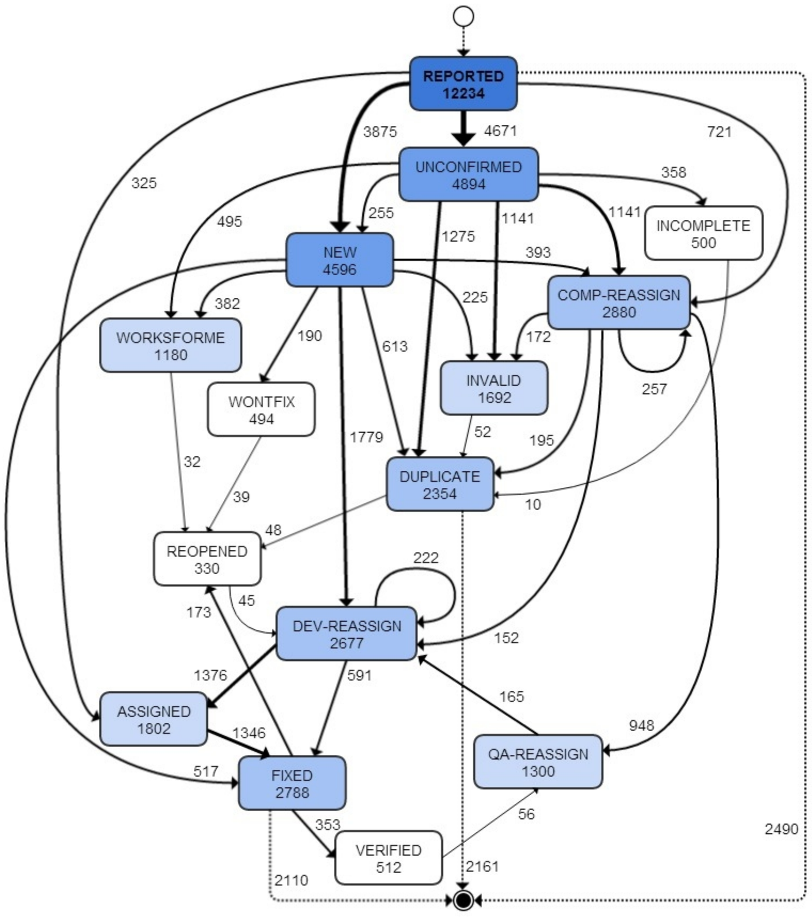


Figure 3-3: Process model with main transitions for Firefox. Edges and nodes are labeled with absolute frequency; the thickness of edge and shade of node correspond to absolute frequency.

once resolved should be verified; however, only around 4% of bugs were verified by the QA manager for both the projects, which was significantly less than the resolved bugs. This necessitated the need to uncover the reasons for infrequent verification and address them.

For activities belonging to the closed resolution, we observed that Core had almost double chance of getting a bug *Fixed*, which was the recommended situation. However, for Firefox, a comparatively large number of bugs were marked *Duplicate*, *Invalid*, and *Worksforme*, which meant a lot of practitioners' time was wasted in addressing the issues that were less useful for the overall product quality improvement. Therefore, some actions were needed to reduce the presence of such bugs; for instance, a better search mechanism before reporting a bug or a "Duplicate bug" warning by automated search at the time of bug submission to avoid

Activity	**Firefox**	**Core**
Reported	12234 (30.40)	24253 (27.43)
New	4596 (11.42)	15267 (17.27)
Fixed	2788 (6.92)	12573 (14.22)
Dev-reassign	2677 (6.65)	11471 (12.97)
Comp-reassign	2880 (7.15)	5843 (6.61)
Assigned	1802 (4.47)	5121 (5.79)
QA-reassign	1300 (3.23)	3095 (3.50)
Unconfirmed	4894 (12.16)	2893 (3.27)
Duplicate	2354 (5.85)	2319 (2.62)
Works for me	1180 (2.93)	1599 (1.80)
Verified	512 (1.27)	1081 (1.22)
Invalid	1692 (4.20)	1075 (1.21)
Reopened	330 (0.82)	1045 (1.18)
Wontfix	494 (1.22)	587 (0.66)
Incomplete	500 (1.24)	174 (0.19)
Total	40,233	88,396

Table 3.2: Absolute frequency of activities

duplicates [24][25], a more clear definition on what should be reported as bug to reduce invalid bugs, and the need for a more clear description of bug for better understanding.

Unique Traces: The complete sequence of activities for a case life cycle, that is, from the first activity (embarking the start of the life cycle) till the end is referred to as *trace* for a case (here, ticket). Different tickets can have different sets of transitions, and hence, different traces. For instance, a trace with a sequence A $\rightarrow$ B $\rightarrow$ C is different from that of A $\rightarrow$ B $\rightarrow$ B $\rightarrow$ C because the second case has a loop that was not present in the first case. Only closed tickets were used for the experiment. Effectively, 1164 and 622 unique traces existed for Core and Firefox, respectively. However, the frequency distribution of the traces, that is, the number of cases with a particular trace, was skewed. As a result, 80% of the cases for Core and Firefox had traces from one of the most frequent 2% traces (traces with at least 50 cases) and 3% traces (traces with at least 29 cases), respectively. Therefore, while the majority of the tickets had the same resolution life cycle, there were tickets with infrequent traces. Frequent traces are good automation opportunities, and the infrequent ones are interesting for outlier behavior investigation.

	Firefox			Core		
	Dev	*Comp*	*QA*	*Dev*	*Comp*	*QA*
Avg	50.25	12.67	13.72	32.35	12.17	13.88
Med	9.76	0.22	0.35	5.36	0.17	0.78
Min	8 s	9 s	4 s	7 s	12 s	4 s
1Q	12.96h	18.72m	0.99m	12.24h	21.6m	10.37m
3Q	65.97	2.60	19.40	28.70	2.37	6.97
Max	437.85	334.88	133.05	546.70	450.51	161.14
SD	84.84	41.49	28.30	67.25	43.40	34.53

Table 3.3: Loop duration analysis (default unit is days)

3.3.3 Analyzing the Discovered Process Model

Loop and Back-Forth Analysis: The process model obtained for Core and Firefox had a higher frequency of loop for *Comp-reassign* (component reassignment), *Dev-reassign* (developer reassignment), and *QA-reassign* (QA manager reassignment) as observed in Table 3.4. The first entry in Table 3.4 corresponds to Core and the second to Firefox. The higher number of loops for *Dev-reassign* (1296/222) and *Comp-reassign* (471/257) can be attributed to the following reasons: it was not easy to decide on the component to which the ticket pertains and the person to whom the ticket should be assigned. The number of loops was more than one in some cases; therefore, a lot of time was wasted in making a decision for the right assignment and component identification. This was an undesired delay, which was as high as 50.25 days and 30.35 days (as shown in Table 3.3) for Firefox and Core, respectively, in the case of *Dev-reassign*.

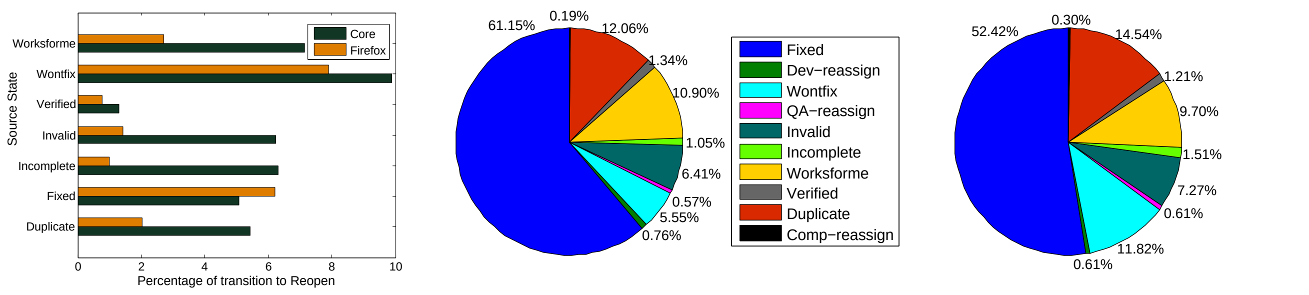
Each cell of Table 3.4 matrix (except for the diagonal values) shows the frequency of back-forth between activity pairs (14 X 14 matrix in which the row activity is state *A* and the column activity is state *B* of the back-forth). We noticed that *Unconfirmed*, *Comp-reassign* (component reassignment), *Dev-reassign* (developer reassignment), *QA-reassign* (QA manager reassignment), and *Reopen* were activities frequently involved in the back-forth pattern. Our experimental results revealed that a *Fixed* ticket being reopened and then again resolved as *Fixed* was very frequent (refer to Table 3.4: 466 and 114 times for Core and Firefox, respectively), indicating regression bug or disagreement of reporter with the fix. A back-forth loop phenomenon involving states such as *Invalid*, *Worksforme*, *Duplicate*, and *Wontfix* with *Reopened* shows disagreement (can be an error in judgment or a classification task that is

Activity	*Unconfirmed*	*New*	*Comp-reassign*	*Dev-reassign*	*QA-reassign*	*Assigned*	*Invalid*	*Reopen*	*WorksForMe*	*Incomplete*	*Won't Fix*	*Duplicate*	*Fixed*	*Verified*
Unconfirmed		$\frac{3}{5}$					$\frac{22}{53}$		$\frac{3}{22}$	$\frac{5}{8}$	$\frac{1}{16}$	$\frac{13}{35}$	$\frac{}{2}$	
New	$\frac{13}{5}$					$\frac{2}{1}$								
Comp-reassign	$\frac{}{2}$	$\frac{75}{17}$	$\frac{471}{257}$	$\frac{66}{4}$	$\frac{245}{90}$	$\frac{}{1}$	$\frac{6}{7}$		$\frac{3}{3}$	$\frac{1}{}$	$\frac{2}{1}$	$\frac{7}{4}$	$\frac{6}{2}$	
Dev-reassign	$\frac{3}{1}$	$\frac{66}{21}$	$\frac{92}{17}$	$\frac{1296}{222}$	$\frac{21}{3}$	$\frac{172}{99}$		$\frac{2}{}$				$\frac{2}{2}$	$\frac{9}{1}$	
QA-reassign		$\frac{1}{}$	$\frac{192}{65}$	$\frac{13}{1}$	$\frac{88}{29}$	$\frac{8}{2}$						$\frac{1}{}$	$\frac{5}{}$	$\frac{3}{}$
Assigned		$\frac{16}{7}$												
Invalid	$\frac{13}{22}$							$\frac{10}{3}$	1		$\frac{}{1}$			
Reopen							$\frac{5}{2}$		$\frac{2}{2}$		$\frac{5}{7}$	$\frac{4}{1}$	$\frac{52}{16}$	
WorksForMe	$\frac{2}{9}$							$\frac{17}{8}$		$\frac{3}{1}$			$\frac{2}{}$	
Incomplete	$\frac{4}{4}$							1			$\frac{}{1}$		$\frac{1}{}$	
Won't Fix	$\frac{4}{15}$							$\frac{9}{12}$				$\frac{}{1}$		
Duplicate	$\frac{8}{25}$						$\frac{1}{1}$	$\frac{13}{4}$	$\frac{}{1}$		$\frac{}{1}$		$\frac{1}{1}$	
Fixed	$\frac{7}{3}$							$\frac{466}{114}$	$\frac{6}{}$					
Verified														$\frac{9}{3}$

Table 3.4: Loop and Back-forth confusion matrix where numerator corresponds to Core and denominator corresponds to Firefox

nontrivial) between developers. One explanation was that the team members might not be convinced with the initial decision and hence reopened but later closed the ticket (after resolving the issue), resulting in loss of productivity and increase in the mean time to repair. Back-forth between *Unconfirmed* and any resolution was permitted according to the predefined Bugzilla life cycle, and we also observed it in the as-is process.

Reopen Analysis: Figure 3-4a shows the percentage of bugs getting reopened given resolution with a specific status. The absolute frequency for each status is presented in Table 3.2. The interpretations derived from Figure 3-4 are as follows:



(a) Percentage of issues in given state with transition to Reopen

(b) Distribution of activities with transition to Reopen for Core

(c) Distribution of activities with transition to Reopen for Firefox

Figure 3-4: Reopen analysis.

- A comparatively high percentage of bugs closed with *Wontfix*, *Invalid*, *Incomplete*, and *Worksforme* labels got reopened for Core. To improve the quality and understanding, more effort should be made to ensure that sufficient information was retrieved from the reporter before closing. We suggest two ways: (1) being interactive and clarifying things with the reporter after a bug has been reported, and (2) improving initial ticket reporting template to capture sufficient details beforehand, thereby reducing the delay. The disagreement in the priority of bugs should be minimized by defining clearer guidelines to decide the priority.

- The chances of getting a *Duplicate* bug reopened in Core (around 5%) were more than double of that in Firefox (around 2%). A thorough understanding of the bug should be gained before marking a bug *Duplicate*. The decision should not be based on superficial attributes [35].

- After *Wontfix*, bugs resolved as *Fixed* were prone to being reopened in Firefox (6%). A proper understanding of the root cause was necessary for fixing a bug [35]. Assumptions should be avoided or minimized. If unclear, the owner should ask the reporter. Also, we recommend verification of resolved tickets to reduce reopening; however, it was not done for the majority of the tickets.

- The verified bugs were rarely reopened (1% or less). However, a small percentage of tickets were verified; therefore, we suggest that the tickets likely to get reopened [35][36] should be recommended to the QA manager for verification.

Of all the reopened bugs, the major contribution was from the tickets resolved with the status *Fixed*, *Duplicate*, *Wontfix*, *Worksforme*, and *Invalid*. Others constituted a lower percentage, as evident from Figure 3-4c for Firefox and Figure 3-4b for Core. As the number of *Fixed* bugs was high (refer Table 3.2), even if a lower percentage of *Fixed* got reopened, it had a higher contribution toward bugs being reopened. Hence, the reasons for reopening *Fixed* bugs should be dealt with high priority.

Bottleneck Analysis: We computed the median time for transitions observed in the derived process model and made the following observations:

1. The median time taken for the transition *New*→*Assigned* was 21.2 h and 3.8 days for Core and Firefox, respectively. This indicated that the assignment of bug took more time for the Firefox project compared with Core. Therefore, we suggest to leverage automatic triaging solution approaches [26][28] to make the assignment more efficient for Firefox.

2. Experimental results revealed that the resolution of cases where the status was *Worksforme*, *Wontfix*, and *Incomplete* was more time-consuming. For instance, a resolution from *New* to *Worksforme* and *Wontfix* was taking an exceptionally long median duration of 17.5 weeks and 24 days for Firefox and 20.6 weeks and 21.8 days for Core, respectively. Therefore, if the bugs likely to be resolved with one of such status can be detected automatically, then the practitioners' time can be utilized for resolving other bugs, which are likely to be fixed.

Conformance Analysis: We considered only closed cases for the conformance analysis. As we captured more activities than defined in the Bugzilla life cycle[11], we created a defined process model on the basis of documented guidelines. We created the adjacency matrix corresponding to the defined process model. The value of Fitness metric, *FM*, using Algorithm 1 was 0.86 and 0.91 for Core and Firefox, respectively. It clearly showed that Firefox had high conformance compared with Core. However, both the projects had fairly high conformance with the defined process model; 818 traces out of 1164 total unique traces were valid for Core. Similarly, 403 traces out of 622 were valid for Firefox, revealing that usually the

[11]https://www.bugzilla.org/docs/2.18/html/lifecycle.html

Table 3.5: Experimental dataset details (Chromium project)

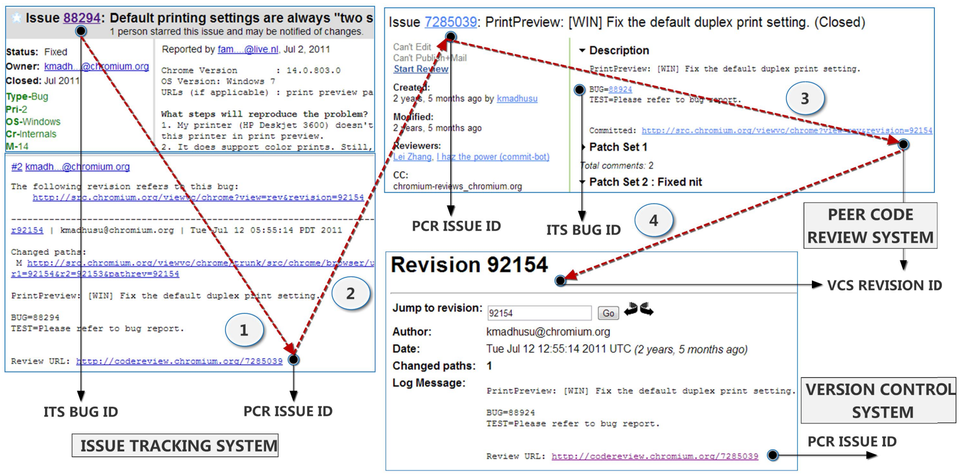
Attribute	**Value**
First issue tracking system (ITS) issue creation date	Jul 1, 2011
Last ITS issue creation date	Jun 30, 2012
Total extracted closed ITS issues	35,035
ITS issues with patches in peer code review (PCR)	10,110
ITS issues with PCR issue reports authorized for access	10,000
Total PCR issues for above ITS issues	19,952
Unique PCR issues for above ITS issues	17,979
Total version control system (VCS) commit (out of all PCR issues)	19,422

invalid traces had low frequency. As $FM < 1$ for both the projects, we analyzed the cause for nonconformance, that is, undefined transitions occurring in the runtime process. Therefore, Algorithm 2 was invoked, which gave the results as 2412 and 739 total inconsistent transitions for Core and Firefox, respectively. The most-frequent inconsistent transition was *Reported* $\rightarrow$ *Assigned* for both the projects with a frequency of 1643 and 305, respectively. Ideally, the *Reported* issue should first be confirmed; however, it was directly assigned in many cases.

3.4 Case Study II: Analyzing Google Chromium Ticket Resolution

We conducted experiments on datasets downloaded from ITS (Google ITS), PCR system (Rietveld), and VCS (Subversion) of Google Chromium browser project, which is a large, long-lived, and complex open-source software project. Issue reports[12], patches[13], and commit details[14] for Google Chromium browser are publicly available; hence, the experimental analysis could be replicated by other researchers and used for benchmarking and comparison. As the data were extracted from multiple information systems that were not explicitly linked, integration of extracted data was performed using textual analysis and regular expression matching.

[12]https://bugs.chromium.org/p/chromium/issues/list
[13]https://codereview.chromium.org/
[14]https://src.chromium.org/viewvc/chrome



Issue 88294: Default printing settings are always "two s
1 person starred this issue and may be notified of changes.

Status: Fixed
Owner: kmadh...@chromium.org
Closed: Jul 2011
Type-Bug
Pri-2
OS-Windows
Cr-Internals
M-14

Reported by fam....@live.nl, Jul 2, 2011

```
Chrome Version        : 14.0.803.0
OS Version: Windows 7
URLs (if applicable) : print preview pa

What steps will reproduce the problem?
1. My printer (HP Deskjet 3600) doesn't
this printer in print preview.
2. It does support color prints. Still,
```

#2 kmadh...@chromium.org

```
The following revision refers to this bug:
    http://src.chromium.org/viewvc/chrome?view=rev&revision=92154

------------------------------------------------------------------
r92154 | kmadhusu@chromium.org | Tue Jul 12 05:55:14 PDT 2011

Changed paths:
 M http://src.chromium.org/viewvc/chrome/trunk/src/chrome/browser/u
r1=92154&r2=92153&pathrev=92154

PrintPreview: [WIN] Fix the default duplex print setting.

BUG=88294
TEST=Please refer to bug report.


Review URL: http://codereview.chromium.org/7285039
```

Issue 7285039: PrintPreview: [WIN] Fix the default duplex print setting. (Closed)

Can't Edit
Can't Publish+Mail
Start Review
Created:
2 years, 5 months ago by kmadhusu
Modified:
2 years, 5 months ago
Reviewers:
Lei Zhang, I haz the power (commit-bot)
CC:
chromium-reviews_chromium.org

Description

```
PrintPreview: [WIN] Fix the default duplex print setting.

BUG=88924
TEST=Please refer to bug report.

Committed: http://src.chromium.org/viewvc/chrome?view=rev&revision=92154
```

Patch Set 1
Total comments: 2
Patch Set 2 : Fixed nit

Revision 92154

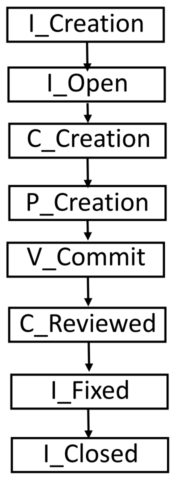
Jump to revision: 92154 Go
Author: kmadhusu@chromium.org
Date: Tue Jul 12 12:55:14 2011 UTC *(2 years, 5 months ago)*
Changed paths: 1
Log Message:

```
PrintPreview: [WIN] Fix the default duplex print setting.

BUG=88924
TEST=Please refer to bug report.

Review URL: http://codereview.chromium.org/7285039
```

(a) Snapshot for mapping three information systems: ITS, PCR, and VCS.



(b) Process map.

Figure 3-5: Illustration to map multiple software repositories, and the corresponding process map.

3.4.1 Data Extraction, Integration, and Transformation

We extracted 1-year data of Google ITS starting from July 1, 2011, to June 30, 2012, using Google issue tracker APIs (Table 3.5). Data for 35,035 issue reports (tickets) were extracted, and all the extracted issues were found to be closed. Some issues reported in Google ITS required source code change (patch) for resolution, which were peer reviewed to avoid defects before they were committed into the source code. Our focus was to study the process followed from the inception (issue reported in ITS) till resolution for the issues requiring code changes. We identified such issues by performing a textual analysis of comments. We started with ITS issue report and mapped to PCR by detecting the presence of code review system URL[15] [16] in comments marked as step 1 in Figure 3-5a. We obtained PCR issue ID from the links posted in comments. It facilitated mapping to PCR issue depicted as label 2 in Figure 3-5a, followed by extraction of PCR patch report details. Around 29% (10,110) of the total ITS issues had at least one link to the PCR system in comments. We extracted the PCR issue ID from all the comments of these ITS issues and the PCR report details for the same. We found that the PCR issue details were accessible for 10,000 ITS issues as mentioned in Table 3.5. We considered the issues with at least one PCR report for further analysis. As shown in

[15]codereview.chromium.org
[16]chromiumcodereview.appspot.com

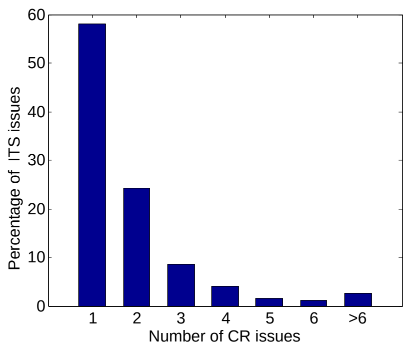


Figure 3-6: Distribution of number of code review (CR) issues for ITS issues.

Figure 3-6, most of ITS issues (around 58%) had only one PCR issue associated with them, and few had more than six also. Overall, we extracted 17, 979 unique PCR patch report details and observed that some patches addressed more than one issue. Therefore, the total mapping count from ITS to PCR was 19, 952, which clearly showed that the relationship between ITS and PCR was *Many-to-Many*.

The patch report contains a brief description and an initial patch submitted by the author while raising an issue in the code review system. The issue is assigned to the reviewer(s), and subsequent patches are submitted for the same issue to address the comments of reviewers. The code change is committed to source code after the approval of reviewers, and corresponding unique revision ID is generated in the VCS (subversion). However, if a patch is not approved by the reviewers, then the PCR issue is closed without commit. Therefore, the mapping between PCR and VCS is *One-to-Zero* if closed without commit, and *One-to-One* if patch set committed successfully as unique VCS commit. As depicted in step 3 of Figure 3-5a, the PCR issue description had "Committed": as part of its text where the link to VCS was posted after commit. We extracted revision ID details from the PCR issue report description and used it for mapping to VCS (labeled 4 in Figure 3-5a). We found that 19, 422 out of 19, 952 PCR issue reports had links to VCS posted in the description. Commit data for all the derived revision IDs were extracted from the VCS.

After extraction and integration of data from all the three repositories, we transformed it to make it suitable for multiperspective process mining. Table 3.6 presents a list of identified important activities, role of people performing the activity, and significance of activity for each information system. There were eight distinct activities for ITS, three for PCR, and

Table 3.6: List of activities, its significance, and the role of the performer for all information systems

Information System	**Activity**	**Description**	**Role**
Issue Tracking System	I_Creation	Issue reported in ITS	Bug Reporter
	I_Open	Open bug status label	Bug Owner, Triager, QA Manager
	I_Fixed	Resolved as Fixed	Bug Owner
	I_Invalid	Illegible, spam, and so on	Bug Owner, Triager, QA Manager
	I_Duplicate	Similar to other issue	Bug Owner, Triager, QA Manager
	I_WontFix	Can't repro, Working as intended, Obsolete	Bug Owner, Triager, QA Manager
	I_Verified	Resolution verified	QA Manager
	I_Closed	ITS progress ends	Bug Owner
Code Review System	C_Creation	Initial patch reported in PCR	Patch Author
(suffix: seq. no.)	P_Creation	Subsequent patch submit with corrections	Patch Author
	C_Reviewed	Code review process ends	Patch Reviewer
Version Control System	V_Commit	Code change committed	Patch Committer

one for VCS. The code review system activities were distinguished by a sequence number showing the order of PCR issue occurrence in ITS. For example, if the third PCR issue is raised to resolve the same ITS issue, sequence number 3 is added as a suffix; thus, activity is *C_Creation3*. Similarly, activities (such as *P_Creation and V_Commit*) pertaining to the given PCR issue are suffixed with the same sequence number. We considered only these activities because they were sufficient for our analysis.

Fields of event log were derived from the extracted data where *caseID* was basically the ITS issue ID, *activity* was one of those listed in Table 3.6, and *timestamp* was the time when an activity was performed. As the activities involved in the life cycle of an issue spanned across three systems, we interlaced transformed event log from the three systems such that the activities pertaining to the same ITS issue were identified by the same *caseID* (here, ITS issue ID). We replaced the statuses *Started, Untriaged, Available, Assigned, Unconfirmed, and Accepted* (part of ITS labels for the open status of a bug[17]) with a common activity

[17]http://www.chromium.org/for-testers/bug-reporting-guidelines

Open, as this minimization would simplify the control flow analysis without missing any useful information for the given study.

An instance is presented in Figure 3-5b where the activities of derived event log were ordered according to increasing timestamp. It depicted control flow for an instance used to explain the integration of three repositories. Here, an issue was reported followed by patch creation, commit, and completion of review, and finally closed after getting fixed. The event log obtained after transformation for 1-year issues could be used to obtain a general runtime process model and for process mining from multiple perspectives. The final transformed event log used for the experiments was made public[18].

3.4.2 Process Discovery

We imported an event log for 9744 cases (with < 7 code review issues) to Disco for process model discovery. The generated process model had 32 nodes, as shown in Figure 3-7, each corresponding to an activity. Out of the 32 nodes, 8 activities were from ITS and the remaining 24 ($6X4$) were generated from 4 unique PCR and VCS activities (with 6 distinct sequence numbers). The process model presented in Figure 3-7 shows the main transitions (most significant and correlated) for simplicity and clarity. The label on the edges represents the absolute frequency of transition, and the value in an activity node is the total number of times that the activity was performed in the complete event log. The shade of a node and thickness of an edge in the process model corresponded to the absolute frequency of activity and transition, respectively. We made the following observations from the discovered process map:

1. The number of cases reduced with an increase in PCR sequence number, as shown in Table 3.7. The majority of the issues were resolved with one patch issue reported to PCR. Out of $9,744$ cases, 678 cases (ITS issues) had more than 3 code review issues associated with them. Only 123 cases had 6 code review issues. Around 97% of total PCR issues ($16,302$) were committed to VCS. Around 42% of the submitted patches, that is, PCR issues with a direct transition from *C_Creation* to *V_Commit* did not

[18]https://github.com/Mining-multiple-repos-data/experimental_dataset

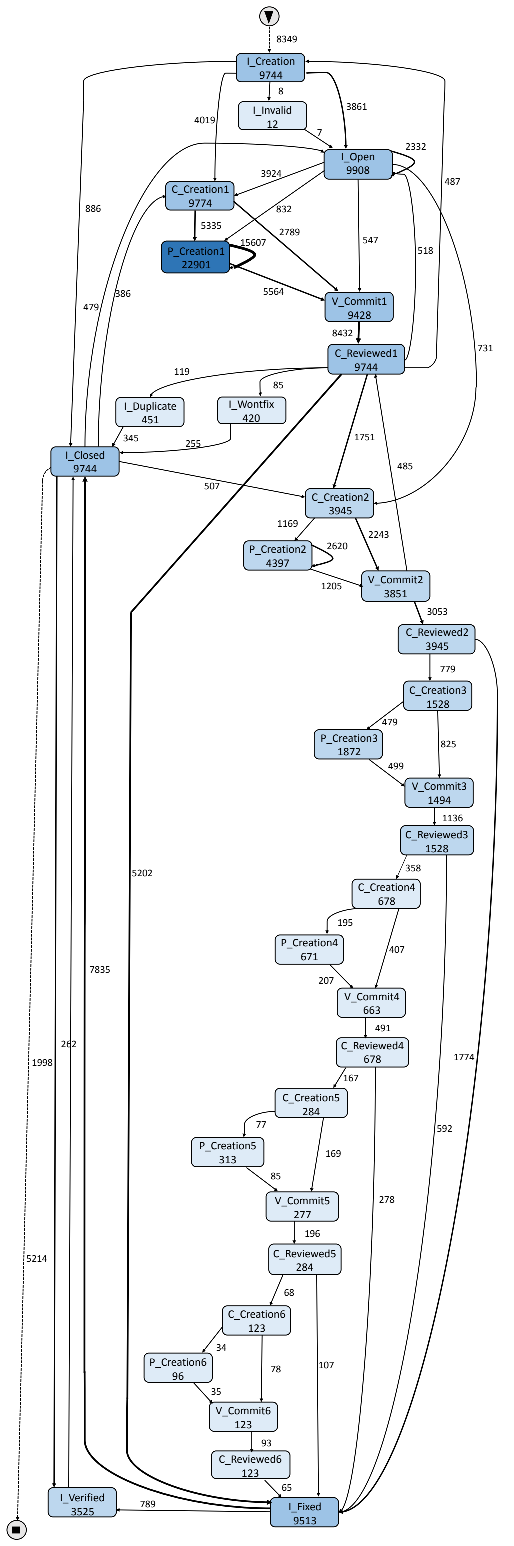


Figure 3-7: Process map for the Chromium bug resolution process spanning three information systems.

Table 3.7: Distribution for the number of ITS issues with given code review issue sequence number and total committed cases

CR sequence no.	**Total cases (% of total issues)**	**Committed cases (% of total cases)**
1	9,744 (100)	9,428 (96.7)
2	3,945 (40.5)	3,851 (97.6)
3	1,528 (15.7)	1,494 (97.7)
4	678 (6.9)	663 (97.7)
5	284 (2.9)	277 (97.5)
6	123 (1.3)	123 (100)
Total	16,302	15,836 (97)

need corrections and were approved for commit without any changes.

2. As observed from Figure 3-7, code review issues were created sequentially one after the other. The patch author reported a patch, which was committed to VCS. Sometimes if an issue was not resolved with the committed patch, another patch was reported for the same issue. As the need for more patches could be realized only after review and commit of current patch, patches for the same ITS issue were reported sequentially.

3. A high percentage (89%) of the cases were resolved as *Fixed*, which was desired, and contributed toward the overall software quality improvement. This indicated that if the source code was changed to resolve an issue, it was highly likely to get fixed, as code change is an indicator of more careful involvement.

4. Some issues were initially marked with wrong statuses as *Duplicate* or *WontFix* or *Invalid*, which were *Fixed* later. It could happen if additional information was available to fully understand an issue or the initial status was assigned incorrectly. It emphasized the need to fully understand an issue to reduce wrong label assignment and extra delay in reopening wrongly closed issues.

5. Final bug resolution was verified for only 35.5% of the cases, highlighting the need to identify reasons for infrequent verification and address them.

   Unique Traces: We had 4453 unique traces for the dataset, which was fairly large. However, the distribution was skewed with some traces being more frequent, as 50% of the cases were covered with only top 6% of unique traces. The most frequent trace covering 4.39% of the cases was *I_Creation* → *C_Creation1* → *V_Commit1* →

C_Reviewed1 → *I_Fixed* → *I_Closed*, which was the minimum sequence of activities to successfully fix an issue.

3.4.3 Analyzing the Discovered Process Model

Anti-patterns: Anti-patterns represent erroneous transitions in the process models. It is important to analyze anti-patterns for a process analyst to detect and eliminate erroneous transitions, thus improving the overall process quality. We identified anti-patterns (undesired transitions) and the cause of their existence for our process model.

It was evident from the discovered process model (Figure 3-7) that *I_Creation* was the first activity for the majority of the cases (8349 out of 9744), indicating that an issue was reported in ITS followed by patch submission to PCR to resolve an issue. However, 1356 cases had *C_Creation1* as the first activity, indicating that a patch was first submitted to PCR followed by an issue creation in ITS. We observed that for the cases where ITS issue was reported after PCR issue, around 55% of the cases had more than one PCR issue associated with them. On the contrary, for cases where ITS issue was reported first, around 40% of them had more than one PCR issue. We believe that ITS issue was reported after PCR issue for such cases after the patch author realized that only one code review issue might not be sufficient for the fix, hence making issue reporting crucial for traceability. It is recommended[19] to first report an issue in ITS before submitting a patch to PCR to maintain a complete record of code changes. However, we identified 1356 cases where this practice was not followed. The remaining 39 cases had an unexpected behavior in which the patch was directly committed to VCS without review followed by issue creation in PCR and ITS. One reason could be some critical project fixed by code committers. However, this situation of emergency happened very rarely. Therefore, efforts should be made to ensure that an issue was first reported in the ITS, followed by further progress spanning across PCR and VCS to resolve an issue.

[19]http://www.chromium.org/developers/contributing-code

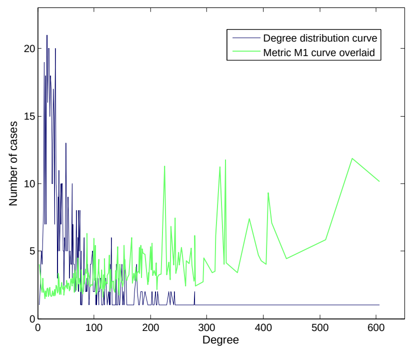


Figure 3-8: Degree distribution graph with metric M1 curve (Section 5.3.3) overlaid for analysis.

3.4.4 Organizational Analysis

Open-source projects such as Google Chromium are driven by volunteer contributions and are important to investigate the interaction pattern between various contributors to improve the efficiency of the project.

Joint Cases: We considered only the actors with participation in more than three cases. The degree distribution curve is plotted in Figure 3-8, where the horizontal axis represents degree and the vertical axis represents the total number of actors having that degree. We observed from the degree distribution curve in Figure 3-8 that most of the actors had degree up to 50, with some having more than 100 also. The number of actors decreased with an increase in degree, which was also verified statistically. The Pearson correlation coefficient was 0.5575, with negative sign indicating the relation between the reduction in the number of actors with an increase in degree. It meant that most of the actors worked with comparatively few people and few had a large social circle. We evaluated and plotted metric $M1$ for each degree d, which is represented with a green line in Figure 3-8. The correlation coefficient between degree and weighted average degree ($M1$) was 0.6661 (fairly high), showing a positive correlation between them. It supported the observation that the strength of interaction was also high for individuals with a large social circle. Effectively they were *active contributors* working on multiple cases together with multiple people (both weighted degree and degree are high).

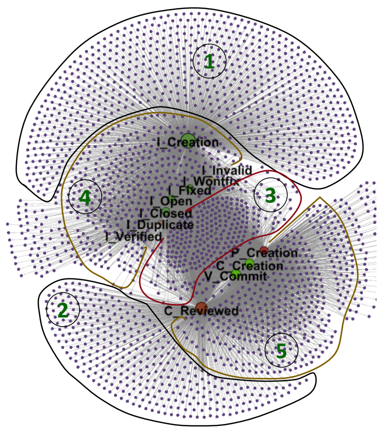
We calculated relative working together metric and presented a list of the top five most

Table 3.8: List of top 5 instances working together with relative working together metric values.

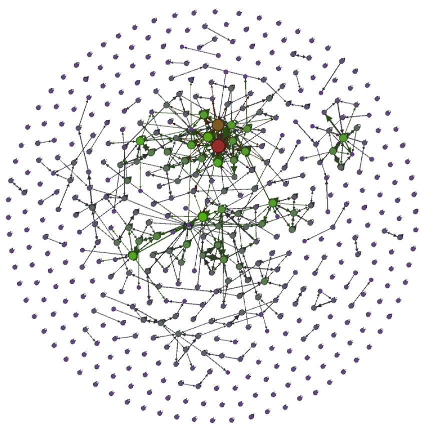
p1	**p2**	**Cases together**	$p1 \bowtie_L p2$	$p2 \bowtie_L p1$
A	B	358	0.094	0.053
C	B	337	0.093	0.050
D	B	314	0.080	0.047
C	A	256	0.070	0.067
D	C	249	0.063	0.068

frequently working together actor pairs (instances) in Table 3.8. The names of the performers were aliased for anonymity. Interestingly, we noticed that active members interacted more with active members. We observed that A was twice closely related to B than B to A because A relatively worked more often with B on all his cases. Similarly, for C-B and D-B, the relative working together metric showed high tendency of C and D to work with B. However, for C-A and D-C, both the actors equally tended to work with each other. Therefore, we can recommend a group of people to work on the same case based on the strength of relation between them.

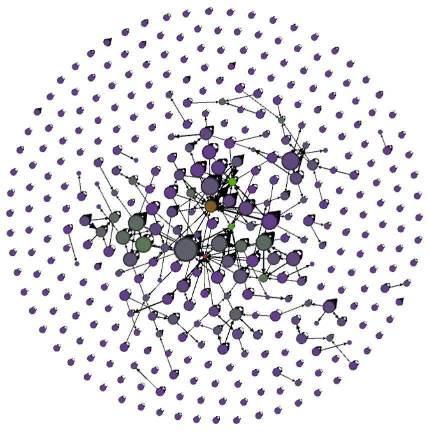
Joint Activities: Figure 3-9a depicts the relation between performers and the activities they performed, where the size of each vertex is proportional to the degree and the color ranges from blue (minimum), green (medium) to red (maximum) corresponding to the weighted degree. If an actor performed an activity, then an edge is drawn between the actor and the activity. There are 2160 unique performers involved in 9744 cases. We notice from Figure 3-9a that 1236 unique reporters reported an issue in ITS. A major section of performers (marked with label 1) reporting issues in ITS was isolated, that is, a large number of actors only reported issues and did not participate in any other resolution activity. Here we had diverse performers reporting issues to ITS for a few number of times. A total of 915 unique performers reviewed patches, and we noticed that a majority of reviewers did not participate in any other activity. Oval 2 in Figure 3-9a highlights a big group of reviewers who were specialists in reviewing the patches. We identified a group of actors reviewing patches more frequently using the available social graph with high weighted edges and gave them commit rights to improve the overall process performance and role assignment. There were comparatively few contributors performing more than one role, that is, *generalists*. We



(a) Social network representing relation between activity and performers (resources).



(b) Sociogram representing handover of work between performers where size of node is proportional to out-degree, color shade corresponds to in-degree ranging from blue (minimum) to red (maximum).



(c) Sociogram representing subcontract of work between performers where size of node is proportional to out-degree, color shade corresponds to in-degree ranging from blue (minimum) to red (maximum).

Figure 3-9: Analysis from Organizational Perspective

observed that the code review activities had comparatively high weighted degree (nodes with bright green and red shades). This implied that more number of distinct people were engaged with ITS whereas less number of dedicated people were involved with patch submission and review activity.

We observed that the group of actors who participated in all the three systems was very small, labeled as 3 in Figure 3-9a. The majority of the actors contributed to only one system by performing single or multiple activities confined to the same system (refer labels 4 and 5 in Figure 3-9a). Therefore, a small group of contributors (labeled as 3) was very crucial as they had knowledge of multiple systems and were *core generalist* contributors.

Handover: We filtered instances with handover frequency less than 15 to remove infrequent instances (not important for our analysis of frequent handover identification). We identified 988 unique handovers, including self-loops having frequency more than the threshold (15). The sociogram obtained for handover had 472 vertices, as shown in Figure 3-9b. Out of 988, most of the instances (429) are self-loops, which means that a good number of performers performed subsequent activities more than 15 times. We noticed many isolated vertices as the performers only had self-loops and no frequent handover with other performers. The size of the vertex corresponds to out-degree, and the color varies with in-degree where blue is for low, green is for medium, and red corresponds to a high range of in-degree. We observe from Figure 3-9b that nodes with high in-degree (red color) had high out-degree (relatively big), with few exceptions where the in-degree was comparatively greater than the out-degree, that is, a small-sized node with green color. Therefore, no performer had the highest authority (out-degree significantly greater than in-degree) to only hand over work to others, which conformed to the expectation from an open-source project where the contributors are volunteers and take up tasks of their choice. The highest weight edges were self-loops, where weight is the frequency of handover, supporting continuous task execution by the same performer. There were instances with handover between different individuals, with frequency as high as 221 and 197. Also, we observed in some instances that the edge was bidirectional, implying the handover of work both ways. However, many instances existed where the work was transferred only in one direction. Hence, we could say that only the destination vertex (second actor) followed the activity performed by the source vertex (first actor). Therefore, we identified handover dependency between different performers without causality validation.

Subcontracting: We identified 6771 subcontract instances with most of the instances having a low frequency. We filtered instances with subcontracting frequency less than 11 to focus on frequent and interesting instances. Overall, we had 577 instances (edges including self-loops) above threshold involving 370 unique performers (vertices) as shown in Figure 3-9c. The size of the vertex was proportional to the out-degree, and the color of the node ranged from blue (lower) to red (highest) for the in-degree. There were more nodes with a relatively large size in blue or green color as shown in Figure 3-9c, indicating that these were the performers with case subcontracted to them by few other performers; however, they subcontracted work to more performers. Very few nodes were with red color (high in-degree) and small size (less out-degree), implying that more individuals subcontracted work to them. As observed in Figure 3-9c, many nodes (on the periphery) were with only self-loops, signifying that the same performer performed subsequent activities multiple times (at least more than 2) with no subcontract to any other performer. In fact, the frequency of self-loops (weight of self-loop edges) was comparatively high, which meant that the same performer often continued working multiple times. At the center of the sociogram in Figure 3-9c is a chain of subcontracts between different performers, showing that they were more related to each other as work was subcontracted between them.

For the Google Chromium browser project, we also compared the process for issue type, *Performance* and *Security* [106]. A common issue resolution process is defined for the Google Chromium project. In this study, we aimed to highlight the differences in process in practice for different bug types. This study used the process cube structure [117], which was discussed in our another study [106].

3.5 Case Study III: Analyzing Ticket Resolution Process for a Large Global IT Company

We performed a case study on IT support ticket data for a large global IT company. Unlike open-source, ticket resolution in industries has expected SLRT, which is agreed upon a priori between the service provider and the client (user) as part of the service-level agreement [114][118]. A ticket is assigned to an *analyst* responsible for servicing the ticket within the associated SLRT. It is crucial to service within the agreed service level because nonfeasance leads to a penalty on the service provider [119]. The analyst can ask for user inputs while resolving the ticket. When this happens, the state of the ticket changes to *Awaiting User Inputs (AUI)*. To prevent spurious penalty on the service provider, the service-level clock pauses while the ticket is in the *AUI* state. Nevertheless, the time spent while remaining in the *AUI* state adds to the URT.

Figure 3-10 illustrates a real example of the ticket life cycle from the information system of a large global IT company. As shown, a user reported a ticket with a short description: “Please install the following software on my machine. This is a CORE software as per EARC.” EARC is a software classification system internal to the organization. The user also provided ticket-specific details, such as hardware asset id, platform, software name, and software version. Based on a priori agreed-upon service levels, the resolution time of 9 h was associated with the ticket. Next, the ticket got assigned to an analyst, a person

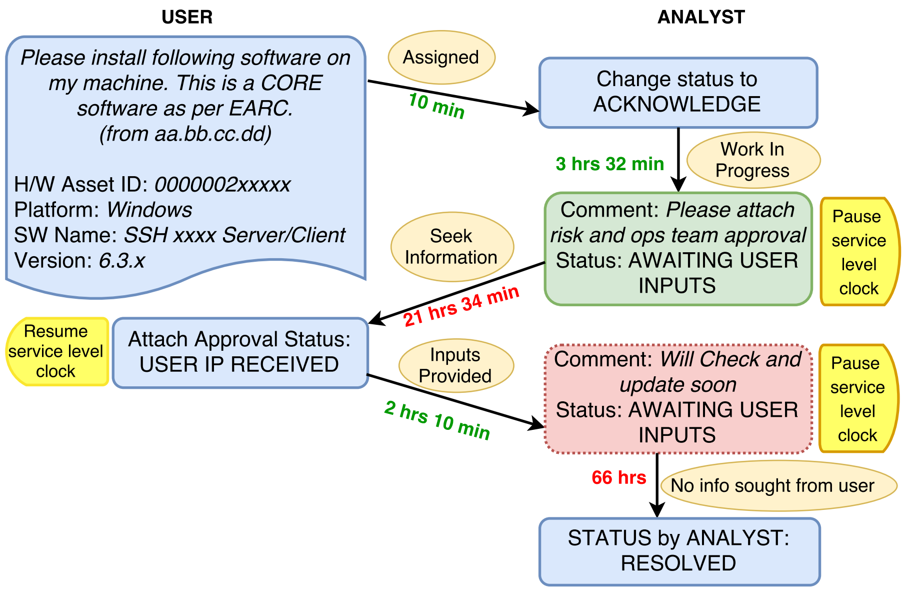

USER
ANALYST
Please install following software on my machine. This is a CORE software as per EARC. (from aa.bb.cc.dd)
H/W Asset ID: 0000002xxxxx
Platform: Windows
SW Name: SSH xxxx Server/Client
Version: 6.3.x
Assigned
10 min
Change status to ACKNOWLEDGE
3 hrs 32 min
Work In Progress
Comment: Please attach risk and ops team approval Status: AWAITING USER INPUTS
Pause service level clock
Seek Information
21 hrs 34 min
Resume service level clock
Attach Approval Status: USER IP RECEIVED
Inputs Provided
2 hrs 10 min
Comment: Will Check and update soon Status: AWAITING USER INPUTS
Pause service level clock
66 hrs
No info sought from user
STATUS by ANALYST: RESOLVED

Figure 3-10: A real example of the ticket life cycle from a large global IT company, illustrating the problem of delay (here delay of 87 h 34 min) in the overall URT due to multiple user input requests by the analyst.

responsible for servicing the ticket. The analyst started working on the ticket and requested the user to "provide risk and ops team approval," changing the ticket status to *AUI*. The user attached the approval after 21 h 34 min (as labeled on the edge in Fig. 3-10). This time was not counted toward SLRT, but it was experienced by the user. Such *real*, information-seeking user input requests could have been avoided if the user had been preempted at the time of ticket submission and required to provide the risk and ops team approval upfront. Sometimes after receiving the user's input, the analyst again changes the status to *AUI* with the comment "Will check and update soon" (highlighted with the dotted outline). Inspecting this comment, we noticed that no information was sought from the user; subsequently, the ticket status changed to *Resolved* without any input from the user. The time for this transition, that is, 66 h, was also experienced by the user, but not included in the measured resolution time. Such *tactical*, non-information-seeking user input requests needed to be detected and handled separately. Summarizing, the resolution time measured by the service-level clock, that is, 5 h 52 min (10 min + 3 h 32 min + 2 h 10 min), was significantly lower than the URT of 93 h 26 min (10 min + 3 h 32 min + 21 h 34 min + 2 h 10 min + 66 h). Consequently, for the presented example, the measured resolution time was within the agreed threshold of 9 h, that is, there was no service-level violation. However, the user did

Table 3.9: Experimental dataset details for the case study in a large global IT company

Attribute	**Value**
Duration	One quarter of 2014
Total extracted *Closed* tickets	593,497
Closed tickets with category *Software*	154,092 (26%)
Total tickets with at least one *AUI state*	88,039 (57%)

not experience the agreed service quality because of the two user input requests.

We discovered a runtime (reality) process from ticket data using the proposed framework to analyze the user input requests and their impact on the URT.

3.5.1 Data Extraction and Transformation

We downloaded data of closed tickets for one quarter and archived them in the organization's ticket system. We ignored open tickets because we wanted to analyze user input requests in the ticket life cycle and the resolution time. Data included the required information about a ticket starting from the time of ticket submission until it was closed. As summarized in Table 3.9, there were $593,497$ closed tickets, out of which we conducted our study on tickets from the software category because it was the most-frequent category constituting 26% of the total tickets. We transformed data for all closed software tickets ($154,092$ tickets) to make them suitable for process mining. In this study, the event log consisted of events having four attributes: *ticket ID*, *activity*, *timestamp*, and *service-level clock state*. The fields were derived from the downloaded ticket data, where *ticket ID* uniquely identified the tickets. We selected *ticket ID* as case ID to visualize the life cycle of a ticket, that is, transition between different activities in the discovered process. *Activity* captures the progress of ticket life cycle, for example, logging of ticket, assignment of ticket to analysts, making a user input request, and marking a ticket as resolved. We included in the event log a subset of the activities that, we believe, captured the progression of tickets, could affect the performance, and was sufficient for the analysis. Also, we validated the list of activities with the manager. The list of activities along with the significance of each activity is included in Appendix B and also made publicly available [120]. All events had an associated *timestamp*, that is,

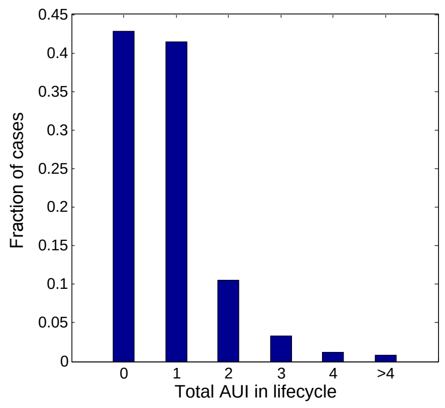


Figure 3-11: User input request distribution: percentage of cases with the given number of AUI state.

the time when the activity was executed. The timestamp was converted into the consistent time zone and captured for each event. The service-level clock state (resume/pause) was inferred for an event from the documentation, which clearly stated the activities for which the service-level clock paused. For example, the service-level clock paused when asking for user inputs, marking a ticket as resolved, and closing a ticket.

3.5.2 Process Discovery

We imported the transformed event log for $157,092$ tickets to Disco for generating the process model and presented the derived process model for the complete process in Appendix B (also made publicly available [120]). The *AUI* state was present in 57% of the tickets, and 27.5% of them had multiple user input requests in their life cycle. With a total number of $125,330$ comments, we observed from the distribution curve depicted in Figure 3-11 that a majority of the cases had one or two user input requests in the life cycle, and some cases had more than four user input requests. In Figure 3-12, we present only the transitions involving *AUI* activity for analyzing its transition patterns. To understand the transition distribution, percentage of transitions from a source state S to a destination state D was measured as:

$$TransitionPercentage, S_D = \frac{S \rightarrow D \ transition \ frequency \times 100}{Absolute \ frequency \ of \ S} \tag{3.2}$$

The *AUI* state acted as D for incoming edges and S for outgoing edges. Incoming edges

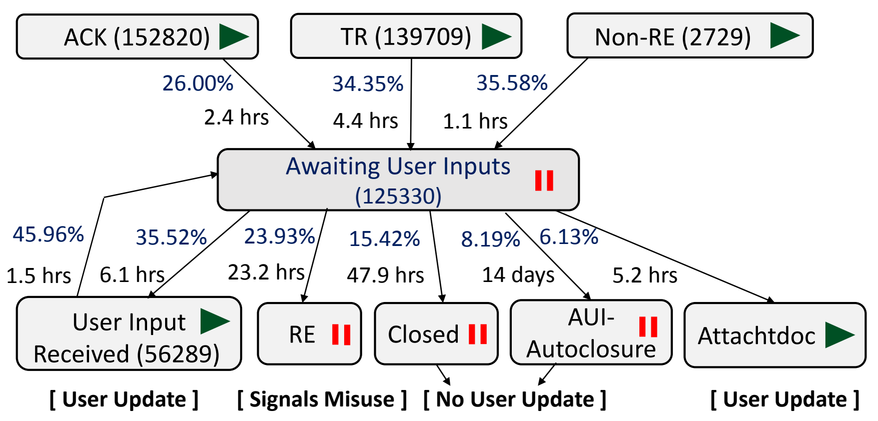


Figure 3-12: *AUI* state transition pattern showing user response classes where S_D and median transition time are edge labels. The state of the service-level clock is indicated using play/pause icons. The activities were as follows: ACK, ticket assigned to an analyst; TR, transfer of ticket to another analyst; non-RE, user marks a ticket as not resolved; Awaiting User Inputs, analyst makes a user input request; User Input Received, user provides input for the user input request; RE, analyst marks a ticket as resolved; Closed, user closes the ticket; AUI-Autoclosure, ticket auto-closed as user did not provide inputs within the defined limit of 14 days; Attachtdoc, user attaches a document.

gave us an intuition on the source state, that is, the activities often followed by user input requests by analysts. Similarly, the outgoing edges allowed us to investigate user response behavior to user input requests by analysts and, thus, the possibility of requests being *real* and *tactical*. The transition percentage for both incoming and outgoing transitions is labeled in Figure 3-12.

3.5.3 Analyzing the Discovered Process Model

Delay due to User Input Requests: From Figure 3-12, we observed that analysts seeked inputs when assigned a new ticket (*ACK*) or when a ticket was transferred (*TR*) to them from other analysts for 26.00% and 34.35% of the instances, respectively. It indicated that as they started working on the ticket, they identified a need for additional information and hence started by asking for inputs. Interestingly, *AUI* was a successor state for *User Input Received* (for 45.96% of the instances) signaling that input from user leads to another user input request. The most common source states for *AUI* were *ACK*, *TR*, and *User Input Received*, together constituting around 90% $[(152,820 \times 0.26 + 139,709 \times 0.3435 + 56,289 \times 0.4596)/125,330]$ of the total incoming transitions (Figure 3-12). *Non-RE* (user marks a ticket as not resolved after analyst says it is resolved) was followed by user input requests in 35.58% of the instances but constituted merely 0.7% $(2,729 \times 0.3538/125,330)$ of total user input requests. We explored outgoing edges and classified destination states broadly into the following two classes:

- *User update:* We observed that users provide inputs, *User Input Received* (comment from the user) or *AttachtDoc* (document attached by the user), for around 42.00% of the requests. We conjectured that user input requests with these destination states were mostly information seeking (real) and thus updated by the user. For around 15.42% of the instances, the user did not provide information and explicitly closed a ticket instead, that is, the destination state was *Closed*.

- *No update:* As shown in Figure 3-12, 23.93% of the total *AUI* stated transit to *RE* (resolved) without any update from the user. We conjectured that such user input requests were more likely to be the *tactical* ones as the analysts managed to resolve the ticket without receiving user inputs. For 8.00% of the cases, a ticket was auto-closed. That is, the destination state was *AUI-Autoclosure* because no user action was performed in response to the user input request within the defined time limit of 14 days (as enforced in the given information system).

We measured the URT and compared it with the SLRT to capture the gap between them. To evaluate URT for ticket i, we used the total time elapsed between ticket assignment and final resolution of ticket, without excluding user input waiting time and nonbusiness hours, as follows:

$$URT_i = ts_i(Resolved) - ts_i(Assigned) \tag{3.3}$$

where ts is timestamp at which ticket i is marked *Resolved* and *Assigned*.

We analyzed only the cases that were resolved and never reopened. Overall, 105,539 such cases existed for which we evaluated the URT. The *Software category* had tickets with three service resolution time thresholds as per the organization's service-level agreement: 9 h, 18 h, and 36 h. We grouped tickets into three categories on the basis of same SLRT. As shown in Table 3.10, *a high percentage of cases had URT more than the agreed-upon SLRT.*

Table 3.10: Gap between SLRT and URT (SLRT, Service-level resolution time; URT, user-experienced resolution time.)

Class	**#Cases**	**SLRT**	**Median URT**	**Cases with URT > SLRT**
1	16,110	9 h	21 h	62.46% cases
2	83,939	18 h	26.34 h	72.49% cases
3	5,490	36 h	77.84 h	76.14% cases

However, the service-level violation was recorded for very few cases[20] because the waiting time was not counted toward the measured resolution time. The median resolution time experienced by a user was 21 h, 26.34 h, and 77.84 h for cases with SLRT of 9 h, 18 h, and 36 h (Table 3.10), respectively. This highlighted that the URT was much higher than the measured resolution time due to user input requests, indicating a bottleneck in the ticket resolution life cycle.

To summarize, process mining analysis showed that 57% of the tickets had user input requests in the life cycle. Users provided input to around 42% of the total user input requests, which we considered as potentially *real* requests. For around 23% of the cases, the ticket was resolved without any user inputs, thus corresponding to potentially *tactical* requests. User input requests caused a significant gap between the measured resolution time and the URT. The findings clearly highlighted the need to reduce real and tactical user input requests.

3.6 Threats to Validity

Case studies were performed on specific projects, although large and long lived. While the proposed process mining analyses provided actionable insights on the given projects, they might not generalize to projects with different characteristics. Since case studies were performed on open-source projects, in some cases we did not have access to sufficient data to support our inferences.

For Case Study I, we analyzed data extracted from the Bugzilla issue tracking system of the popular open-source Firefox browser and Core project. While we identified the bottlenecks, that is, more time-consuming transitions, we did not have information about the nonworking hours and time-zone differences, which could influence the inferences. We iden-

[20]We cannot reveal exact numbers because of confidentiality.

tified some deviations as part of the conformance analysis; however, we could not validate whether those were permissible deviations or indeed the violation of the defined process.

For Case Study II, we used heuristics to integrate multiple information systems, that is, Issue Tracking System (ITS), Peer Code Review (PCR) system, and Version Control System (VCS), for Google Chromium ticket resolution. As the integration approach relied on the presence of URLs with specific format in the ITS and specific text string in PCR, the integration was prone to error, although we performed manual analysis to mitigate such cases.

In Case Study III, we analyzed the transition pattern for awaiting user inputs in a large global IT company and conjectured that there were both real and tactical user input requests. Tactical user input requests were observed in Volvo IT organization ticket data [121] and the data for the large global IT company investigated as part of this case study. However, tactical user input requests can be very rare or not present in other IT support ticket data for other organizations with different characteristics (such as small size, less workload, and lenient service-level resolution time limit). Also, it is possible that the user responded to a non-information-seeking user input request (maybe by expressed displeasure), but it was recorded as 'User Input Received' and thus falsely inferred as real user input request. Similarly, it is possible that the user did not respond to a real user input request and thus might appear tactical from the transition pattern. However, we could detect such tactical and real user input requests by analyzing the comments as discussed in the next chapter.

3.7 Summary

We explored applications of various process mining tools and techniques to analyze the event log data generated by various information systems (such as issue tracking system (ITS), peer code review system (PCR), version control system (VCS), and IT infrastructure support ticketing system) used during software ticket resolution. We conducted a series of case studies on data extracted from Bugzilla ITS of the popular open-source Firefox browser and Core project; on data of open-source Google Chromium project for simultaneously mining ITS, Reitveld PCR system, and subversion VCS; and on data of ticketing system for a large global

IT company. Using the proposed framework, we discovered the process model and analyzed it from multiple perspectives, such as bottleneck identification, reopen analysis, loops and back-forth, anti-patterns, conformance analysis, and delay due to user input requests. This helped us understand the actual process at a granular level and identify inefficiencies.

Some of the findings of the case studies were as follows:

3.7.1 For Case Study I: Bugzilla Issue Tracking for Firefox and Core

The runtime process model was discovered by process mining event logs of 12, 234 Firefox and 24, 253 Core issue reports. The analysis of the discovered process model revealed inefficiencies, such as reopen, back-forth, bottlenecks, and deviations, in the designed process model. The following observations were made:

- A significant number of bug reports (1296 for Core and 222 for Firefox) had developer reassignment, causing significant delays (refer Table 3.3) in ticket resolution [26][27][28]. In fact, maximum loops were for developer reassignment, that is, repeated reassignment. Therefore, the assignment of tickets needs improvement [26][28].

- Resolution of tickets as *wontfix* and *worksforme* was time-consuming. Also, several bug reports with these resolutions were reopened, signaling the need for improvement in identifying such tickets.

- Bugs were reopened from different resolution states for various reasons. We observed that not all the reopened tickets were resolved as *Fixed*. Some of them were again resolved with the initial status (back-forth), indicating disagreement. Verified bugs were rarely reopened; however, a lower percentage of tickets were verified. Therefore, it will be interesting to recommend ticket verification based on the chances of ticket getting reopened. Also, the existing solutions for reopen prediction [56][36] can be deployed if the reopening of bugs is identified as an inefficiency.

- Identification of duplicate tickets needs improvement because many duplicates were reported that went undetected at the time of reporting. They can be eliminated by adopting some of the existing automated techniques [24][25].

- While the value of the fitness metric (0.86 and 0.91 for Core and Firefox, respectively) was quite high for both projects, there is a need to understand the impact of the identified deviations because not all the deviations degrade the performance.

3.7.2 For Case Study II: Google Chromium Ticket Resolution

Process map (reality) discovered from the runtime event log for 9744 Google Chromium ITS issues, with resolution activities spanning across three information systems Chromium ITS, Rietveld PCR, and Subversion VCS, revealed the control flow for the complete life cycle of issues. A process map was derived with 32 states (activities) and core transitions from which some of the interesting findings were:

- A fairly large number of cases, that is, 1395 (around 15%), had an issue in ITS instantiated after patch submission in PCR or commit in VCS. Ideally, for traceability reasons, a ticket's life cycle should start from the issue reporting in ITS, followed by patch submission in PCR and commit in VCS. The impact of this anti-pattern needs further investigation. If it degrades the quality, then there is a need for a better process control mechanism to mitigate such undesirable deviations.

- Using organizational analysis, we analyzed the interaction pattern between various actors. It helped identify generalists and specialists and more active contributors.

3.7.3 For Case Study III: Ticket Resolution Process for a Large Global IT Company

The process model was discovered for $593,497$ tickets of a large, global, IT company and analyzed for the delay caused by user input requests.

- The process mining analysis showed that 57% of the tickets had user input requests in the life cycle. Users provided input to around 42% of the total user input requests, which we considered as potentially real requests. For around 23% of the cases, the ticket was resolved without any user inputs, and thus corresponded to potentially tactical requests.

- User input requests caused a significant gap between the measured resolution time and the user-experienced resolution time. The findings clearly highlighted the need to reduce real and tactical user input requests.

Open source projects were analyzed in Case Study I and Case Study II whereas ticket resolution process for a large global IT company was analyzed in Case Study III. Commercial projects are driven by Service Level Agreement (SLA). So, we focused the analysis in Case Study III on time perspective that is, resolution time which is an important metric in SLA. Specifically, delay caused due to user input requests in the ticket resolution process is analyzed. Open-source projects do not have the notion of SLA so we could not perform similar analysis for the open source projects. In Case Study I and Case Study II, we focused on bottlenecks, loops, back-forth, reopen, anti-patterns and conformance analysis. Therefore, while the same proposed approach was used for process model discovery and analysis for both open source and commercial projects, the perspective of analysis was different thus could not be compared.

Case Study I was performed first in which data from single software repository that is, Bugzilla issue tracking system was analysed. Hence, when selecting the second case study we picked one in which we needed multiple repositories for the analysis. The third case study was on commercial project where the analysis was driven largely by Service Level Agreement (SLA) as improving SLAs was the most important business goal for the organization.

Some of the observations made in the presented case studies are known, explored problems in the literature, which can be mitigated by deploying the existing solutions. Some inefficiencies identified have not been explored earlier for the ticket resolution process in these projects. These can lead to improvements in the ticket resolution process. In the next chapter, we describe one such improvement, namely, to reduce user input requests causing a delay in the ticket resolution life cycle.

Chapter 4

Reducing User Input Requests in Ticket Resolution Process

User inputs are asked during the ticket resolution process due to various reasons such as incomplete or unclear initial ticket [114][122][123][124]. Further, non-information-seeking user input requests can also be performed merely for pausing the service-level clock, thus achieving a service-level target of resolution time [115]. Interactions for user input requests cause delays and degrade the ticket resolution process [115][122][123][124].

Also, from the case study III (refer to Chapter 3) on the tickets of a large global IT company, we found that around 57% of the tickets had user input requests in the life cycle, causing user-experienced resolution time to be almost twice as long as the measured service resolution time. We observed that the user input requests were broadly of two types: *real*, seeking information from the user to process the ticket; and *tactical*, when no information is asked but the user input request is raised merely to pause the service-level clock. Therefore, to reduce the overall user input requests, both real and tactical types should be minimized. The work presented in this chapter was motivated by *the need to minimize the overall user input requests in tickets' life cycle, and thus to reduce user-experienced resolution time and enhance user experience.*

To achieve this, we proposed a machine learning-based system that preempts a user at the time of ticket submission to provide additional information that the analyst is likely to ask, thus reducing real user input requests. We also proposed a rule-based detection system to identify tactical user input requests. We conducted a case study on the IT Infrastructure Support (ITIS) data of a large global IT company (same as the data for case study III in Chapter 3) demonstrating the usefulness of the proposed solution to reduce user input requests. Together, the proposed preemptive and detection systems could efficiently bring down the number of user input requests and improve the user-experienced resolution time.

4.1 Information Needs for Ticket Resolution

Various studies have been conducted to identify information needs and preempt users for information given on the incomplete ticket. Bettenburg *et al.* [124] conducted a survey on developers and users from Apache, Eclipse, and Mozilla to identify the information that made good bug reports. Further, they designed a tool Cuezilla that provided feedback to the user at the time of ticket reporting for enhancing bug quality. Yusop *et al.* [122] conducted a survey focused on reporting usability defects, and the analysis of 147 responses revealed a substantial gap between what developers provided and what software developers needed when fixing usability defects. These studies captured the information deemed important in the opinion of the users and developers. As opposed to this line of work we did not rely only on the intuition and domain knowledge of users and developers but performed data-driven analysis. Therefore, we focused on what information developers needed as opposed to what information developers believed they needed.

Chaparro *et al.* [125] analyzed 2912 bug reports from 9 software systems and observed that while most of the reports (i.e., 93.5%) described observed behavior, only 35.2% and 51.4% of them explicitly described Expected Behavior (EB) and Steps to Reproduce (S2R). Therefore, they developed an automated approach, DeMIBuD, to detect the absence of EB and S2R in bug descriptions. While they alerted reporters about missing EB and S2R for efficient ticket resolution, we preempted for specific information needs to service the ticket.

Ko *et al.* [126] looked at thousands of bug report titles for several open-source projects and identified fields that could be incorporated into new bug report forms. They analyzed only the titles of the bug reports, not the comments to determine the information asked during bug resolution. Breu *et al.* [123] identified eight categories of information needs by analyzing the interaction between developers and users on a sample of 600 bug reports from the Mozilla and Eclipse projects. They found that a significant proportion of these interactions were related to missing or inaccurate information. They observed some rhetorical questions (no information asked), but did not consider them for detailed analysis. While the interaction between developers and users was analyzed, it was for a small sample of bug reports. We focused on IT support tickets and analyzed the analyst comments for a large number of tickets.

When reporting a ticket, a template corresponding to the ticket category and subcategory is chosen. For example, if a user selects the category as *software* and subcategory as *install*, the ticketing system automatically asks the user to provide details as per the corresponding template. Although ticket-reporting templates try to capture the required details, they have limitations, motivating the need for the preemptive model:

- Users do not provide all the details asked in the initial template because of limited understanding or time [114]. A balance needs to be maintained between two contradictory demands: ask as much information as possible to help the analyst to service the ticket in the best possible way and as little information as possible not to put too much burden on the user submitting the ticket. It is not possible to make all the fields mandatory because this would make it difficult for the users to submit their requests. The preemptive model can help in such situations by preempting only if the missed information is indeed crucial for processing the ticket. For example, if a user submits a ticket in the software installation category and writes a description as "install latest version of MS office" but leaves the version field blank, the system should allow the user to submit this ticket without preemption because an analyst can service the ticket by installing the latest version.

- If a user selects the wrong category for the ticket, the corresponding template will not capture the information required to resolve the ticket. The learned model can preempt the required information because the ticket description provided by the user is used as one of the features for preemption and the model does not rely on the chosen ticket category.

- Users tend to provide incorrect or unclear information [114] to pacify the system, which the learned model can preempt. For example, if a user mentions the version as some random value, such as *xx* or *2.3* for MS Office, then the learned model still can preempt and indicate that the version has been asked for similar tickets.

- Some tickets have specific information needs, which are not captured in the corresponding template, for example, if a user requests for installing the software that has a requirement such as approval for purchasing software license in case of specific proprietary software. Such information needs are not always intuitive for users and hence can be preempted by the learned model.

Effectively, the preemptive model should facilitate dynamic information collection for faster processing of the reported ticket. Being a preventive measure, it helps in improving the efficiency by eliminating later interaction delays and enhancing user satisfaction [114].

4.2 Preemptive Model: For Real User Input Requests

The preemptive model is an automated learning-based system deployed at the time of submitting a new ticket. It preempts the user to provide the information required for servicing the ticket. As depicted in Figure 4-1, the major steps involved in designing the preemptive model are preprocessing, feature extraction, training classification model, and preemption for a new ticket at the time of submission. To preempt the information needed for a given ticket, we learn the model to predict the following:

- **P1:** To process a given ticket, will there be user input request?

- **P2:** If there will be a user input request according to **P1**, what is the specific information that is likely to be asked by the analyst?

To train a supervised model for **P1**, the ticket is labeled as 1 if *Awaiting User Inputs* state is present in the ticket's life cycle, otherwise 0. This information is derived from the event log extracted for each ticket.

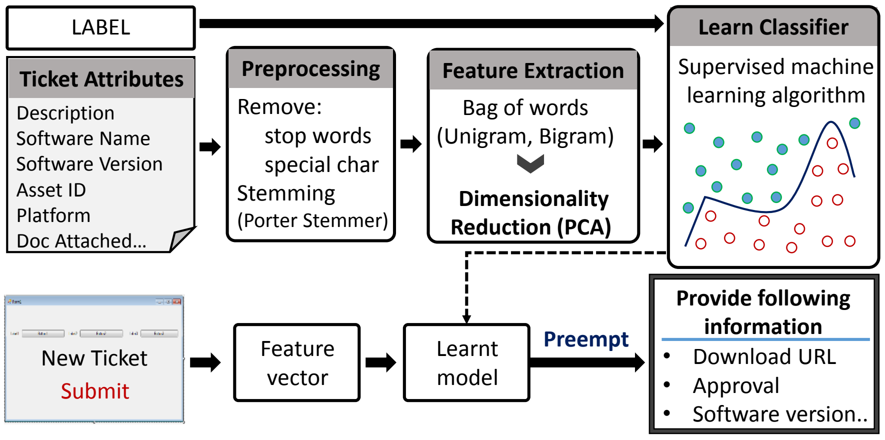


Figure 4-1: Preemptive model to preempt users with additional information needs at the time of ticket submission with broadly two stages: training and preemption.

4.2.1 Ground Truth

To train and evaluate the model for **P2**, we established the ground truth for information needs. The ground truth [GT(x)] for a ticket with respect to specific information is defined as follows:

$$GroundTruth, GT(x) = \begin{cases} 1, & \text{if x information asked in ticket's life cycle.} \\ 0, & \text{if x information not asked in ticket's life cycle.} \end{cases} \tag{4.1}$$

First, the information asked by the analysts (such as software name, software version, machine IP address, operating system, and manager approval) in the user input request comments was identified on the basis of the manager's domain knowledge and manual inspection of the user input request comments. Manual inspection was performed by the author and one of her colleague for a disjoint set of comments (a random sample of 1000 comments each) to identify the information needs. Information needs identified by the author and her colleague were compared to create the consolidated list. They used different terms to represent the same information needs, which were made consistent. Both of them identified the same 23 information needs with one exception, that is, asking the user the duration for which the requested software would be used, which was identified by only one of them because it was a rarely asked information. All the information needs mentioned by the manager turned out to be a subset of the consolidated list.

Establish ground truth by annotating comments using keywords-based approach: The ground truth for every information need was established using a *keyword-based* approach [127]. A list of keywords corresponding to each information need was prepared iteratively. The initial set of keywords was created using the domain knowledge of the managers. For example, for information need software version, "software version, sw version, and software number" are some of the commonly used terms, which were included in the keywords list. Porter stemming and case folding of comments and keywords were performed to improve the matching of keywords with the comments. If the comment contained the keywords, it was annotated with the corresponding information need. Thereafter, we (author and her colleague) manually investigated the disjoint set of randomly selected unannotated comments (around 500 each) to identify the comments missed out using a given set of keywords. Keywords were added to the initial set to capture the missed-out comments. Also, a disjoint set of annotated comments (50% of the total annotated by author and her colleague respectively because it was typically a small set) was manually analyzed by the author and her colleague to eliminate wrongly annotated comments. The keywords were updated to distill the wrongly annotated comments. The comments were now annotated with the updated set of keywords. This process was repeated two to three times until very few/no updates were made in the set of keywords. Similarly, keywords were created for every information need. Keywords for information can vary across the organization based on their specific terminologies.

Evaluate ground truth established using keywords-based annotation: To evaluate the keywords-based annotation, we decided to establish the ground truth for a set of comments manually and compared it with the keywords-based annotation. For the ground truth data, we requested second-year students of B.Tech in Computer Science of the university for annotation. Each participant was promised a cash gift (of 1000 INR) as a token of gratitude. We received interest from nine students and shared the details with each of them. Three of them dropped out, and the remaining six were given a short in-person demonstration of the tool (screenshot made publicly available on GitHub [120]) that we designed for convenient annotation. The tool had a simple UI that showed comment for annotation and a list of keyphrases, including manually identified 23 information needs as an option. Each participant was given a set of 4000 different comments and 15 days' time as agreed by students, thus ensuring that they performed annotation without any pressure. Finally, we received annotated files from 5 participants, that is, 20000 annotated comments. The author randomly verified 50 annotated comments for each student that is, a total of 250 annotated comments (12.5% of total annotated). For every information need, keywords-based annotation was compared with ground truth annotated by students. We observed that the keyword-based annotation was consistent with the manually annotated ground truth for $90\% - 95\%$ comments for various information needs. We compared the top five information needs only: software version, IP address, approval, operating system, and asking location/cubicle ID of the user. Manual inspection of the inconsistently annotated comments showed that, in some cases, the keywords-based annotation was incorrect (the keyphrases did not match mostly because of rare typos by analysts) and, in some cases, the students' annotations were incorrect (attributed to human error). Therefore, we ignored this inconsistency. This validated that the keywords-based approach correctly annotated the comments in most cases.

To annotate the ticket, every user input request comment for a ticket was checked for its label. The ticket was labeled as 1 for the given information need if any of its comments were annotated with the same information; otherwise, it was labeled as 0.

4.2.2 Ticket Preprocessing

A ticket consists of a short description and fields to capture category-specific information about the ticket, such as software name, version, platform, and attachment such as manager approval. Some of the ticket attributes can be free-form text data and hence require preprocessing as shown in Figure 4-1. Common textual preprocessing practices, such as case folding, stemming, stop words, and punctuation removal, were performed [128]. We performed stemming using Porter stemmer [129]. Removing classical stop words, such as "a" and "the", was not sufficient because some stop words were specific to the context. For context-specific stop word removal, we combined all the tickets into one text file and extracted the term frequencies (tf) of the unique tokens from the text. We manually created a dictionary of stop words for a given context, which contained words such as "Dear", "Please", and "Regards". A complete list of stop words is made publicly available at the Github [120].

4.2.3 Feature Extraction

As shown in Figure 4-1, a bag-of-words feature model is used to represent each unstructured feature extracted from the ticket. A bag-of-words representation is known to extract good patterns from unstructured text data [130]. The bag-of-words model can be learned over a vector of unigrams or bigrams or both extracted from the text data. For instance, first we tokenized the description of the ticket shown in Figure 3-10 and then stemmed the tokens, that is, "following" was stemmed to "follow". After stemming, we removed the stop words "on", "my", "this", "is", "a", "as", and "per". The resultant bag-of-words consisted of unigrams "install", "follow", "software", "machine", "core", and "EARC". For most bag-of-words representations, gram (unigram or bigram) features found in the training corpus had weights such as binary or term frequency or term frequency-inverse document frequency [128][131]. We used the bag-of-words feature with the term frequency weights. Concatenation of bag-of-words features (both the unigrams and bigrams) with features corresponding to other fields, such as platform name, was used as the feature description for the entire ticket. Given the high-dimensional and sparse nature of the final representation, learning a classifier might be affected by the curse of dimensionality [132]. Therefore, we applied principal component analysis (PCA), a feature vector dimension reduction technique with minimum information loss, such that 95% of the eigen energy was conserved [133]. For preemption, a feature vector extracted from the ticket submitted by a user was mapped to the reduced dimensional space learned from the training data.

4.2.4 Training and Preemption

The preemptive system was learned over features extracted using tickets' data at the time of submission. To address **P1**, a binary classifier was trained over a set of labeled tickets to classify if the user input request was made for a given ticket or not. If the classifier predicted the class as 1, that is, a user input request was made, the next question (that is **P2**) was to identify the specific information likely to be asked, such as the version number of software or approval for processing. To identify the need for each of the possible information, an independent binary classifier was constructed. As shown in Figure 4-1, when a new ticket was submitted, the cumulative results of the learned binary classifiers suggested the subset of information that could be further required to easily process the ticket. By dividing this complex task into simple binary classifiers, more flexibility was added to the preemptive model. If a new information need was identified in the future, a new binary classifier could be trained for the corresponding information without the need to retrain any of the existing classifiers. In our study, we used a supervised learning model, support vector machines (SVM) [134]. SVM is a binary linear classifier that attempts to find the maximum margin hyperplane, such that the distance of data points from either of the classes is maximized. Furthermore, it performs classification very effectively using a technique called a kernel trick, by implicitly mapping input data into a higher-dimensional feature space, where linear classification is possible. SVM is a popular choice and is often used in the literature [27][135][136][137][138][139]. To compare the efficiency of SVM for the proposed preemptive system, we evaluated other commonly used classifiers such as naive Bayes [140], logistic regression [141], and random decision forest [142]. The performance of a classifier strongly depends on the value of its input parameters, whose optimal choice heavily depends on the data being used [143]. We chose the parameters for the classifiers using grid search [144] and heuristics [145].

4.2.5 Evaluation

A 50/50 *train/test split protocol* was followed to train and evaluate the classifier. In comparison with a more lenient protocol, such as 80/20 split, the proposed split was less risk-prone in terms of generalizability [146]. To address the challenge of imbalanced class labels in train data, we performed random undersampling of a majority class as recommended by Japkowicz [147]. For creating the training set in a binary classification setting, 50% of the data points were randomly taken from the minority class and an equal number of data points were randomly sampled from the majority class. Thus, it was ensured that the training data had an equal number of data points from both the classes. The remaining 50% data of the minority class and all the remaining points of the other class were included in the test split for evaluation (testing). To make a realistic estimation of the classifier performance and to avoid any training bias, we performed random subsampling cross-validation (also called

Monte Carlo cross-validation [148]) five times where new training and test partitions were generated (at random) each time using the aforementioned protocol. The evaluation metrics were averaged over the five rounds, and the standard deviation was computed.

Accuracy places more weight on the majority class than on the minority class, and thus is prone to bias in case of imbalanced datasets [149]. Therefore, additional metrics, such as precision and recall, are used. Classes with labels 1 and 0 correspond to positive and negative classes, respectively. TP and TN denote the number of positive and negative examples that are classified correctly, while FN and FP denote the number of misclassified positive and negative examples, respectively. We evaluated the performance of the learned classification model on the test set using the following evaluation metrics:

Accuracy = (TP + TN)/(TP + FN + FP + TN)

Precision of positive detection = TP/(TP + FP)

Recall of positive detection = TP/(TP + FN)

4.3 Detection Model: For Tactical User Input Requests

Identifying tactical input requests is important because such requests degrade user experience. Some of the user responses recorded in the ITIS information system of a large global IT company are as follows:

- "I have already provided all the necessary inputs. Please take actions."
- "Kindly let me know what inputs are required from my end. As mentioned in my earlier comments, I have already provided the necessary information but I still see the status as Awaiting User Inputs. It's already been about a week since I submitted this request and the issue has not been resolved as yet. Request you to kindly do the needful."

Although users and managers recognize tactical user input requests, following are the challenges in handling this practice, thereby highlighting the need for an automated detection system:

- By the time users recognize tactical user input requests and give feedback, the user experience has already been degraded. With the automatic detection system, it is possible to identify such requests in a proactive way and prevent users from receiving such requests, thus enhancing user experience.

- Merely looking at the complaints gives a biased impression because not every user raises a complaint about such tactical user input requests. Raising complaints is an additional effort for the users, which every user may not like to do. Moreover, many users (specifically new ones) are not familiar with the process and the fact that service-level clock pauses when a ticket is in Awaiting User Inputs state. Thus, they do not realize the need to complain about such experiences.

- A manager needs to look at the comments manually to decide if an input request seeks any information or not, that is, if it is tactical. This control is human intensive and not always possible given the high number of input requests made by a team of analysts every day.

- Automatic detection allows to derive actionable insights, thus helping managers make informed decisions. For example, if tactical requests are made by specific analysts, then they are tackled at the individual level. However, if they are practiced by the majority of analysts, then organization-level decisions are taken, such as redefine service-level resolution time limit or employ more analysts.

It is difficult to handle the tactical user input requests because of perpetual competition. The proposed detection model was an initial attempt to identify tactical user input requests. The detection model identified tactical user input requests in real time by analyzing analysts' comments when changing the status to *Awaiting User Inputs*. For this, we suggested classifying the user input requests using a keyword-based rule classifier. As part of this classifier, we created a set of rules using which the comments were annotated. Rules are a set of regular expressions derived to represent the keywords for tactical user input requests in a concise way. Rules are created for identifying user input requests where no direct information is asked. The domain knowledge of managers can be used to create an initial set of rules, which can be updated iteratively by manually inspecting the tactical comments from the data corpus. Once the ruleset is ready, any user input request by the analyst is checked against it for the classification. If the user input request is identified as tactical, it is logged in the ITIS information system and the manager may be notified to take suitable actions.

As opposed to the preemptive component, we did not use machine learning because of the differences in the context. As part of the preemptive model, information need is preempted for resolving a ticket at the time of ticket submission, whereas in the case of a detection model, a comment by an analyst during the ticket's life cycle is classified (not preempted) as tactical or not. For **P2**, the ground truth is created for a ticket by analyzing the analyst comments, such as if a version is asked in some comment, then the ground truth for a ticket is labeled as 1 with respect to a class *version*. Therefore, the preemptive model takes a ticket as an input and preempts the information need using the learned models. Unlike the preemptive model, detection of tactical user input requests requires learning a classification model from labeled analyst comments. Because we manually created the ground truth label for tactical comments using keywords-based approach (as done for **P2**), learning a classification model does not add value. Comments can be classified as tactical using the keywords-based approach as and when they are written by the analyst. Therefore, for a given scenario, a set of rules to concisely represent the manually identified keywords for tactical user input requests is sufficient. Learning a classification model for tactical user input requests would have been an option if we had human-annotated tactical user input requests available.

The detection model identifies whether an input request by the analyst belongs to one of the classes below. The classes are created on the basis of data analysis and discussion with the manager for a given case study. A separate set of rules is derived for each category.

- *Temporize:* The analyst indicates that the ticket will be handled soon and mentions things such as "Work in progress" and "Will check and update".

- *Invalid:* No valid character is present in the string; comments such as empty strings or strings consisting of a few special characters only.

- *Contact Me:* The analyst asks the user to contact him/her over phone or chat or meet in-person instead of asking for specific information.

- *Will Transfer:* The analyst informs the user that the ticket will be transferred to another analyst and marks the state as *Awaiting User Inputs*. The ticket is transferred to another analyst later instead of transferring directly.

- *Done So Check:* The analyst asks the user to check if the resolution is satisfactory. Ideally, the analyst should mark the ticket as *Resolved* when done with a resolution from their side and let the user reopen, if unsatisfied.

This classification helps managers to decide on the appropriate course of action. For example, if the class is *Invalid*, the user input request can be blocked and if the class is *Contact Me*, then it can be logged for clarification with the involved user and analyst to verify whether there was a need for contact.

For *evaluation*, we requested managers to randomly select comments from the classified ones and indicate if they have been wrongly labeled. This ensured high precision, but it was difficult to comment on recall. We did not know how many tactical comments were missed because of the incomplete class list or incomplete dictionary for a class.

After explaining the details of the preemptive and detection model, we present a case study to illustrate its effectiveness.

4.4 Case Study: IT Support System of a Large Global IT Company

To demonstrate the usefulness of the proposed preemptive model and the detection model, we present a case study on the same organization's data as in case study III (refer to Chapter 3). We conducted experiments on a total of 96,756 closed tickets belonging to the top subcategory (tickets were labeled with specific subcategory), that is, install, within the software category. A total of 57.25% of the tickets had user input requests in the life cycle.

Table 4.1: Number of class-wise data points in the ground truth and test-train split for the proposed preemptive model. Class 1, if the information is asked in the ticket life cycle; Class 0, information is not asked in the ticket life cycle.

Preempted	Ground Truth		Train Set		Test Set	
Information	**Class 1**	**Class 0**	**Class 1**	**Class 0**	**Class 1**	**Class 0**
Awaiting User Inputs	55,398	41,358	20,679	20,679	34,719	20,679
Software Version	1,174	95,582	587	587	586	94,996
Approval	3,686	93,070	1,843	1,843	1,843	91,227
IP Address	2,750	94,006	1,375	1,375	1,374	92,632

4.4.1 Preemptive Model

To learn the model for **P1** (to process the ticket, will there be user input request), tickets were labeled on the basis of *Awaiting User Inputs* state in the life cycle. As shown in Table 4.1, 55398 (57.25%) tickets belonged to class 1, that is, they had at least one user input request in the life cycle. The bias due to tickets with only tactical user input requests in the life cycle affects the outcome of $P1$ because the model is learned to predict the class for a ticket as 1 (i.e., some information will be asked) even if it had just tactical user input requests. It is because such tickets also had *Awaiting User Inputs* state in the life cycle. However, the outcome of $P2$ takes care of this limitation: if $P1$ predicts class as 1, binary classifiers for every information need are executed and all of them give the output as 0 because none of the information was asked for the given ticket. Therefore, the user is not preempted to provide any information. Moreover, only 3117 tickets (i.e., around 6% of total tickets in class 1 for $P1$) just had tactical user input requests and still were assigned ground truth label as 1. Thus such cases were ignored. Effectively, the preemptive model remains independent of detection model and accurately preempts user for the additional information needs.

We observed from the comments that different information was asked by analysts, such as manager approval, operating system, location or cubicle ID, machine ID, IP address, project code, purpose of download, problem screenshot, download URL, software name, and software version. Therefore, the information requested by the analysts can be classified into three categories: information, such as the IP address or the machine ID, that can be derived automatically; information, such as software name or software version, that is explicitly asked in the ticket template; and information, such as the manager approval, that is not explicitly asked in the ticket template but might be requested by the analyst under specific circumstances. To represent each of these categories, we addressed **P2** (what specific information is likely to be asked) for the IP address, software version, and manager approval, corresponding to 3.5%, 1.5%, and 4.8% of the $77,333$ user input requests derived from $55,398$ tickets, respectively. To illustrate the effectiveness of the preemptive model to predict information needs, we chose specifically these three for the aforementioned categories because these were one of the frequent information needs and evaluated as part of the keyword-based annotation (refer to Section 4.2.1); thus, the ground truth was close to reality. Similarly, independent binary classifiers could be trained for preempting other information needs.

The ground truth is labeled for the information needs using a keyword-based approach (as discussed in Section 4.2.1). All the information needs and the corresponding list of keywords were made publicly available [120]; however, labeled data could not be shared because of company policy concerns. We noticed from the ground truth in Table 4.1 that a relatively small percentage of tickets belonged to class 1, that is, the data were imbalanced, likely to overfit to the majority class [150].

The following information was extracted from a user-submitted ticket: description, software name, software version, platform, doc-attached, and time of reporting. Doc-attached is a binary field indicating the presence or absence of an attachment. Platform is a categorical attribute with seven unique values, such as Windows, Linux, and Unix. Time of reporting is mapped to three ranges in a day, that is, morning (before noon), afternoon (from noon to 4 PM), and evening (after 4 PM). Overall, we had 21 possible values for a time corresponding to the 7 days of the week. Description, software name, and software version were free-form text fields. The data from all the three fields for a given ticket were concatenated and preprocessed using case folding, stemming (using the Porter stemmer [129]), and stop words removal. We manually created a stop words dictionary, on the basis of term frequency, for given context publicly available [120]. Also, we removed all the punctuation marks except period because the period is used in the IP address mentioned in the description.

As shown in Table 4.1, the testing and training data were created as per the 50/50 *train/test split protocol* with random undersampling of the majority class (cf. Section 4.2).

Table 4.2: PCA is applied to reduce feature dimension, and a reduced feature vector is used for training. The table presents the number of features before and after applying PCA.

Preempted Information	**#Features before PCA**	**#Features after PCA**
Awaiting User Inputs	1357	493
Software Version	33	17
Approval	119	50
IP Address	96	42

Preprocessed text field for tickets in the training data were represented as a term frequency vector of both unigrams and bigrams. Many unigrams and bigrams have very low frequency adding to the feature sparsity. Thus, we eliminated them by setting the term frequency threshold as 150. We started with a low threshold, tried for random values, such as 50, 100, 150, and 200, and observed that 150 worked the best, given the trade-off between the model computation time and the performance. The reduced feature set of unigrams and bigrams was concatenated with the other three features (platform, doc-attached, and time of reporting) to represent a ticket. Thereafter, we reduced the dimension of the ticket feature vector by applying PCA. We noticed from Table 4.2 that the feature length was different for models corresponding to different information needs because the training data set was different.

Using the list of features and the labeled training data set, an SVM was trained with different kernels, such as linear, polynomial, and radial basis function (RBF) kernel, using LIBSVM [134]. We found experimentally that RBF kernel performed the best with $c = 8$ and $g = 2$, where c and g are the input parameters. A grid search was performed using a validation set and $c = 8$ and $g = 2$ were obtained as the best set of optimal parameters [144]. The performance of the learned classification model is shown in Table 4.3 on the test set using the discussed evaluation metrics in Subsection 4.2.5. Further, the performance of the proposed SVM classifier was compared with some baseline and popular classifiers in the literature, such as naive Bayes, logistic regression, and random decision forest. For logistic regression, the threshold hyperparameter was manually fine-tuned to be 0.5. Breiman *et al.* discussed some experimental heuristics to tune the parameters of RDF [145]. Based on these intuitions, the parameters of RDF used in our experiments were number of trees = 200, bootstrap ratio = 0.7, and subset of features per tree = 0.6. The average results obtained

Table 4.3: Table showing the performance of the prediction model by comparing various popular classifiers with the proposed SVM. The best results are from SVM for any information needed. LR, Logistic Regression; NB, naive Bayes; RDF, random decision forest; SVM, support vector machine.

	Evaluation Metric	Awaiting User Inputs	Version	Approval	IP Address
NB	Accuracy	54.38±0.09	62.19±0.54	70.41±0.29	60.41±0.72
	Precision	53.42±0.08	63.71±2.29	73.54±0.23	63.21±1.24
	Recall	68.48±0.32	57.58±7.03	63.75±1.20	49.92±1.06
	F-Score	60.02±0.13	60.29±4.03	68.29 ± 0.7	55.77±0.82
LR	Accuracy	61.85±0.17	62.84±0.98	72.55±0.53	63.13±0.47
	Precision	62.21±0.11	65.78±1.33	76.98±0.80	65.95±0.52
	Recall	60.37±0.45	53.63±2.91	64.35±0.75	54.30±1.46
	F-Score	61.28±0.24	59.05±1.84	70.1 ± 0.56	59.55±0.91
RDF	Accuracy	99.51±0.01	91.36±0.92	97.73±0.30	96.58±0.42
	Precision	99.83±0.02	94.78±1.45	99.36±0.24	98.39±0.56
	Recall	99.20±0.03	87.60±2.84	96.07±0.75	94.72±0.72
	F-Score	99.51±0.02	91.03±1.68	97.69±0.41	96.52±0.46
SVM	Accuracy	99.83±0.01	94.96±0.69	99.28±0.17	98.69±0.14
	Precision	99.94±0.02	95.59±0.92	99.56±0.23	98.89±0.20
	Recall	99.73±0.02	94.28±1.82	99.00±0.24	98.49±0.26
	F-Score	99.83±0.01	94.92±1.03	97.25±0.17	98.69±0.16

over five-times repeated random subsampling for all the classifiers are tabulated in Table 4.3. The receiver operating characteristic (ROC) curve is presented in Figure 4-2 for SVM and random decision forest classifier as they performed better for all the four models. The major observations drawn from the results were as follows:

1. The proposed SVM classifier provided the best overall classification accuracy in the range of 95%–99% for all the information needs. SVM was expected to perform the best with optimal parameter values as it was regarded as one of the best classifiers in the literature for text classification tasks [27][135][136][137][138][139]. It was observed that both precision and recall of the classifier were high, suggesting that the classifier was not biased toward any particular class. It was possible because random undersampling of the majority class was performed at the time of training to handle an imbalanced class problem.

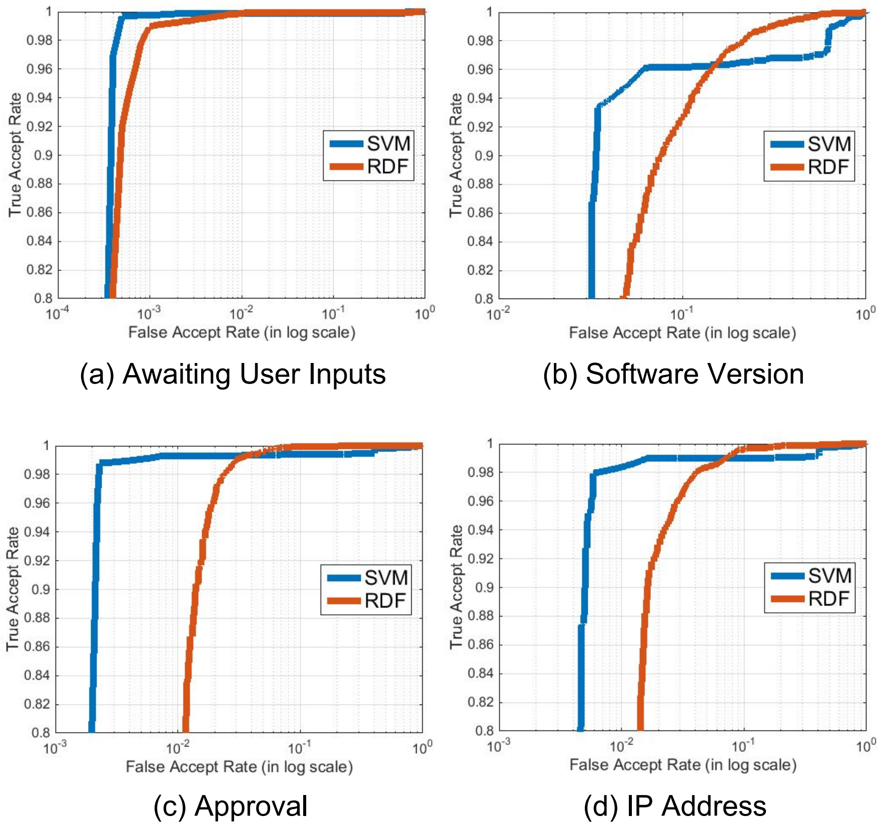


Figure 4-2: ROC for SVM- and RDF-based preemptive model to illustrate their performance for different information needs: (a) awaiting user inputs, (b) software version, (c) approval, and (d) IP address. The y-axis was cut at 0.8 to zoom in the point of bending for ROC curves.

2. It was observed that an ensemble learning-based classifier, such as random decision forest, performed comparable to SVM. Thus, SVM is not a strict choice for choosing the classifier of the preemptive system. SVM performed a kernel trick to project the feature space into a suitable higher-dimensional space where linear classification was possible, while RDF combined the classification results of multiple individual classifiers, making the classification decision robust. The logistic function of the regression classifier tries to fit a linear boundary in the provided feature space, leading to an approximate classification. Hence, logistic regression performs poorly compared with SVM and RDF. As a sparse feature representation is obtained from the bag-of-words model, models such as naive Bayes perform poorly in trying to fit distribution for the data.

3. Figure 4-2 shows the ROC curve plotted between the false accept rate (in log scale)

Table 4.4: Categories in tactical user input requests with total comments in each class, percent of total comments, and example keywords

Class	**#Comments**	**% of Comments**	**Example Keywords**
Temporize	6272	8.11%	In progress, working, will do it
Invalid	645	0.83%	No alphanumeric character
Will Transfer	482	0.62%	Transfer to, assign to
Contact Me	21081	27.26%	Ping me when free, call me @
Done So Check	4414	5.70%	Installed, completed, check

and true accept rates, comparing the performance of SVM and RDF classifiers across all information needs. The ROC curve shows the trade-off between sensitivity (also called recall) and specificity, providing the number of true detects for a given number of fall-outs. For all the information needs, it was observed that SVM performed better than RDF by correctly detecting more than 95% of the test cases.

4. It is to be noted that the test data were unseen data for the classifier. Thus, the performance of the classifier, as shown using the test data, was equivalent to the performance of the classifier as deployed in a real-time environment.

We made the trained preemptive model and code publicly available [120] and they can be used by other researchers in their experiments.

We achieved very high accuracy using SVM and RDF, but it was not overfitted because the test data were different from the training data. This performance was achieved after fine-tuning the parameters to optimal values for given data; otherwise, for some parameter values, the performance was no better than naive Bayes and logistic regression. We evaluated the performance of the presented preemption model for different information needs on real data for a large global IT company. However, the data were from the same organization's IT support information system. Thus, the performance of the preemption model might vary with different ticket dataset, more so because the performance of a classifier strongly depends on the value of its input parameters, whose optimal choice heavily depends on the data being used.

Table 4.5: Real example comments for each category of tactical user input requests

Class	Example Comments
Temporize	"We are trying to find the solution for the problem.We shall get back you soon." "Please provide sometime it will be done asap" "Will check and update the status."
Invalid	"...", "-", " "
Will Transfer	"Will Assigned to L1 Team. They will reach you shortly." "Transferring to concerned person." "This request is not under my scope of work. I will contact admin and transfer it to correct analyst."
Contact Me	"Please ping me when you are available." "Please Ping/Call me once you are at your desk and free so that we can work on your request." "You seem offline. Please contact me once you are available."
Done So Check	"The requested software has been installed. Please check and close the request." "Please check and update." "It has been done. Kindly check it."

4.4.2 Detection Model

A set of rules was derived iteratively for each of the five categories using the approach suggested in Section 4.3. For example, the rule for category transfer was that comment description should be like "*transfer to*" or "*assign to*". Example keywords in generating rules for each category are shown in Table 4.4, and the complete set of rules for reference are made publicly available [120]. The given data set, that is, $77,333$ analyst comments, corresponding to $96,756$ closed tickets for subcategory install were classified using the designed rule-based classifier. We performed stemming and case folding of comments to ensure that matching was case insensitive, and removed special characters. The total number of data points classified to each of the categories is summarized in Table 4.4. Around 42.52% of the total user input requests were classified to one of the five listed categories. Managers expressed that although it was very useful to know comments from the Contact Me category, it needed to be tackled differently compared with other tactical categories. This was because they believed that although no direct information was asked in such comments, the analyst asked the user to contact them. Therefore, there was a high possibility that the analyst asked for inputs in follow-up communication with the user over phone or chat. Hence, whether it is truly tactical or not also depends on the reason for asking the user to contact them, which is not captured in the comment, and therefore cannot be concluded as a clear case of tactical. As a result, comments from the other four categories constituting around 15.27% of the total user input requests are considered as tactical. The most frequent tactical category is *temporize*, constituting 8.11% of the total of user input requests. Interestingly, 645 input requests consist of nonalphanumeric characters such as dash, periods, and NULL. Table 4.5 presents some of the comments from the IT support system, which are classified in the presented classes using the proposed detection model.

For evaluation, we requested 2 managers with experience (as manager in the same organization) of $3-5$ years to independently and randomly pick around 100 comments each from the classified ones. They were requested to make sure that the sample contained comments from all the five categories shown in Table 4.4. They manually inspected the sampled comments and indicated if the comment was wrongly classified to a category. In all cases, the managers agreed that the comment classified to a category indeed belonged to the same. Both the managers mentioned that it was really useful to have categories within tactical because each category might need to be tackled differently. Though the completeness is not guaranteed with this evaluation, the detection model precisely classifies comments to categories of tactical requests, which can be handled accordingly.

The detection model classifies user input requests to refined tactical classes. However, the list of classes is not exhaustive and can vary with the organization. It is possible that an analyst seeks information, which is not really required to resolve a ticket. However, it looks like a real user input request because in the given solution only non-information-seeking user input requests were detected as tactical. Such cases go undetected with the given solution.

4.5 Destination State Analysis

We analyzed the comments classified to one of the tactical categories and the ones not classified to any tactical class (referred to as real user input request) for the destination

Table 4.6: Destination state transition analysis for different types of user input requests

Type of User Input Request / Dest. State	**User Update**			**No Update**		
	User Input Received	**Attach doc**	**Closed**	**Resolved**	**AUI - Auto-closure**	**Transfer**
1. Real User Input Request (44,439)	**18,224 (41.0%)**	4,340 (9.8%)	7,429 (16.7%)	5,678 (12.8%)	4,432 (10.0%)	2,222 (5.0%)
2. Contact Me (21,081)	5,469 (26.0%)	186 (0.9%)	3,093 (14.7%)	**9,202 (43.6%)**	1,143 (5.4%)	1,191 (5.6%)
3. Done So Check (4,414)	941 (21.3%)	58 (1.3%)	**1,517 (34.4%)**	**1,466 (33.2%)**	226 (5.1%)	97 (2.2%)
4. Invalid (645)	160 (24.8%)	4 (0.6%)	29 (4.5%)	**362 (56.1%)**	8 (1.2%)	62 (9.6%)
5. Temporize (6,272)	1,873 (29.9%)	84 (1.3%)	514 (8.2%)	**2,569 (41.0%)**	289 (4.6%)	715 (11.4%)
6. Will Transfer (482)	46 (9.5%)	4 (0.8%)	27 (5.6%)	21 (4.4%)	12 (2.5%)	**350 (72.6%)**

state. Table 4.6 presents the transition of different user input request types to most frequent destination states. We tested if a relationship existed between user input request type and destination state using the chi-square test for independence because both are categorical variables and every cell has an expected value of more than 5. The p-value for the significance test was too low to be computed (less than 0.01). Thus, the two variables were significantly related to each other. From Table 4.6, we made the following observations:

- Most frequent destination state (highlighted with bold) for tactical categories ($2-5$ in Table 4.6) was from No Update, that is, Resolved, AUI-Autoclosure, and Transfer. However, the most frequent destination for real user input requests was from User Update. This validates our conjecture that if no update is obtained from an user for a user input request, then it is more likely to be tactical; and if the user provides any update, then it is more likely to be a real user input request.

- For real user input requests, user inputs were received for the majority (around 41%) of the cases. In some cases (around 17%), tickets were closed by users without getting them resolved.

- For Contact Me AUI, either the user gave some inputs (as destination state is User Input Received for 26.0% comments) or the ticket was Resolved (for 43.6% cases) based on the interaction between the user and the analyst outside the ticketing system (not recorded in the database).

- Done So Check type AUIs have Closed (34.4%) and Resolved (33.2%) as the most frequent destination state. Closed indicates that the user was satisfied with the resolution, hence closed. Many a time, the user confirms with a comment in response to this input request. Hence, User Input Received is also a quite frequent destination state. Apart from this, Resolved is a frequent state indicating that the user did not confirm the resolution and the analyst marked it as Resolved after some time.

- For Invalid AUIs, which are a clear case of tactical input requests, the most frequent destination state is Resolved (around 56.1%). In cases where the destination state is User Input Received, the input from the user is most likely an expression of displeasure or clarification as observed by manually analysing a random set of around 20 such cases.

- For Temporize AUI, maximum transitions are to state Resolved (for around 41.0% times), that is, the analyst actually did not need any information and misused the label. For instances with the destination state as User Input Received, users mostly clarified with analysts the information they are supposed to provide.

- Will Transfer AUI is often followed by the Transfer (for 72.6% times) of ticket with few exceptions.

The p-value (less than 0.01) and aforementioned observations validate our conjecture that destination state gives an indication about the type of user input requests. Therefore, the manager can leverage the transition pattern (output of process mining) to decide if there is need to reduce real or tactical or both user input requests.

4.6 Threats to Validity

The participation by students for the manual annotation of comments was incentivized with a cash gift of 1000 INR, as a token of gratitude. This incentive could have been the motivation for showing interest in the task initially. However, out of nine volunteers, three dropped out after attending the demo and only five out of six actually finished the annotation. The timeline given to the participants was as per their convenience so that they had sufficient time. Given the volunteers always had a choice to drop out and were allowed to finish as per their timeline, we believed that the quality of annotation was not really influenced by the fact that it was incentivized. Further, the author randomly verified 50 annotated comments for each student for a sanity check. Therefore, we believed that the errors in annotation were more likely to be genuine human errors and less likely to be an influence of incentive.

We evaluated the performance of the presented preemptive model for different information needs on real data for a large global IT company. However, the data were from the same organization's IT support information system and thus the performance of the preemptive model might vary with different ticket datasets. This was more so because the performance of a classifier strongly depended on the value of its input parameters, whose optimal choice heavily depended on the data being used. Therefore, the proposed preemptive model might preempt information needs in different contexts less accurately, leading to a smaller reduction in later user input requests.

For preemptive model evaluation, the train/test split is performed randomly and not as sliding window. While the ticket data are time series, the prediction of information needs for ticket resolution does not depend on the timeline. Therefore, we opted for random split. We used one-quarter data during which there were no changes in the ticket resolution process. Further, we did not use timestamp as a feature, but mapped it to three ranges of the day (morning, afternoon, and evening).

The evaluation of the preemptive model was done on the test data but not in production. As the test data were unseen, we believed that the performance was close to reality and the model efficiently reduced user input requests. However, it depends on the way the preemptive model is applied in practice. For instance, if applied as a recommendation system, then there is a performance improvement only if the users choose to provide the preempted information.

The detection model classified user input requests into refined tactical classes. However, the list of classes was not exhaustive and could vary with the organization. While the detection model had high precision, it was difficult to guarantee high recall because of incomplete tactical request classes.

While the ITIS information system is the recommended communication channel between analysts and users, there can be communication over chat and telephone. The data for communication over these channels were not accessible for analysis because of confidentiality and privacy reasons. We distinguished between analysts requesting information outside the ITIS and users providing the information outside the ITIS. The former was unlikely to happen as the SLA clock would not be affected. The latter would not affect the detection model, as the detection model analyzed the comments made when the analysts marked a ticket as Awaiting User Input. However, the validity of the preemptive model might have been affected as follows: The preemptive model inherently reflected the data it was trained upon, and as no information was available about the communication outside the ticket tracking system, the preemptive model might not adequately reflect the information needs expressed in such communication.

4.7 Summary

We analyzed the IT support ticket data in case study III (from Chapter 3) to capture the process reality, especially the user input requests made during the ticket resolution life cycle, by applying process mining. Also, we studied the impact of user input requests on the overall user-experienced resolution time. We observed that around 57% of the tickets had user input requests in the life cycle, causing user experienced resolution time to be almost twice as long as the measured service resolution time. It should be ensured that the information required for ticket resolution is collected from the user upfront, thus reducing real user input requests. However, users do not have a clear idea of what information will be required for resolving a specific ticket. Therefore, in this chapter, an SVM classifier-based preemptive model was learned to preempt users with the need for additional information during the time of ticket submission. The proposed preemptive system preempted the information needs with an average accuracy of 94–99% across five cross validations while traditional approaches such as logistic regression and naive Bayes had accuracy in the range of 50–60%. Also, we noticed non-information-seeking tactical user input requests for the sake of service-level compliance. The rule-based detection model identifies such input requests, and thus can be discouraged. The detection system identified around 15% of the total user input requests as tactical. The performance of the proposed preemptive model and detection model on the real-world data for a large global IT company shows the effectiveness of our solution approach in reducing the number of user input requests in tickets' life cycle.

Reduction in ticket resolution time depends on the application of a preemptive and detection model. Thus, we cannot estimate the effective reduction in resolution time. However, given the observation that user input requests cause user-experienced resolution to be much higher than measured service resolution time, reduction in user input requests definitely leads to a significant reduction in the resolution time. In the existing approach, every information-seeking user input request is considered as real irrespective of whether it is really required to resolve the ticket or not. In the future, we plan to extend the detection model to also identify the cases where unnecessary information is asked, which is another way of making tactical requests. It requires further investigation of user input requests and understanding of information actually being used for resolving the ticket.

Chapter 5

Analyzing Comments in Ticket Resolution Process to Capture Underlying Process Interactions

Activities in the ticket resolution process have comments associated with them. Process discovery, as discussed in Chapter 3, is activity focused, that is, it uses structured logs and does not analyze the comments. However, comments can provide additional information for performing the activity efficiently. One of the problems identified from the survey, as listed in Table 2.1, is "facilitate in-depth understanding of points where things went wrong by deriving and understanding actual process at a more granular level", that is, $P20$. In this study, we aimed at discovering the detailed process model for ticket resolution process, using the information present in the comments. The discovered model can then be used to identify the inefficiencies.

To model the detailed process, we extracted the topical phrases (keyphrases) from the comments generated during the process execution, using an unsupervised graph-based approach. These keyphrases were then integrated into the event log to derive enriched event logs. A process model was discovered using the enriched event logs wherein keyphrases were represented as activities, thereby capturing the flow relationship with other activities and the frequency of occurrence. This provided insights that could not be obtained solely from the structured data (i.e., activities), and these insights could be used to perform the ticket

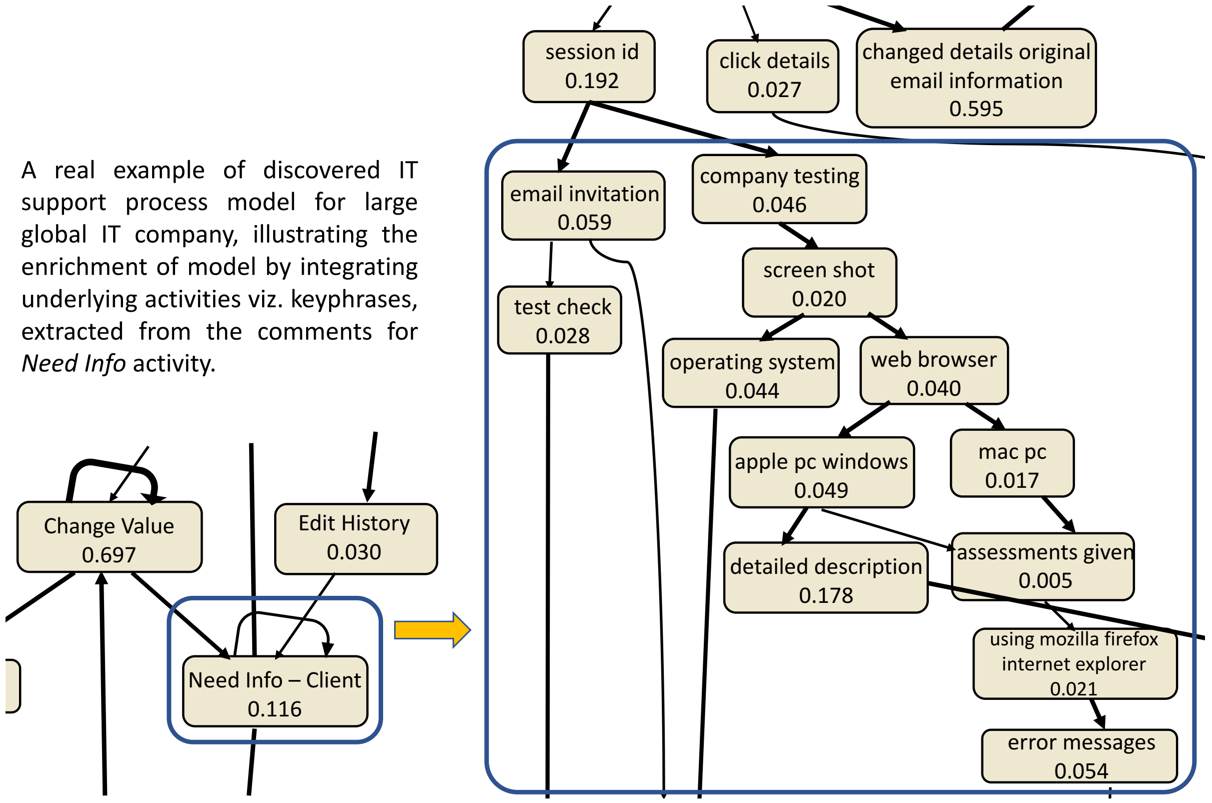

A real example of discovered IT support process model for large global IT company, illustrating the enrichment of model by integrating underlying activities viz. keyphrases, extracted from the comments for *Need Info* activity.
Change Value 0.697
Edit History 0.030
Need Info – Client 0.116
session id 0.192
click details 0.027
changed details original email information 0.595
email invitation 0.059
company testing 0.046
screen shot 0.020
test check 0.028
operating system 0.044
web browser 0.040
apple pc windows 0.049
mac pc 0.017
detailed description 0.178
assessments given 0.005
using mozilla firefox internet explorer 0.021
error messages 0.054

Figure 5-1: A real motivating example that compares the process model from structured logs with the model with underlying activities as per the keyphrases extracted from comments.

resolution process more efficiently. To evaluate the approach, we conducted a case study on the ticket data of a large global IT company. We first extracted the keyphrases from the comments associated with the ticket activities with an average accuracy of around 80%. This enabled us to succinctly capture the additional information about the activities influencing the ticket resolution process and often causing delays, such as extra information required, priority, and severity. The model allowed the managers to understand in detail the process realities and identify opportunities for improvement. In this case, for example, the manager identified that having a bot to capture the information or adding a mandatory field in the initial ticket template, so as to reduce the delays incurred while waiting for information, can reduce the time (he subsequently had his team implement the bot).

5.1 Usefulness of Information in Comments

A lot of rich information is present in the comments generated during the process execution, which needs to be integrated into the discovered process model for in-depth process understanding. The in-depth unstructured data-driven (e.g., comments) insights help effectively identify the inefficiencies and make informed process improvement decisions.

Figure 5-1 shows the snapshot of a real example of a discovered process model snapshot for the ticket resolution process of a large global IT company. As part of the ticket resolution process, an analyst (person responsible for servicing the ticket) can ask the user to provide additional information by writing a comment, which gets captured in the information system as an event, *Need Info - Client*. An analyst can ask for different information, such as error messages and operating system, which gets recorded in the comments. A process model is discovered using only a structured event log containing a single activity, *Need Info - Client* (refer to Fig. 5-1, left panel). We extracted the keyphrases from all the comments corresponding to the activity *Need Info - Client*, using an unsupervised graph-based keyphrase extraction approach. The extracted keyphrases represented information typically asked by analysts, using which an enriched event log was derived where the activity, *Need Info - Client*, was mapped to relevant keyphrases on the basis of the comment. The process model discovered using the enriched event log (refer to Fig. 5-1 - right panel) presented the underlying interactions of the process with activities such as email invitation, screenshot, web browser, operating system and error message, each corresponding to an information asked by the analysts (highlighted in Figure 5-1, right panel). This allowed discovering the in-depth reality, which cannot be observed from Figure 5-1, left panel. Thus, the following informed improvement decisions could be made to mitigate the delays incurred while waiting for information from the user:

- As *detailed description* (relative frequency is 0.178) is asked more often, the IT company should deploy a system such that a user can be preempted at the time of ticket submission to provide the same upfront [151] (as discussed in Chapter 4).

- A user is typically asked to provide an error *screenshot* after asking for *company testing*; therefore, analysts can be preempted to ask both the screenshot and the company

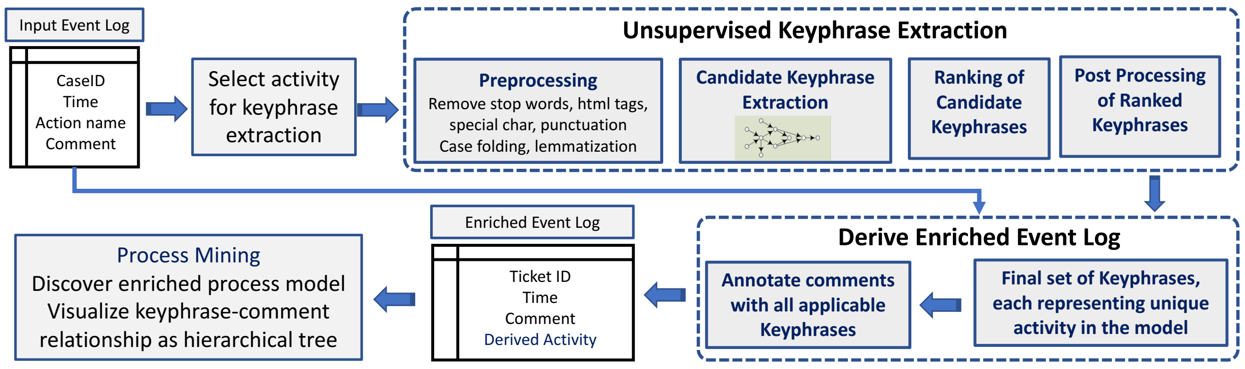

Input Event Log
CaseID
Time
Action name
Comment
Select activity for keyphrase extraction
Unsupervised Keyphrase Extraction
Preprocessing
Remove stop words, html tags, special char, punctuation
Case folding, lemmatization
Candidate Keyphrase Extraction
Ranking of Candidate Keyphrases
Post Processing of Ranked Keyphrases
Derive Enriched Event Log
Final set of Keyphrases, each representing unique activity in the model
Annotate comments with all applicable Keyphrases
Enriched Event Log
Ticket ID
Time
Comment
Derived Activity
Process Mining
Discover enriched process model
Visualize keyphrase-comment relationship as hierarchical tree

Figure 5-2: Proposed approach to discover the underlying process interactions using comments.

testing at the same time.

- As information about the *operating system* and *web browser* is asked in the comments, a bot can be designed to automatically detect this information at the time of ticket submission.

This example highlights the potential of our approach for effective process improvement, by deriving keyphrases from comments corresponding to activities and representing them as part of the discovered process model.

5.2 Proposed Approach

To achieve the objective of integrating knowledge captured in unstructured data, namely comments into the discovered process model, we presented an approach consisting of multiple steps, as shown in Figure 5-2. First, a process analyst can select the activity for which the comments should be analyzed for in-depth process understanding. Performing this selection is important because a granular view of every activity can make the discovered process model look like spaghetti. Also, it needs to be decided on the basis of analysis to be performed, such that activities not captured in the structured logs are inferred from the comments. Thereafter, the comments corresponding to selected activities are preprocessed and used for candidate keyphrase extraction in an unsupervised manner. Extracted candidates are ranked and processed to select most relevant keyphrases which in turn are used to annotate the comments, thus, deriving an enriched event log. Finally, using the enriched event log, the process model capturing the flow relationship and frequency is discovered.

5.2.1 Unsupervised Keyphrase Extraction

Keyphrase extraction aims at automatic selection of important and topical phrases from the body of documents [152]. Automatic keyphrase extraction is used for a wide range of natural language processing and information retrieval tasks such as text clustering and summarization [153][154], text categorization [155], and interactive query refinement [152]. However, the application of keyphrase extraction to business process model enrichment is not explored.

Broadly, keyphrases are extracted using two approaches: supervised and unsupervised. In the supervised approach, a model is trained to classify a candidate keyphrase, requiring human-labeled keyphrases as training data. It is impractical to label training data (in this case, process execution comments), given the effort required for manual labeling. Thus, we focused on the unsupervised approach for our purpose. Unsupervised approaches can be grouped as follows [156]: graph-based ranking, topic-based clustering, simultaneous learning, and language modeling. Graph-based ranking methods are state-of-the-art methods [157], based on the idea of building a graph from the input document. Nodes in the graph are ranked based on their importance to select the most relevant keyphrases. Therefore, we used CorePhrase [158], a graph-based algorithm for topic discovery, that is, keyphrase extraction from multidocument sets based on frequently and significantly shared phrases between documents. The algorithm is domain independent and thus suitable for our purpose with some adaptations. The algorithm first identifies a list of candidate keyphrases from the set of documents and then selects top n-ranked keyphrases for the output by using a ranking criterion. The ranked keyphrases are then postprocessed to be adapted according to the domain.

Preprocessing of Comments: The comments from event logs are preprocessed (*Preprocess* in Algorithm 3) including stemming, case folding, removal of HTML tags, stop words, and special characters. Further, we created a set of unique sentences across all the comments to improve the scalability of the approach. The sentences in the comment were demarcated

Algorithm 3: Unsupervised Keyphrase Extraction

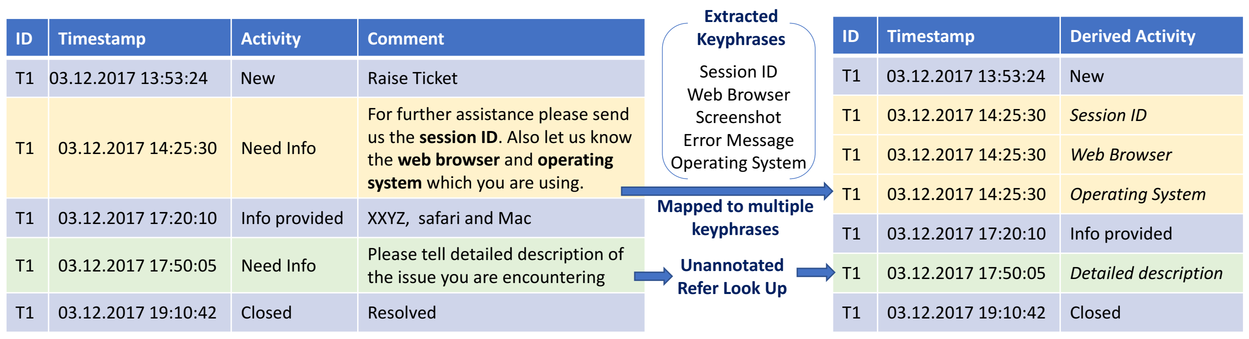
```
1  Input: Initial Event Log EL (CaseID, timestamp, Activity, Comments)
2  Output: Keyphrase List K for Selected Activity A
3  Variables: $Candidate\ keyphrase\ list : M \leftarrow [], lookup_{table} \leftarrow [], score \leftarrow [],$
     $Global\ Graph : G \leftarrow []$
4  $comments \leftarrow SelectActivity(EL, A)$
5  $sentences \leftarrow Preprocess(comments)$
6  $G \leftarrow GlobalGraph(sentences)$
7  for $sentence$ in $sentences$ do
8      $G_c \leftarrow Graph(sentence)$
9      $G_u \leftarrow G - edges(G_c)$
10     $phrases \leftarrow G_u \cap G_c$
11     $M \leftarrow M \cup phrases$
12     $lookup_{table}[sentence] \leftarrow phrases$
13 for $m$ in $M$ do
14     $pf = PhraseFrequency(phrase)$
15     $cf = CommentFrequency(phrase)$
16     $score[m] \leftarrow pf \times -log(1 - cf)$
17 $score_n \leftarrow Sort(score, n)$
18 $K \leftarrow PostProcess(score_n)$
19 return K
```

by a period. A unique set was identified from the initial set of sentences corresponding to all the comments. This significantly reduced the number of sentences to be processed in further steps because the sentences were repeated across various comments (emails). This preprocessing did not affect the final set of extracted keyphrases because the keyphrase for a comment was a set of keyphrases extracted for its constituting sentences.

Candidate Keyphrase Extraction: To extract candidate keyphrases, the algorithm compares every pair of sentences to identify the common phrases. If there are n sentences in the corpus, comparing every pair is inherently $O(n^2)$. However, as highlighted in the CorePhrase [158] algorithm, using a data structure called the *Document Index Graph*(DIG), the comparison can be done in approximately linear time [159]. For our purpose, the DIG stored a cumulative graph representing the entire set of unique sentences (e.g., *GlobalGraph* function in Algorithm 3). When the keyphrase for a sentence has to be extracted, its subgraph is matched (by performing graph intersection) with the cumulative graph (viz. Global Graph) except for the sentence (Line 9 and 10 in Algorithm 3). It gives a list of matching phrases between the sentence and the rest of the sentences. This process generates matching phrases between every pair of sentences in near-linear time with varying length phrases. A master list M is maintained that contains unique matched phrases for all sentences that will be used as a list of candidate keyphrases. A *lookuptable* is also maintained that contains sentences and the corresponding matching phrases (Line 12 in Algorithm 3), which can be used for annotation (discussed in Section 5.2.2).

Ranking of Candidate Keyphrases: Quantitative phrase metrics are used to calculate the *score* representing the quality of the extracted candidate keyphrase. The *score* is computed as $pf \times -\log(1-cf)$, where cf is the comment frequency and pf is the average phrase frequency. Inspired by term frequency-inverse document frequency (TF-IDF) [158], the *score* rewards the phrases that appear in more documents (high cf) rather than penalizing them. For a phrase p, the comment frequency $cf(p)$ is the number of comments in which p appears, normalized by the total number of comments: $\frac{|\text{comments containing } p|}{|\text{all comments}|}$. The average phrase frequency pf is the average number of times p appears in one comment, normalized by the length of the comment in words: $\arg\text{avg}[\frac{|\text{occurrences of } p|}{|\text{words in comment}|}]$

Postprocessing of Ranked Keyphrases: Selected top-ranked keyphrases contain some nonrelevant phrases, that is, phrases that fall out of the domain but still are common in most of the comments. Examples of such phrases are *thank you for contact,* and *contact helpdesk*. Such keyphrases are removed by creating a common domain dictionary that contains unwanted words to be removed from the keyphrases (function *PostProcess* in Algorithm 3). This domain dictionary thus postprocesses the keyphrases to obtain the final set of keyphrases. The following is an example of how postprocessing is applied to the keyphrases:
Extracted Phrase: Please tell unemployment benefits
Postprocessed Phrase: Unemployment benefits
Here words *please* and *tell* belong to an unwanted dictionary, as they are not relevant in keyphrases and thus removed. Also, if a keyphrase is a proper subset of any other selected keyphrase, then it is removed to resolve the spurious multilabel assignment.



ID	Timestamp	Activity	Comment
T1	03.12.2017 13:53:24	New	Raise Ticket
T1	03.12.2017 14:25:30	Need Info	For further assistance please send us the **session ID**. Also let us know the **web browser** and **operating system** which you are using.
T1	03.12.2017 17:20:10	Info provided	XXYZ, safari and Mac
T1	03.12.2017 17:50:05	Need Info	Please tell detailed description of the issue you are encountering
T1	03.12.2017 19:10:42	Closed	Resolved

ID	Timestamp	Derived Activity
T1	03.12.2017 13:53:24	New
T1	03.12.2017 14:25:30	*Session ID*
T1	03.12.2017 14:25:30	*Web Browser*
T1	03.12.2017 14:25:30	*Operating System*
T1	03.12.2017 17:20:10	Info provided
T1	03.12.2017 17:50:05	*Detailed description*
T1	03.12.2017 19:10:42	Closed

Figure 5-3: Example to illustrate comment annotation for deriving an enriched event log.

5.2.2 Annotating Comments with Keyphrases to Derive Enriched Event Log

The initial event log, EL, contains activities and corresponding comments. Once the keyphrases are extracted, each comment in the dataset is analyzed to determine whether one of the keyphrases matches with it. To make the matching consistent, we performed the same pre-processing as mentioned earlier. If a comment contained a keyphrase, it was annotated with the corresponding keyphrase. As we only extracted the top n most relevant keyphrases, some comments might be annotated by one or more keyphrases, while other comments might not be annotated at all. To tackle the latter cases, we referred to the *lookup* table and retrieved the keyphrases for that comment. These keyphrases were added as labels to the comment. Therefore, maintaining the *lookup* table helped in assigning labels to otherwise unannotated comments.

Figure 5-3 depicts a real example of an event log where the first comment for the activity, *Need Info*, is mapped to three keyphrases. However, the second comment is not annotated with any of the extracted keyphrases and, therefore, is mapped to *detailed description* after referring to the lookup table. The enriched event log is generated with a new attribute, *derived activity*, replacing *activity* and *comment* and representing the extracted keyphrases.

5.2.3 Evaluating Keyphrase Extraction and Annoation

We evaluated the quality of extracted keyphrases manually, that is, checked whether they conveyed the required information. We asked two managers from the IT support team of the company to indicate whether the extracted keyphrases conveyed the important content of comments.

Further, we needed to evaluate the annotation of comments. As discussed in Section 5.2.2, a comment can be mapped to multiple keyphrases. Therefore, we used multilabel evaluation metrics that could be *example-based* or *label-based* [160]. We chose the *example-based evaluation metrics* that could capture the average difference between the predicted labels and the actual labels for each test example, and then averaged over all examples in the test set. Thus, unlike *label-based evaluation metrics*, these metrics took into account the correlations among different classes [161], which is of interest here. To evaluate the quality of the classification (here, annotation of comments) into classes (here, keyphrases), we used the following set of metrics, thus capturing the partial correctness [160]:

Let T be a multilabel dataset consisting of n multilabel examples $(x_i, Y_i), 1 \leq i \leq n, (x_i \in X, Y_i \in Y = \{0,1\}^k)$, with a labelset $L, |L| = k$. Let h be a multilabel classifier (here, annotator in Section 5.2.2) and $Z_i = h(x_i) = \{0,1\}^k$ be the set of label memberships predicted by h for the data point (i.e., comment) x_i.

$$Accuracy, A = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Y_i \cup Z_i|} \tag{5.1}$$

$$Recall, R = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Y_i|} \tag{5.3}$$

$$Precision, P = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Z_i|} \tag{5.2}$$

$$F_1 = \frac{1}{n}\sum_{i=1}^{n}\frac{2|Y_i \cap Z_i|}{|Y_i| + |Z_i|} \tag{5.4}$$

$$HammingLoss, HL = \frac{1}{kn}\sum_{i=1}^{n}\sum_{l=1}^{k}[I(l \in Z_i \wedge l \notin Y_i) + I(l \notin Z_i \wedge l \in Y_i)], \tag{5.5}$$

where I is the indicator function which is equal to 1 if $Z_i = Y_i$, else 0. Since *HL* is a loss function, it should be minimum for better performance.

5.3 Case Study: IT Support Ticket Resolution Process

To illustrate the value of integrating knowledge from unstructured data into the discovered process model, we performed a case study on the IT support process data of a large global IT company. The dataset represented interactions between the users and the support team (analysts), and thus, comments were present with relevant activities. While the IT support process was continuously

Table 5.1: Experimental Results where **K**: Total extracted keyphrases in final set, **L**: average number of labels for each comment, **P**: Precision, **R**: Recall, **F1**: F1 measure, and **HL**: Hamming Loss.

Data	**K**	**L**	**P**	**R**	**A**	**F1**	**HL**
Change Value	33	3.63	84.74 %	81.27%	80.26%	82.12%	8.15%
Need Info	16	4.28	84.71%	89.97%	80.21%	86.57%	6.21%

monitored by the process analyst, the unstructured data, for example, comments, were not taken into account.

Data extracted from the organization's ticket system includes the required information about a ticket starting from the time of ticket submission until it is closed. Downloaded data consists of 2620 tickets with 15,819 events in total. We observed from the dataset that two activities (out of 19), *Change Value* and *Need Info*, existed where analysts wrote comments. The number of events with the activities *Change Value* and *Need Info* was 4036 and 280, respectively.

In *Change Value*, changes in the ticket attributes were captured by a descriptive comment as shown below with an anonymized example (for confidentiality):

`Changed Category from "" to "Y". Changed Sub-Category from "" to "test reset". Changed Severity from "" to "Sev 4". Changed Summary from "" to "reset the test". Changed Support Contract from None to Contract1.`

An analyst asks information from the user (here, customer) by writing a comment, which is captured as activity *Need Info* in the database. For example,

`Dear ABC, Thank you for contacting the Support Center. In order to assist you more effectively we ask that you please provide the following information: Are you using a Macintosh Computer (Apple) or a PC (Windows)?: What web browser are you using (Internet Explorer, Mozilla Firefox, Safari, Google Chrome)?: Website you were directed to access: Session ID/login info: Detailed description of the issue you are encountering: Screen shot of error message: Thank you in advance!`

To enrich the event logs, we performed keyphrase extraction for these two activities. This allowed us to precisely capture what values were changed and what information was typically asked by the analysts, in coherence with the complete process flow. IT support data were not made publicly available for confidentiality reasons; however, examples and results were included for an explanation.

5.3.1 Unsupervised Keyphrase Extraction and Enriched Event Log Derivation

Comments for the selected activities, namely, *Change Value* and *Need Info*, were preprocessed. All the preprocessing steps as discussed in Section 5.2.1 were performed and the final set of preprocessed unique sentences was used for candidate keyphrase extraction. As per Algorithm 3, a set of candidate keyphrases is extracted for all the activity sets. Extracted keyphrases were ranked using the scoring function. We selected top 50 keyphrases from the ranked list that were postprocessed as per the data properties. This postprocessed set of final keyphrases was used for annotating the comments as discussed in Section 5.2.2, thus generating enriched event logs . All the data sets (each corresponding to an activity) had some unannotated comments for which we referred to the lookup table, and hence assigned keyphrase. Effectively, the total number of unique keyphrases (K) in the resulting enriched event log was 33 and 16 for *Change Value* and *Need Info*, respectively (refer to Table 5.1). The average number of keyphrases, $L \simeq 4$ for *Change Value* and *Need Info* indicated that multiple important topics were present in a comment, that is, multiple ticket attributes were changed and multiple information was asked in the same comment.

5.3.2 Visualizing and Analyzing an Enriched Process Model

Enriched event logs are used for process discovery using ProM. We presented and compared *Need Info* for the original and enriched process model of IT support process in Figure 5-1. Here, we presented the process model snapshot for IT support process, specifically highlighting the *Change Value* derived activities (refer to Fig. 5-4).

As shown in Figure 5-4, the individual activity *Change Value* was replaced with more specific activities such as *changed category*, *changed company name*, and specific instances of *changed summary*, each corresponding to extracted keyphrases.

As the information captured in comments is integrated into the model, it is possible to derive insights as follows:

- The category was changed for a high percentage of tickets (as the relative frequency was 0.615), highlighting the need for a system to automatically assign a category based on the content of the initial ticket, thus optimizing the time spent for category assignment.
- The summary was changed for various tickets, and some of the most frequent instances were

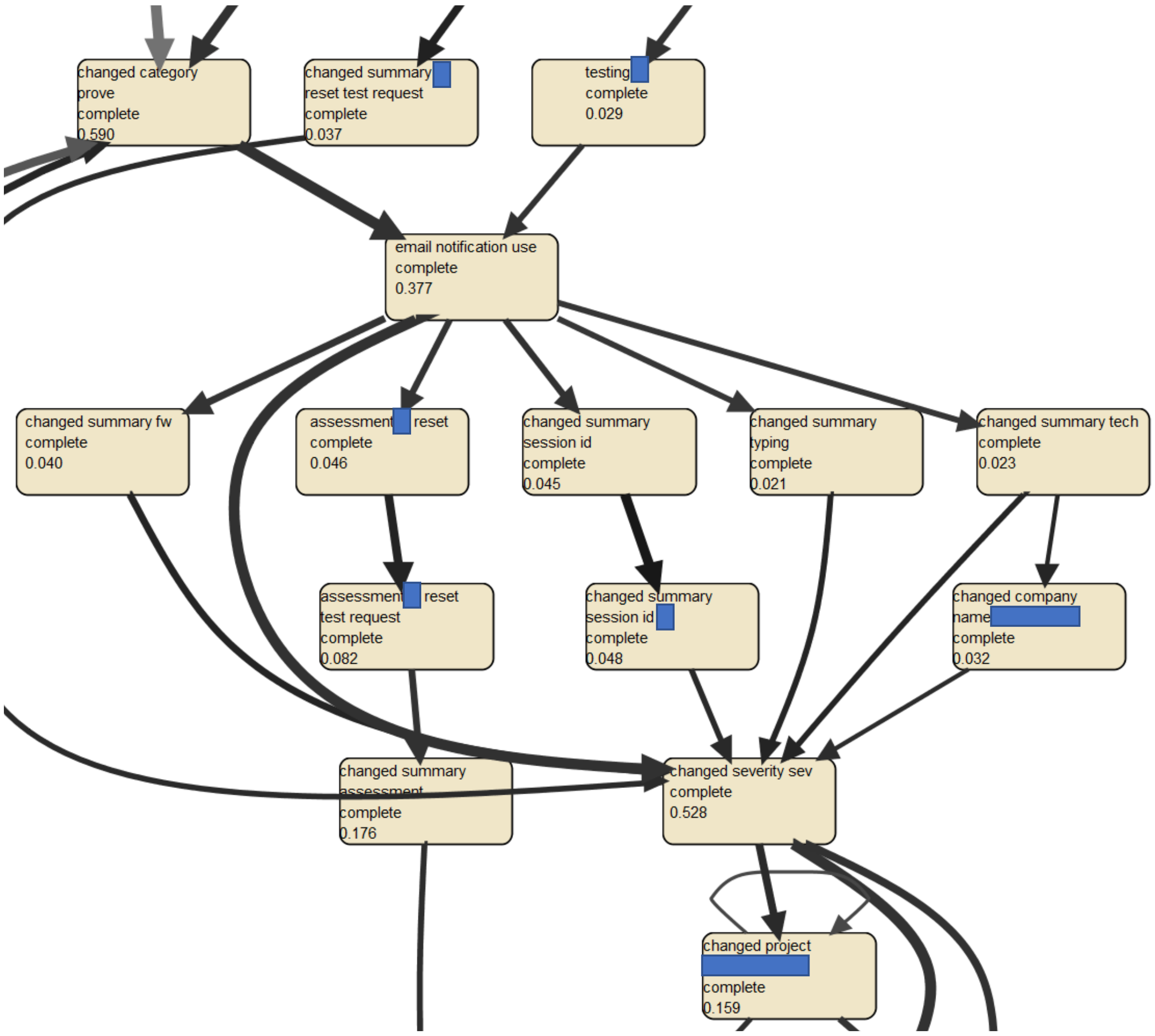

changed category prove complete 0.590
changed summary [redacted] reset test request complete 0.037
testing [redacted] complete 0.029
email notification use complete 0.377
changed summary fw complete 0.040
assessment [redacted] reset complete 0.046
changed summary session id complete 0.045
changed summary typing complete 0.021
changed summary tech complete 0.023
assessment [redacted] reset test request complete 0.082
changed summary session id [redacted] complete 0.048
changed company name [redacted] complete 0.032
changed summary assessment complete 0.176
changed severity sev complete 0.528
changed project [redacted] complete 0.159

Figure 5-4: A real example of discovered IT support process model for a large global IT company, illustrating enrichment of model by integrating keyphrases extracted from the comments for the *Change Value* activity. Some words are masked for confidentiality.

captured as keyphrases in our approach. Hence, we observed many states with a changed summary (suffixed with specific terms), although they all indicated some change in the summary.

- The company name was changed for a small percentage of tickets, which usually happened after the summary was changed in a specific manner. Therefore, the analysts could be preempted in those instances to change the company name in parallel with the summary, thus eliminating the delay.

We showed the process model discovered using structured logs and the enriched discovered process model side-by-side to the manager. He acknowledged that the enriched model helped in making effective process improvement decisions. One of the actionable insights he decided to take forward was to design a robotic process automation solution for the automatic category assignment to a ticket. This could not have been possible without integrating knowledge from the comments into the discovered process model.

5.3.3 Evaluation of Keyphrase Extraction and Annotation

Establish the Ground Truth: To evaluate keyphrase extraction (as discussed in Section 5.2.3), we established a ground truth for comments corresponding to the selected activities. Thus, we needed to first manually identify a set of keyphrases for comments corresponding to selected activities and annotate comments with the same. First, we identified the ticket attributes typically changed (as part of *Change Value* activity) and information typically asked by the analysts (as part of *Need Info* activity) on the basis of managers' domain knowledge and manual inspection of the comments. Manual inspection was performed by the author and her colleague for a disjoint set of comments (random sample of around 25% comments for each) to identify lists of changed attributes and asked information, respectively. Lists by both of them were compared to create a consolidated list. Author and her colleague used different terms to represent the same information, which were made consistent. Both of them identified the same list with a few exceptions (i.e., rarely occurring content), which were resolved. The final list was verified with the manager. Each item in the list was considered as a keyphrase for the respective data set.

Now that the list of ground truth keyphrases was identified, to establish the ground truth , comments were annotated with keyphrases using a keyword-based dictionary [127]. A list of keywords corresponding to each keyphrase was prepared iteratively, for example, the keyword "summary" for the keyphrase "changed summary". If the comment contained keywords, it was annotated with the corresponding keyphrase. Thereafter, the author and her colleague of the paper manually investigated the disjoint set of randomly selected comments to distill the wrongly annotated comments. This process was repeated two to three times until very few/no updates were made in the set of keywords.

As an example, ground truth keyphrases for *Change Value* and *Need Info* were {Changed Category, Changed Sub-Category, Changed Severity, Changed Summary, Changed Support Contract}, and {mac pc, web browser, website directed, session id, detailed description, screen shot}, respectively.

Analysis of Results: Automatically extracted keyphrases can be structurally different from the human-identified ones, although both represent the same topical information. To avoid spurious penalty on the metrics, we took this into account by manually creating a mapping between the two. Table 5.1 shows that the proposed approach performed with an accuracy of around 80% and had a low hamming loss. High F_1 measure ensured that the approach achieved a good balance between precision and recall. Hence, the proposed approach efficiently derived an enriched event log across for the given data set.

5.4 Threats to Validity

An evaluation performed in the case study involved a comparison against manually annotated comments, which could be prone to human error. We believe that as two people (author and her colleague) performed the annotation and verified it with the managers, the evaluation metrics were close to reality.

Performance of the proposed keyphrase extraction annotation approach will vary in other contexts because it depends on text pre-processing and the value of input parameters, namely the number of selected top keyphrases, whose optimal choice heavily depends on the data being used. Further, the approach does not leverage semantics during keyphrase extraction and annotation, which can be resolved using a semantic-based approach. Still we achieved high accuracy for the presented case studies because the communication in business processes, such as IT support, software issue resolution, and customer support, is quite formal. Specifically, the comments written by support agents and developers have consistent terminology to represent the same information, driven by the domain.

When the event log is enriched, an activity is replaced with a set of activities, each corresponding to an extracted keyphrase. This can make the discovered process model look like spaghetti. Therefore, it is recommended to enrich the event log based on the analysis to be performed and limit the number of extracted keyphrases (top n) accordingly.

5.5 Summary

Process mining techniques are activity focused and do not consider comments generated during process execution. We presented a multistep approach to integrate hidden knowledge captured in unstructured text, namely comments, into the discovered process model. This was achieved by extracting keyphrases in an unsupervised manner and using them to annotate the comments thus deriving enriched event logs. We observed that the keyphrase extraction and annotation approach performed with an average accuracy of around 80% across different data sets. Further, we discovered the process model using a derived enriched event log and highlighted the value of enhanced process model in deriving actionable insights.

Our future plan is to extend the keyphrase extraction approach, such that the semantics is leveraged, and compare it with the proposed approach to analyze whether the insights derived from the discovered business processes are further enhanced.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Identifying Changes in Runtime Behavior of a New Release to Facilitate Anomaly Detection

Some code changes are made to resolve a ticket. This change can lead to anomalies, such as regression bugs. We aimed *to detect whether ticket resolution could cause some anomalous behavior, so as to reduce the post-release bugs.* It was one of the top five problems identified in the survey, that is, "Enable early detection and prevention of defects instead of fixing them during the later stage by understanding patterns of escaped defects" (refer to $P2$ in Table 2.1). We have already seen the effect of applying process mining to the software repositories for improving the ticket resolution process. Next, we investigated the usefulness of process mining to monitor the execution behavior of a new release and thus detect inconsistencies introduced due to code changes.

We proposed an approach to automatically discover application Execution Behavior Models (EBMs) for the deployed and the new version using the unstructured text of the execution logs, that is, the print statements generated during the software execution. The differences between the two models were identified and enriched such that spurious differences, for example, due to logging statement modifications, are mitigated. The differences were visualized by identifying the diff regions within the discovered behavior model. This allowed to efficiently analyze the differences for various purposes such as anomaly detection, release decision making, and dynamic profiling.

To evaluate the proposed approach, we conducted a case study on Nutch, an open-source application, and an industrial application. Nutch is an open-source web crawler software project.

Its source code, commit history, and issues data are publicly accessible. This allowed us to control the logging level, investigate code changes across the commit, and generate execution logs. It was possible to generate execution logs by passing a set of URLs as input, thus making it feasible to perform the case study. We discovered the EBMs for the two versions of applications and identified the diff regions between them. By analyzing the regions, we detected bugs introduced in the new versions of these applications. The bugs were reported and later fixed by the developers, thus, confirming the effectiveness of our approach.

6.1 Need for Monitoring Changes in Runtime Behavior

To ensure a high-quality release, the upcoming release was staged in the production environment using strategies, such as blue-green deployment [162], dark launches [163], canary release [162][164], and shadow testing [165], and its performance was monitored [166][162] to quickly identify whether it was misbehaving [165][164]. A vast amount of data were logged during the execution of the new and previously deployed software versions. The existing monitoring systems kept track of suspicious events in logs (e.g., errors, warning messages, and stack traces) and raised alerts. However, such systems did not leverage the unstructured data captured in the execution logs to efficiently derive and compare the dynamic behavior of the new and the previously deployed versions in a holistic manner.

Execution logs have been extensively studied in contexts such as anomaly detection [167][168], identification of software components [169], component behavior discovery [170], process mining [171], behavioral differencing [172], failure diagnosis [173], fault localization [174], invariant inference [175], and performance diagnosis [176][164].

Krka *et al.* [177] proposed algorithms, such as CONTRACTOR++, State-enhanced k-tails (SEKT), and Trace-enhanced MTS inference (TEMI), that make use of inferred value-based program invariants to aid the construction of an FSA from execution traces. Le *et al.* [178] proposed SpecForge, a specification mining approach that synergizes many existing specification miners. SpecForge decomposes FSAs that are inferred by existing miners into simple constraints (i.e., model fission). It then filters the outlier constraints and fuses the constraints back together into a single FSA (i.e., model fusion). Le *et al.* [179] proposed Deep Specification Miner, an approach that uses deep learning for mining FSA-based specifications. This approach uses test case generation to create a set of execution traces for training a Recurrent Neural Network-Based Language Model

(RNNLM). Although these approaches mine execution traces, the focus is on mining specifications and not execution behavior models for runtime monitoring.

Jamrozik *et al.* [180] proposed an approach, Boxmate, to mine sandboxes and, thus, prevent Android apps from suspicious behaviors. Boxmate uses an automated test case generation tool and records the occurrences of sensitive API methods and input parameters. Le *et al.* [181] proposed an approach to create more effective sandboxes that could distinguish malicious and benign activities during the execution of Android apps. All these approaches use test case generation for creating execution logs and do not focus on identifying runtime behavior when the code is changed across the versions.

Goldstein *et al.* [172] analyzed system logs, automatically inferred Finite State Automata, and compared the inferred behavior to the expected behavior. However, they worked on system logs with predefined states, while we identified these states (templates) first. Cheng *et al.* [182] proposed to extract the most discriminative subgraphs that contrasted the program flow of correct and faulty execution. Fu *et al.* [176] derived a Finite State Automata to model the execution path of the system and used it to detect anomalies in new log sequences. However, these were supervised approaches assuming the presence of ground truth for correct and faulty executions to learn a model. Nandi *et al.* [168] detected anomalies by mining the execution logs in distributed environment; however, anomalies were detected within the same version, without differentiating between the flow graphs of the two versions. Tarvo *et al.* [164] automatically compared the quality of the new version with the deployed version using a set of performance metrics, such as CPU utilization and logged errors however, they did not detect the differences in execution flow, which was crucial for finding discrepancies. A set of techniques were used to compare multiple versions of an application. Ramanathan *et al.* [183] considered program execution in terms of memory reads and writes and detected the tests whose execution behavior was influenced by these changes. Ghanavati *et al.* [184] compared the behavior of two software versions under the same unit and integration tests. According to them, if a test failed in the new version, a set of suspicious code change was reported. This approach worked best when comprehensive test suites were available. In this chapter, we present a novel approach to automatically detect discrepancies in the fast-evolving applications, which was achieved by identifying the differences in the runtime behavior of the deployed and the new version, derived by mining the execution logs.

Real Example from Nutch: As part of an issue *[NUTCH-1934]*[1], the class *Fetcher* counting

[1]https://issues.apache.org/jira/browse/NUTCH-1934

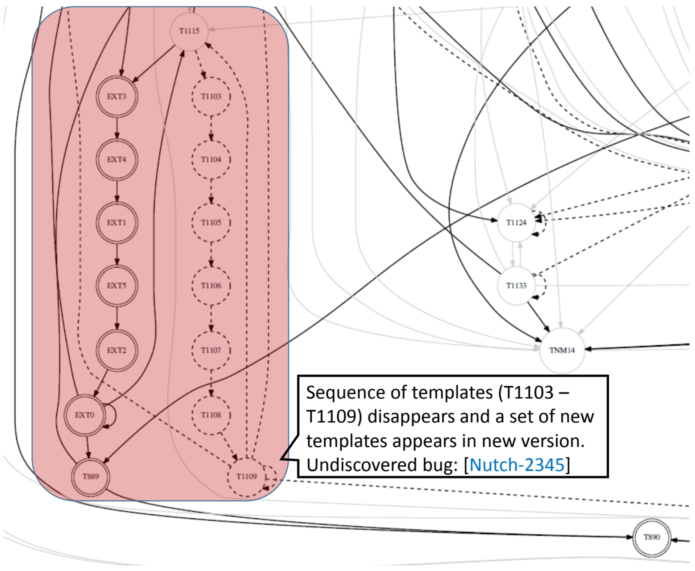

T1115
EXT3
EXT4
EXT1
EXT5
EXT2
EXT0
T889
T1103
T1104
T1105
T1106
T1107
T1108
T1109
T1124
T1133
TNM14
T890
Sequence of templates (T1103 – T1109) disappears and a set of new templates appears in new version. Undiscovered bug: [Nutch-2345]

Figure 6-1: A real motivating example capturing the differences between the execution behavior model of the two versions for an open source project, Nutch. Vertices added in the new version are double encircled; bold corresponds to edges, and dashes to deleted vertices and edges. The analysis of differences allowed us to discover a bug that we reported as NUTCH-2345. The bug was fixed by the Nutch developers.

ca. 1600 lines of code was refactored to improve the modularity. We compared the version before and after refactoring to identify differences between the two versions. We used Nutch to crawl a set of URLs, thus generating the execution logs for both the versions. We mapped the generated execution logs to templates (print statements) derived from the Nutch source code using string matching. A subset of log lines was not mapped to any source code template (i.e., from the third-party library) and clustered using a combination of approximate and weighted edit distance clustering. An execution behavior model was discovered automatically for each of the versions using the respective templatized execution logs. Each vertex in the model corresponded to a unique template. Using our automated approach, many diff regions were detected between the two discovered models.

Figure 6-1 presents one of the diff regions, that is, deletion of a set of vertices $T1103$–$T1109$ (represented as dashes) from the class *Fetcher.java* and addition of new vertices $EXT0$–$EXT5$ (double circled) from apparently the third-party library (prefixed with EXT). We manually investigated this diff region and found that the code fragment corresponding to templates $T1103$–$T1109$ was moved from *Fetcher.java* to *FetchItemQueue.java*[2]. Inspecting *FetchItemQueue.java* we found that *FetchItemQueue**s*** was used as logger instead of *FetchItemQueue*. Consequently, the log messages from *FetchItemQueue* had a wrong class name, and thus were not mapped to the corresponding source code logging statement and treated as log statements from the third-party library ($EXT0$–$EXT5$).

[2]http://svn.apache.org/viewvc?view=revision&revision=1678281

This issue was introduced in Nutch 1.11 and fixed after we reported it[3] in Nutch 1.13. Using our approach, the issue would have been detected in the version 1.11 itself. This highlights the potential of our approach for discovering anomalies by analyzing automatically identified diff regions.

6.2 Proposed Approach

The proposed approach considered execution logs and source code for the deployed and the new version as starting points. The approach leveraged execution logs without instrumenting the code because instrumentation overhead was not possible in the fast-evolving production software [185]. Nevertheless, execution paths were successfully captured from the existing logs because in practice, sufficient logging was done to facilitate runtime monitoring [186][187].

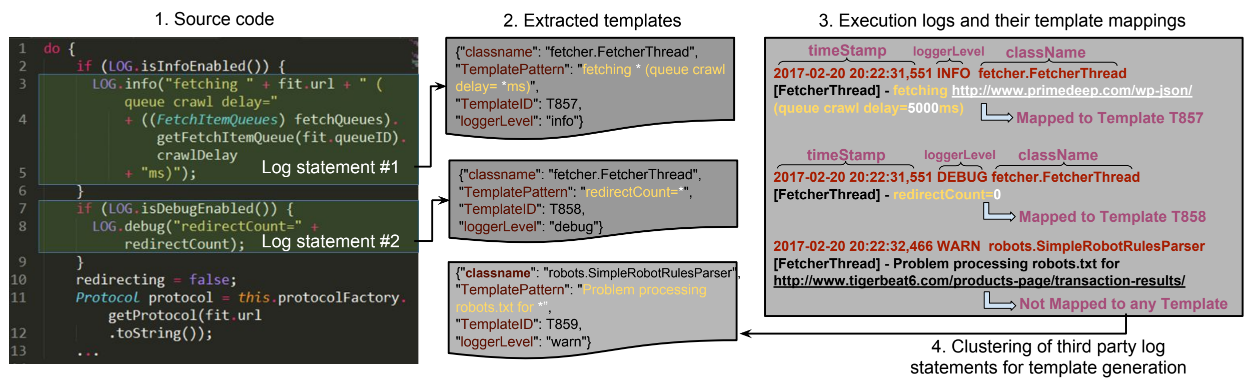
Our approach consisted three broad phases: *template mining* to map each line in the execution log to a unique template (Section 6.2.1), *execution behavior model mining* to derive execution behavior models from the templatized logs and refine the model using a multimodal approach (Section 6.2.2), and *analysis of the model differences* to identify the differences between the execution behavior models and classify them into cohesive diff regions (Section 6.2.3).

6.2.1 Template Mining

A template is an abstraction of a logging statement in the source code consisting of a fixed part and variable part (denoting parameters) [188][189]. Templates often manifest themselves as different log messages because of the presence of parameters. Thus, identifying the templates from the execution log messages has inherent challenges [168]. If no source code is available, templates can be inferred by clustering log messages [190][168]. However, often log messages from different logging statements are clustered together, resulting in inaccurate templates. As we had access to source code, we extracted templates using regular expressions (Fig. 6-2).

Derive Templates from the Source Code: In this step, the print statements were identified from the source code along with the class name and severity level (e.g., INFO, WARN, and DEBUG) [186]. We searched for the logging statements in the source code using regular expressions with some enhancements to identify ternary print statements and ignore commented logging statements in the source code. As shown in Figure 6-2, the logging statement was parsed and represented as

[3]https://issues.apache.org/jira/browse/NUTCH-2345



```
do {
    if (LOG.isInfoEnabled()) {
      LOG.info("fetching " + fit.url + " (
          queue crawl delay="
          + ((FetchItemQueues) fetchQueues).
              getFetchItemQueue(fit.queueID).
              crawlDelay
          + "ms)");
    }
    if (LOG.isDebugEnabled()) {
      LOG.debug("redirectCount=" +
          redirectCount);
    }
    redirecting = false;
    Protocol protocol = this.protocolFactory.
        getProtocol(fit.url
        .toString());
    ...
```

```
{"classname": "fetcher.FetcherThread",
"TemplatePattern": "fetching * (queue crawl delay= *ms)",
"TemplateID": T857,
"loggerLevel": "info"}

{"classname": "fetcher.FetcherThread",
"TemplatePattern": "redirectCount=*",
"TemplateID": T858,
"loggerLevel": "debug"}

{"classname": "robots.SimpleRobotRulesParser",
"TemplatePattern": "Problem processing robots.txt for *”,
"TemplateID": T859,
"loggerLevel": "warn"}
```

```
2017-02-20 20:22:31,551 INFO  fetcher.FetcherThread
[FetcherThread] - fetching http://www.primedeep.com/wp-json/
(queue crawl delay=5000ms)

2017-02-20 20:22:31,551 DEBUG fetcher.FetcherThread
[FetcherThread] - redirectCount=0

2017-02-20 20:22:32,466 WARN  robots.SimpleRobotRulesParser
[FetcherThread] - Problem processing robots.txt for
http://www.tigerbeat6.com/products-page/transaction-results/
```

Figure 6-2: The source code (1) templates were extracted (2), and log lines were mapped to them (3). Log lines from external libraries were clustered to create new templates (4).

a regular expression, which was then assigned a unique template ID. Class name and severity level were also stored as additional information to disambiguate templates having an identical invariant pattern but appearing in different classes of the code.

Although the complete source code was used to extract templates for the deployed source code version, we only analyzed the *diff* between the two source code revisions to extract the templates for the new version, as indeed, continuous deployment encourages incremental changes. Not only was the extraction more efficient, it also ensured that the unchanged templates between the two versions were represented by the same template ID. The main shortcoming of *diff* was that if a logging statement was modified, it was represented in the *diff* as a combination of addition and deletion, which was interpreted as the addition of a new template and deletion of the old template. Thus, the two execution behavior models appeared different for the templates, which were actually the same. As the modification of logging statement was frequent [186][187], we addressed this shortcoming using a novel multimodal approach for template merging and model refinement (Section 6.2.2).

Templatize Log Messages: In this step, a template ID was assigned to each log line appearing in the execution logs by matching with templates obtained from the previous step. The class name and severity level (if included as part of the log messages) were used as additional matching parameters to reduce the search space for the match (Fig. 6-2). Although regular expression matching could find the matching template, log lines matching multiple templates, templates with no fixed part, and log lines generated by the third-party libraries required special treatment. If a *log line matched with more than one template*, it was mapped to the most specific template, that is, the template with the largest fixed part. If *a class contained one logging statement without a constant part*, then all the unmapped log lines from that class with the same logging level were mapped to it[4]. Finally, *log lines from external sources* such as third-party libraries for which we had no access to the source code, could not be templatized as explained earlier. These log lines were clustered using a combination of approximate clustering [168] and weighted edit distance similarity [176]. Each cluster generated after the refinement was represented as a template and is assigned a unique template ID. Thereafter, nontemplatized log lines were matched with the templates derived from the clustering step, so that all the log lines were assigned a unique template ID.

6.2.2 Discovering Execution Behavior Model for Deployed and New Version

Execution Behavior Model (EBM) is a graphical representation of the templatized execution logs capturing the relationship between the templates. Each vertex in the model corresponded to a unique template, and the edges represented the flow relationship between the templates. As the template represented a logging statement from the code, EBM captured a subset of possible code flows.

The accuracy of identified diff regions directly depends on the accuracy of the EBM mining, which, in turn, depends on the accuracy of the template mining. As discussed in Section 6.2.1, the execution logs were templatized with high precision using the source code. However, for log lines being generated from the third-party libraries, we had to resort to the clustering-based technique with inherent limitations. This limited the template mining accuracy and, consequently, the accuracy of EBM mining. This was even more apparent in the new version because only a limited number of logs were available, hindering accurate mining [168]. Further, inconsistency in the templates because of the modified log statements in the source code being recorded as new templates led to many spurious differences between the compared models, thus making the *diff* analysis practically less effective. To overcome this problem, we proposed an iterative EBM refinement strategy using multimodal signals, that is, *text* and *vicinity* (i.e., predecessors and successors in EBM) of the template.

Iterative execution behavior model refinement: We derived execution behavior model for the deployed (EBM_d) and the new version (EBM_n) using corresponding templatized execution logs. We compared EBM_d and EBM_n to identify the vertices which are present in EBM_n but not

[4]The case with multiple such statements is very rare and hence does not affect our approach.

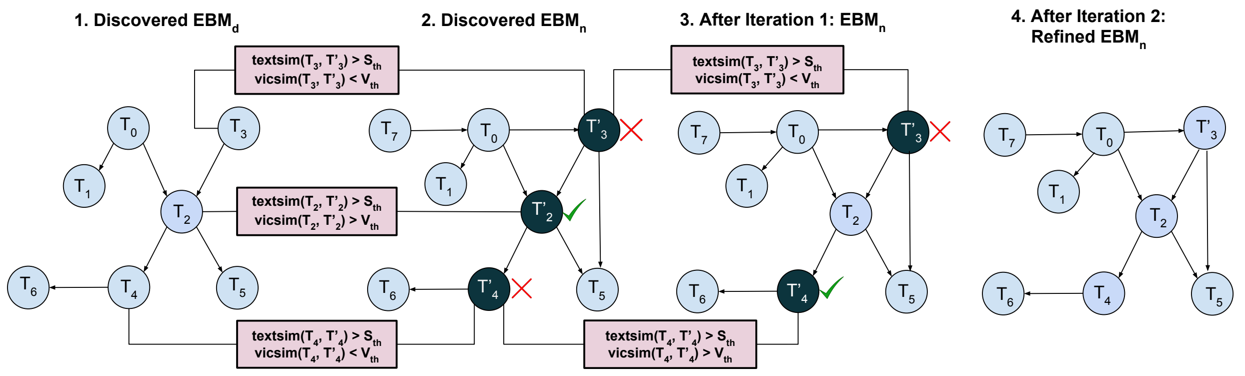

1. Discovered EBM_d
2. Discovered EBM_n
3. After Iteration 1: EBM_n
4. After Iteration 2: Refined EBM_n
textsim(T_3, T'_3) > S_th vicsim(T_3, T'_3) < V_th
textsim(T_2, T'_2) > S_th vicsim(T_2, T'_2) > V_th
textsim(T_4, T'_4) > S_th vicsim(T_4, T'_4) < V_th
textsim(T_4, T'_4) > S_th vicsim(T_4, T'_4) > V_th
T0 T1 T2 T3 T4 T5 T6 T7 T'2 T'3 T'4

Figure 6-3: Stepwise illustration of the multimodal approach for Execution Behavior Model (EBM) refinement. EBM_d is for the deployed version, and EBM_n is for the new version; textsim captures the text similarity, and vicsim captures the vicinity similarity.

in EBM_d (i.e., $\triangle T_{add}$) and vice-versa (i.e., $\triangle T_{del}$). It is possible that the vertex from the $\triangle T_{add}$ set is actually the same as the vertex from the $\triangle T_{del}$ set, but captured as a different template as discussed earlier. We identified and resolved such cases using the proposed multimodal approach, thus reducing the spurious *diff* and making the comparison more effective.

One of the multimodal signals that we used was the *textual similarity* between the templates from $\triangle T_{del}$ and $\triangle T_{add}$. If there were m templates in $\triangle T_{del}$ and n templates in $\triangle T_{add}$, then the similarity was calculated between $m \times n$ pairs. The pairs with textual similarity above a threshold were captured as *potential merge candidates*. We did not merge the templates simply based on text similarity because two textually similar templates corresponding to different logging statements in the code could exist. Hence, to improve the precision, we evaluated the similarity for one more modality, that is, *vicinity similarity*, where vicinity is the set of predecessors and successors. If the vicinity similarity was above a threshold, the templates were marked as identical. Thresholds for textual similarity and vicinity similarity could be selected based on grid search and fine-tuned to project requirements [168].

We continued the process iteratively, with each step leading to a more refined EBM_n. With every iteration, some of the vertices were marked as identical, which, in turn, could change the value of vicinity similarity for other candidate pairs. We stopped the iterations when no more candidate pairs could be merged and the EBM_n output of subsequent steps no longer changed.

Example 6.2.1. Consider the EBMs shown in Figure 6-3. By comparing EBM_d and EBM_n, we observed that $\triangle T_{del} = \{T_2, T_3, T_4\}$ and $\triangle T_{add} = \{T'_2, T'_3, T'_4\}$. We calculated *text similarity* for all the nine pairs and found the *potentially similar candidate set*, $C = \{(T_2, T'_2), (T_3, T'_3), (T_4, T'_4)\}$.

Vicinity similarity was checked for all the candidates, and in the first iteration vicinity similarity was above the threshold only for one pair, (T_2, T_2'), which was marked as identical and removed from C. In the next iteration, the remaining pairs from C were analyzed for the vicinity similarity, which was found to be greater than the threshold for (T_4, T_4'), which was again marked as identical and removed from C. Only one pair, (T_3, T_3') was not marked as same because its vicinity similarity was below threshold, even though the textual similarity was high. Consequently, *diff* set after EBM_n refinement was reduced to $\triangle T_{add} = \{T_3'\}$ and $\triangle T_{del} = \{T_3\}$.

6.2.3 Analyzing Differences between Execution Behavior Models

Amar *et al.* [191] proposed an approach for log differencing using finite-state models. They generated concise models to describe the execution and highlight the differences using two algorithms, 2KDiff and nKDiff. The focus was on identifying sequences of length k that belonged to one log (or set of logs) but not to the other. However, we aimed to identify the differences across two execution behavior models, which represents a consolidated view for many logs, and present them in a cohesive region for efficient analysis.

As EBMs were graphs, identifying the differences between them could be seen as the graph isomorphism problem [192], known to be in NP. However, as we ensured the consistency in the template ID across the two models, the comparison of the two models was simplified. The refined models were compared to identify the following differences: sets of vertices, $(\triangle diff_v)$ and edges, $(\triangle diff_e)$ that were added/deleted, as well as the set of vertices for which the relative frequency of outgoing transitions had changed ($\triangle diff_{dist}$ in EBM_n) compared with EBM_d. For efficient follow-up analysis, we grouped the identified differences into cohesive regions such that the related differences were investigated as a single unit.

Example 6.2.2. Deletion of T1103–T1109 and the corresponding edges, and addition of EXT0–EXT5 and the corresponding edges in Figure 6-1 were grouped together.

Vertex-anchored region: Intuitively, we wanted to find the maximum point from which the difference in execution behaviors was observed and the minimum point up to which there were differences in the execution behavior. It was highly likely that the differences with same maximum point were caused due to modifications in the same code, thus they should be investigated as a single unit.

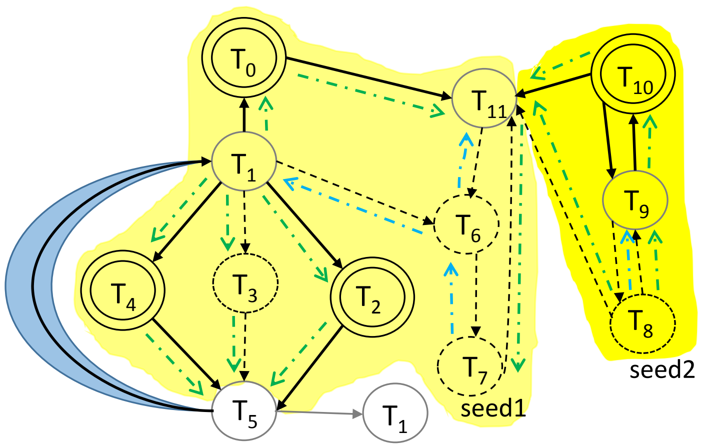


Figure 6-4: Illustration showing two vertex-anchored regions (different shades of yellow) and one edge between the unchanged vertices (blue). Blue pointers correspond to backtracking, and green pointers depict forward tracking.

A vertex, v_i was randomly selected from $\triangle \mathit{diff}_v$ as a seed to detect the region. We back-traversed the graph till an unchanged ancestor (i.e., vertex common between the two models) was detected along all the paths to v_i. All the vertices and edges along the path (including unchanged ancestor) were marked as part of the region. For all marked vertices, all the outgoing branches were traversed and marked till an unchanged child vertex (i.e., vertex common between the two models) was detected. The unchanged child vertex was not included in the region because the boundary of the region was defined till the last difference in the included path. Effectively, a region covering a set of vertices and edges was identified. The process was repeated as long as there remained unmarked vertices in $\triangle \mathit{diff}_v$. At the end of this step, all vertices from $\triangle \mathit{diff}_v$ and *some* edges from $\triangle \mathit{diff}_e$ were marked as a part of the same region. We called these regions as *vertex anchored regions*.

Example 6.2.3. Consider Figure 6-4 where $\triangle \mathit{diff}_v = \{T_0, T_2, T_3, T_4, T_6, T_7, T_8, T_{10}\}$ and $\triangle \mathit{diff}_e = \{(T_0, T_{11}), (T_1, T_0), (T_1, T_4), (T_1, T_3), (T_1, T_2), (T_1, T_6), (T_4, T_5), (T_3, T_5), (T_2, T_5), (T_{11}, T_6), (T_6, T_7), (T_7, T_{11}), (T_{10}, T_{11}), (T_{10}, T_9), (T_9, T_{10}), (T_9, T_8), (T_8, T_9), (T_8, T_{11}), (T_5, T_1)\}$. We chose T_7 as the first seed and back-traversed its incoming path (blue pointers) up to the maximum unchanged vertices, that is, $\{T_1,\ T_{11}\}$, marking vertices $\{T_7, T_6, T_1, T_{11}\}$ and edges $\{(T_7, T_6), (T_6, T_1), (T_6, T_{11})\}$. Next, the outgoing branches were traversed (green pointers) till unchanged vertex was detected and the corresponding vertices were marked. As a result, the light-yellow region was created consisting of $V_{r1} = \{T_0, T_2, T_3, T_4, T_6, T_7\}$ and $E_{r1} = \{(T_0, T_{11}), (T_1, T_0), (T_1, T_4), (T_1, T_3), (T_1, T_2), (T_1, T_6), (T_4, T_5), (T_3, T_5), (T_2, T_5), (T_{11}, T_6), (T_6, T_7), (T_7, T_{11}), (T_5, T_1)\}$ from diff_v and diff_e, respectively. For the next iteration, we chose T_8 as a seed from the set of uncovered vertices in diff_v and repeated the process to identify another region. The second region became $V_{r2} = \{T_8, T_9, T_{10}\}$ and $E_{r2} = \{(T_{10}, T_{11}), (T_{10}, T_9), (T_9, T_{10}), (T_9, T_8), (T_8, T_9), (T_8, T_{11})\}$. Hence, all the vertices from diff_v and a subset of diff_e were grouped in one of the cohesive regions.

Edge-anchored region: Not all edges from $\triangle \mathit{diff}_e$ belonged to one of the vertex-anchored regions. These were mainly the edges added/deleted between unchanged vertices and should be analyzed separately. We referred to each of these edges along with its vertices as an *edge-anchored region*.

Example 6.2.4. After detecting the vertex-anchored regions in Figure 6-4, only one edge in $\triangle \mathit{diff}_e$ was unmarked. The only edge-anchored region was hence $V_{r3} = \{T_1, T_5\}$ and $E_{r3} = \{(T_5, T_1)\}$.

Distribution-anchored region: Apart from the aforementioned two cases of structural changes (addition or deletion of vertex or edge) in EBM, we investigated the vertices common in both the versions of the model to detect the deviations in changes in the relative frequency of outgoing transitions. To capture the distribution change, for a given vertex v and its outgoing transitions common between the two models, we computed $\frac{|f_d(i) - f_n(i)|}{f_d(i)}$, where $f_d(i)$ ($f_n(i)$) is a relative frequency of transition i in EBM_d (EBM_n) among the outgoing transitions of v common between the two models. If the metric value was above the threshold for at least one transition from the vertex v, it was marked as the *distribution-anchored region*. Threshold needed to be decided manually based on the project requirements such that minor changes were discounted (i.e., not considered as part of the differences) and major changes were marked in the differences.

6.3 Case Studies

We performed case studies on two different applications: (1) Nutch[5], an open-source web crawler and (2) an industrial log analytics application. We have already shown some initial results on the Nutch project in Section 6.1 and discussed the other findings. Also, all the artifacts such as execution logs, templatized logs, execution behavior model and diff files were made publicly available for the reproducibility of the results[6]. Details of the industrial application could not be divulged for confidentiality reasons. We selected these applications primarily because of the availability of the source code and historical data on bugs and the corresponding fixes, as well as

[5]http://nutch.apache.org/
[6]https://github.com/Mining-multiple-repos-data/Nutch-results

Table 6.1: Properties of the two versions of Nutch application

Attribute	Nutch	
	Ver 1	Ver 2
Classes	415	420
Total LOC	67658	67891
Logging statements in src	1098	1097
Total lines in execution log (approx)	94137	125695
Total [Info, Debug]	19K,73K	26K,98K
Total [Error,Warn]	408,178	354,604
Vertices in model	106	104
Edges in model	328	310

frequently occurring incremental changes in these applications. Execution logs for these projects were not available; therefore, we used a custom load generator to generate logs for different source code versions.

6.3.1 Open-Source Project: Nutch

Two Nutch versions were used: (1) before the commit for [NUTCH-1934], henceforth called version 1 (deployed/prod version) and (2) after the commit for [NUTCH-1934], henceforth called version 2 (new version). This commit was considered a major change as a big class *Fetcher.java* (ca. 1600 lines of code) was refactored into *six* classes. Table 6.1 presents the details of the two Nutch versions. We derived the templates from the source code for version 1, henceforth called $templates_{v1}$. To derive the templates for version 2, $templates_{v2}$, 46 templates were deleted, and 47 templates are added to $templates_{v1}$ in accordance with the code *diff* (here, *git-diff*) between the two code versions. We generated execution logs for both the versions by crawling same URLs (i.e., mimic prod). This means that here the custom load was a manually created list of URLs (around 1000), which was passed as input. We observed that the number of log lines generated for version 1 was less than that for version 2 (cf. Table 6.1). The execution logs for version 1 and version 2 were templatized using $templates_{v1}$ and $templates_{v2}$, respectively. Around 12% of log lines were not templatized, and hence were clustered. Therefore, 80 clusters were obtained. The clusters were further refined and grouped using weighted edit distance, reducing their number to 26. Nontemplatized log lines were matched with the templates generated after clustering and every line in the execution log was templatized for both the versions. We discovered the execution behavior model (EBM) for both the versions, EBM_1 and EBM_2, and refined them using a multimodal approach.

The behavior model showed that there were 53 added and 47 deleted vertices in EBM_2 as compared with EBM_1. For every pair of the added and deleted vertices, text similarity was calculated from the source code. The *text similarity* was found to be above a threshold (here, 0.8) for 36 out of 2491 pairs, and the corresponding vicinity was compared in EBM_1 and EBM_2. *Vicinity similarity* was also found to be above a threshold (i.e., 0.5) for all the 36 candidate pairs. Thus, these vertices were marked to be the same templates across the two EBMs. For better understanding, diff refinements file was made publicly available at the *link*[8]. As a result, all the templates which are captured as a new template because of refactoring got mapped to the corresponding old templates, reducing the number of differences significantly. Refined EBM_2 was compared with EBM_1 to identify and analyze the differences. The final refined model with diffs was made publicly available[6] and included in the Appendix C.

We observed several differences that were grouped as cohesive regions using the approach discussed in Section 6.2.3.

We identified one region, which is explained in Section 6.1. In the additional region, we observed (1) deletion of vertex corresponding to template "Using queue mode : byHost" (though present in source code of both the versions) and (2) significant change in the distribution of a vertex $T1135$, such that the edge $T1135 \rightarrow T1131$ traversed only twice in EBM_1 but 601 times in EBM_2. Both observations were related to *FetcherThread.java*, which was investigated manually, and a bug was identified in the way URLs are redirected. Instead of following the correct redirect link, the code followed the same link over and over again. After the maximum number of retries exceeded, further processing of the URL stopped with the message $T1131$ ("- redirect count exceeded *"), thus increasing the frequency of this edge traversal. This bug has already been reported as NUTCH-2124[7] and attributed to patch commit we are analyzed. This validated the findings of our approach and highlighted its usefulness. Therefore, using our approach we not only detected differences but also provided the context to derive actionable insights.

6.3.2 An Industrial Application

The EBM generated automatically by our code is shown in Figure 6-5 with annotations. Gray denoted the part which is common in EBM_d and EBM_n, the dashed denoted the part present

[7]https://issues.apache.org/jira/browse/NUTCH-2124

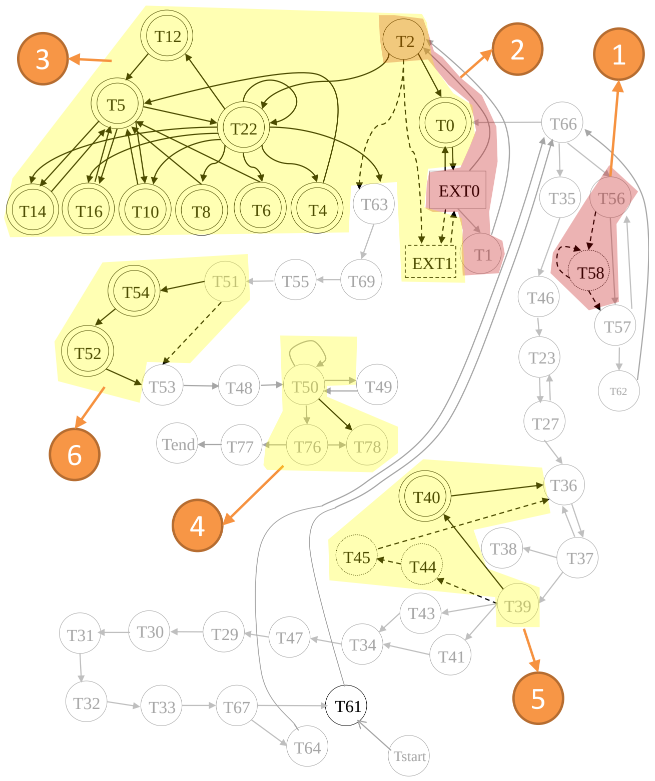


Figure 6-5: Annotated EBM highlighting the regions of *diff* for internal analytics application. Yellow regions are identified as evolutionary change and the red ones correspond to an anomaly. The gray part is common in EBM_d and EBM_n, the dashed part is present only in EBM_d but not in EBM_n, and the bold edges/double encircled parts are present in EBM_n but not in EBM_d

only in EBM_d but not in EBM_n, and the bold edges/double encircled corresponded to the part present in EBM_n but not in EBM_d. We selected two code revisions (referred to as $v1$ and $v2$) of the project such that it captured different kinds of code changes possible in the software development cycle. As shown in the Figure 6-5, our approach detected six different regions of change between the two revisions, which we explain as follows.

Region 1: The $T58$ template was present in both the source code versions but was not observed in EBM_n. Manual inspection of the code and commit history revealed it to be actually a bug caused due to faulty regular expression match and hence one conditional statement was skipped.

Region 2: A shift in distribution between edges $(EXT0, T1)$ and $(EXT0, T2)$, that is, increase in transition to $T1$ by a factor of 8. Manual inspection revealed that the cause of this anomaly was a wrong Boolean condition check, which caused flipping of the distribution between two conditional statements.

Region 3: Many new nodes appeared in EBM_n because a new Java class was added (identified in a manual investigation), which got invoked in the new version, that is, this was an evolutionary design change. Addition of $T0$, however, was not exactly related to this change. It was from the class that invoked the new feature but was added in Region 3 alongside the new class because of its close proximity.

Region 4: It had only one change, namely, the addition of edge $T50 \rightarrow T78$ and an accompanying decrease in the frequency of $T76 \rightarrow T78$. Manual investigation highlighted that $T78$ corresponded to a new exception check added in the class containing $T50$. Thus, whatever was not caught at $T50$ level was caught at $T76$.

Region 5: The main change was the addition of node $T40$ and disappearance of nodes $T44$ and $T45$. Both $T44$ and $T45$ were exception nodes existing in both code revisions, while $T40$ was a new node. Manual inspection revealed that this change was actually a result of bug fix, that is, for `ArrayOutOfBounds` exception. This validated that the bug fix worked as intended.

Region 6: Two new nodes appeared in EBM_n, and an investigation of the revision history revealed that a new function was added with two prints, which was invoked just after $T51$ in the code, causing an evolutionary change.
To summarize, our approach successfully detected all seven regions of the code change between the two code revisions. It coalesced two of the regions (in Region 3), but did not affect the usability of our approach as these regions were in proximity. Manual investigation of diff regions in EBM highlighted regression bugs and validated the evolutionary changes.

6.4 Threats to Validity

The performance of the approach depends on the pervasiveness of logging; hence, if logging statements are less in number, then it may not be possible to derive useful inferences. However, given that logs are the primary source for problem diagnosis, sufficient logging statements are written in the software [186].

We conducted experiments on one open-source project and one proprietary project to illustrate the effectiveness of the approach. However, both were Java-based projects using Log4j library for logging; thus, they were very similar in terms of logging practice. Although the approach does not make any project-specific assumptions, it is possible that the performance can vary for different project characteristics. The accuracy of the multimodal approach depends on the thresholds and thus can vary across projects.

The approach assumed the presence of an identifier (i.e., thread ID) to capture the trace for execution. As the thread ID was often present in the execution logs [193], it was fair to make this assumption. When the identifier for execution was not present, the execution behavior model could be mined using other techniques [168]. To keep our approach language independent and lightweight, we did not use static analysis techniques. Also, static analysis does not capture the complete reality of execution behavior that gets influenced by production configuration.

6.5 Summary

We presented an approach to efficiently highlight the differences in the execution (runtime) behavior caused due to code changes in evolving applications. We automatically discovered the runtime behavior model for the deployed and the new version by mining the execution logs. The models were compared to automatically identify the differences presented as cohesive diff regions. As we used a graphical representation, we not only identified diff regions but also the context to facilitate in-depth analysis.

Our preliminary evaluation on the open-source project Nutch and industrial log analytics application illustrated the effectiveness of the approach. Using our approach, we were able to detect multiple bugs introduced in the new version for both the applications. Following the analysis, we found that some of the detected bugs were already reported in their issue tracking system; therefore, we reported the remaining ones that were later fixed by the developers.

We mined the differences between the execution behavior models of the two versions but did not associate the differences with the change type, for example, bug or other evolutionary change (such as feature addition/deletion). This kind of classification not only helps in the quick resolution of bugs but also acts as an additional check to see whether all the release items have been properly taken care of before signing off on deployment. As part of the future work, identified diff regions can be automatically classified as anomaly, thus helping developers drill down to the root cause commit(s) using revision history.

Chapter 7

Conclusion

Software maintenance is a complex phase of software project involving multiple activities and consumes a significant portion of the total software project cost. Ticket resolution is an important part of software maintenance which is the focus of this thesis. Given its importance and the cost involved, an organization defines how ticket resolution should be carried out. It is known that the performance of an organization can be improved by improving the process. Therefore, the ticket resolution process needs to be continuously improved to make it more efficient.

A large volume of data is generated during ticket resolution, which is archived in software repositories. The data is mined to uncover interesting insights and actionable information for effective performance improvement decisions. The existing studies facilitate a variety of tasks during ticket resolution by applying various data mining techniques on software repositories; however, the focus in these studies was not end-to-end process analysis. In this thesis we analyzed ticket resolution using process mining techniques and derived insights to support efficient process improvement.

First, we identified the software process-related challenges, which can be addressed using process mining. We conducted qualitative surveys and interviews of more than 40 managers in a large global IT company. We identified 30 challenges, out of which more than 10 challenges corresponded to ticket resolution from the software maintenance phase. We attempted to address a few of the identified challenges pertaining to the software maintenance ticket resolution process.

One of the challenges identified from the survey was the need to analyze the data generated during the ticket resolution process to capture process reality and identify improvement opportunities. We proposed a framework for analyzing software repositories for ticket resolution from diverse perspectives, by applying process mining. The framework consists of three main steps: (1) data extraction from multiple repositories and integration, (2) transformation of the data to an event log, and (3) multiperspective process mining from the transformed event log. Using multiperspective process mining, we discovered a process model that captured the control flow, timing and frequency information about events. We then studied the inefficiencies, such as self-loops, back-forth, ticket reopening, timing issues, delays due to user input requests, and effort consumption. We also analyzed the degree of conformance between the designed and the runtime (discovered) process model. We conducted a series of case studies on the open-source Firefox browser and Core project, open-source Google Chromium project, and the IT support process of a large global IT company. The data on tickets was obtained from the Issue Tracking System (ITS) for the project (e.g. Bugzilla). We also used repositories for the Peer Code Review (PCR) system and Version Control System (VCS), where available. For each of the projects, a separate ticket resolution process was discovered and analyzed leading to diverse observations. For example, in Google Chrome, we observed that for around 14% cases, ticket was instantiated in ITS after patch submission in PCR or commit in VCS (ideally, for traceability reasons, a ticket's life cycle should start from ticket reporting in ITS followed by patch submission in PCR and commit in VCS). Moreover, for these tickets the number of patch revisions, and hence the resolution time, was higher. In Firefox and Core, we found that a significant percentage of tickets underwent multiple developer reassignment causing delays in resolution. Also, we identified two categories of tickets (wontfix and worksforme) that consumed the maximum ticket resolution effort. We noted that several tickets in these categories got reopened, signaling the need for improvement in identifying such tickets.

For the IT support process of the large global IT company, we found that around 57% of the tickets had user input requests in the life cycle, causing user-experienced resolution time to be almost double the measured service resolution time. We observed that user input requests were broadly of two types: real, seeking information from the user to process the ticket; and tactical, when no information was asked but the user input request was raised merely to pause the service-level clock. We proposed a machine learning-based system that preempts a user at the time of ticket submission to provide additional information that the analyst is likely to ask, thus reducing real user input requests. We also proposed a rule-based detection system to identify tactical user input requests. This system which predicted the information needs, exhibited an average accuracy of $94\% - 99\%$ across five cross-validations, while the traditional approaches, such as logistic regression and naive Bayes, exhibited an accuracy in the range of $50\% - 60\%$. The detection system identified around 15% of the total user input requests as tactical with a high precision. Together the proposed preemptive and detection systems could efficiently bring down the number of user input requests and improve the user-experienced resolution time.

Process mining uses largely structured data, such as event logs, and does not leverage the rich information from unstructured data, such as comments and emails. From the survey, we identified the need for granular process analysis to support efficient process improvement. Therefore, we explored unstructured data generated during process execution to capture the underlying process interactions to ensure effective process improvement decisions. To achieve this, we extracted topical phrases (keyphrases) from the unstructured data using an unsupervised graph-based approach. The keyphrases were then integrated into the event log, which then got reflected in the discovered process model. This provided insights that could not be obtained solely from structured data. To evaluate the usefulness of this approach, we conducted a case study on the ticket data of the large global IT company. Our approach extracted keyphrases from the comments associated with the tickets with an average accuracy of around 80% across different data sets. This enabled us to succinctly capture the additional information in the comments regarding issues influencing the ticket resolution process and often causing delays, such as extra information required, priority, and severity. This allowed the managers to make decisions, such as implement a bot to capture the information or add a mandatory field in the initial ticket template, thus reducing the delays incurred while waiting for information.

Some code changes are made to resolve tickets which can lead to an anomaly such as regression bugs. We aimed to detect whether ticket resolution caused some anomalous behavior, so as to reduce the post-release bugs, one of the important challenges identified from the survey. To achieve this, we proposed an approach to discover the execution behavior of the deployed and the new versions using execution logs (which contain outputs of all the print statements along with related information such as time, thread ID, statement number, and so on). Differences between the two models were then identified and refined, such that spurious differences due to logging statement modifications were eliminated. The differences were presented graphically as regions within the discovered behavior model. This allowed programmers to identify anomalous behavior changes that were not consistent with code changes, thereby identifying potential bugs that might have been introduced during the code change. To evaluate the proposed approach, we conducted a case study on Nutch (an open-source application), and an industrial application. We discovered the execution behavior models for the two application versions and identified the differences between them. By manually analyzing the regions, we were able to detect bugs introduced in the new versions of these applications. The bugs were reported and later fixed by the developers, thus, confirming the effectiveness of our approach.

In this thesis, we explored the potential of applying process mining using various data sources to improve various aspects of the ticket resolution process, an important part of software maintenance. We applied the proposed approaches to a series of case studies on data sets of commercial and open-source projects. Although we believe that the case studies are representative, the proposed approach should be applied to different data sets to establish generalizability. To support the reproducibility of our case studies, a large part of the data (with the data from the industrial partners being the only exception) has been made publicly available [120].

We believe that leveraging diverse data sources and applying analytics intelligently have more potential for process improvement. Information from other sources such as emails, chat logs, and screen recordings can further enhance process improvement. Such analyses usually focus on identifying the inefficiencies. However, we observed in this thesis that they also led to automation opportunities, making the process more efficient.

Appendix A

Survey Responses: Identifying Opportunities for Process Improvement

Problem statements listed in Table 2.1 are derived to represent multiple items collected from the first survey. Each statement has **task** (or **actual problem**) as the first component followed by **cause and benefit** as the second component. Original detailed items for every problem statement are presented here for better understanding - the text is literally taken from the survey responses. Statements with [M] belong to the maintenance phase; the *italicized* ones are for the ticket resolution process; and ♣ are the ones we attempted to address.

1. [M] ♣ *Identify BOTTLENECKS and inefficiencies causing delay in ticket resolution process to take remedial actions and have better estimation in future. [Chapter 3]*

   - Estimate Vs Actuals: always an issue. Even though the delivery/release of products happens on time but the development team has to slog every time. Better time estimation based on historical data and understanding of bottlencks to take remedial actions in future.
   - Team is spending lot of time with Client s on reviews and finalizing design, code, model etc. More often than not review effort is 2-3 times more what was originally estimated. Can we review the different artifacts, process followed (including setting up meetings, follow-ups etc.) and see where we are lacking?

- Where most of the effort are being spent by the team? Is it on solving bugs, or clarification of bugs with the users by having communication, reporting the bugs?

- Where most of the effort are being spent by the team? Is it on solving bugs, or clarification of bugs with the users by having communication, reporting the bugs?

- Why high time for resolution of issues.
- Though we have metrics to estimate time, they are not really helpful as the learning curve is not taken into consideration. Therefore time not met and always a risk. Identify bottlenecks more objectively because of which reality overshoots the estimation.
- If we can have visibility to the instances where how much time and elapse happening at each activity. Simplify to extent where is is generic and configurable.
- Bottleneck identification in process is manual after every release. It is ok with small team and projects, need automation for larger projects

2. [*M*] ♣ *Enable early detection and PREVENTION OF DEFECTS instead of fixing them during the later stage by understanding patterns of escaped defects. [Chapter 6]*

   - How to make sure that the whole rework thing can be minimized because of reasons like requirement change, misunderstanding, misallocation, early detection of defects etc.
   - Understand pattern of escaped issues by the team.
   - Assuming requirements are complete. Integrate testing with development. Understand the problem and if it can be handed by process so that we deliver the system with 0 defect.
   - Look at ongoing logs to prevent defects/responses rather than fixing them
   - Systematic and timely handling of issues which originate due to missing requirement or missed out scope, decisions, approval etc.
   - Understand the production defects and process of their inception, resolution
   - From Dev Complete -> QA complete, find escaped defects like normal flow or alternate flow. Then alter the process to include a new line also to take care of them
   - Can we improve code review and ensure that it is done more efficiently. How to reduce rework caused because of delayed defect detection? Identify the efficient set of process

practices to improve code review. May be redesign checklist based on learning from current defects.

- Though we have ticket tracking tools in place, we have to maintain the root causes and solutions separately. This becomes very inconsistent as the resources keep on changing. If PM can pull the log, analyse and the problem if it is recurrent, provide its stats, root cause, best solution reco, prevention.

3. Avoid putting efforts on LESS SIGNIFICANT ACTIVITIES by identifying redundant or unnecessary steps of process.

    - Capture unique cases. Do cost benefit analysis to understand from various variants which are the best ones with the right balance of resources deployed and outcome.
    - Identify time consuming but unnecessary processes etc.
    - Identify redundant activities which cause delay. Some sort of cost benefit analysis. Few activities are done just as part of process where they hardly contribute towards the overall outcome. Identify those activities from history variants and refine the process.
    - Can I put a check to find the redundant parts of process to optimize lifecycle and focus on critical activities. Attribute Scope and Quality to all the activities and find which activities can we cut with the minimal impact.
    - Models like waterfall and any other has multiple phases defined in standard process. However, can we adapt the process automatically for changing project requirements. E.g. if the project is small and similar to previous project we can say that design phase didn't add much value therefore it is not required

4. Automatic ADAPTATION OF PROCESS according to different project specifications that is, design process based on knowledge of similar successful projects instead of selecting process only on the basis of experience.

    - The base requirements keep changing. I wish to have more clear understanding of why we do what we are doing. I mean process justification to have clear understanding on correlation with the overall outcome.
    - Adherence to process is not high. Conviction that if I follow theprocess, it will help is missing.

- Models like waterfall and any other has multiple phases defined in standard process. However, can we adapt the process automatically for changing project requirements. E.g. if the project is small and similar to previous project we can say that design phase didn't add much value therefore it is not required

5. [M] *Inspect REOPENED issues to identify the root cause and recommend verification for future issues based on learning from issues reopened in the past. [Chapter 3]*

   - Tracking of defects that were reopened after it has passed verification
   - Understand the reopen pattern with the cause to solve. E.g. do we need to have more test cases or its with specific people or component. Learn it and preempt.
   - If we can know that there is set of issues which tend to be reopened so that we can make verification mandatory. Features like time, effort, type of interaction, criticality, priority can also be considered.
   - Identify the people whose cases are retested. Point anomalies and patterns to identify developers.
   - How do you manage historical data to improve process. Preempt iisues during maintenance. It is not possible to verify resolution of all the issues, may be based on history if we can preempt and say this type of issue is more likely to get reopened or cause regression defects therefore, verify that issue.
   - How to make sure that the whole rework thing can be minimized because of reasons like requirement change, misunderstanding, misallocation, early detection of defects etc.

6. [M] Need for efficient TASK ALLOCATION mechanism by considering individuals' skills, interests, and expertise as well as team compatibility for better utilization of resources.

   - Maintenance Phase: Lack of experience in support related tasks. Effective Knowledge Management mechanism. Lights on coverage.
   - Streamline token Allocation mechanism
   - Assign task more efficiently by analysing their comfort in terms of team compatibility and expertise.
   - Can we optimize testing team size and resources by some new processes.

- Assignment to right technically skilled professionals.
- Just like traveling salesman problem is shown alternative routes in case of traffic blockage, in case of a project roadblock, alternative resource allocation should be shown.
- Resource - skill set mapping for efficient task allocation spc. For large teams
- No efficient mapping of skills vs job done sometimes
- Staffing and estimates not done accurately becoz of which project lands in red
- Efficient task allocation very essential
- Design team with complimentary skills.
- Better mechanism to allocate task keeping strengths and interests also into consideration
- Currently task allocation based on Mangers awareness about the team. It is difficult if got to manage completely new team. Help to allocate task if new team to be managed.

7. Various approvals (such as managers' approval) are part of software development lifecyle (SDLC) and need better management. Design a process for seamless approvals to reduce delays.

    - A better managed approval process
    - Delay in approval for config changes

8. Mechanism for CONTINUOUS PROCESS EVOLUTION based on best practices of individuals who exercise the process. Therefore, improve process by encouraging on-the-job learnings of people.

    - Process Vs Individual: Even though the process is defined and is standard, it again depends on individual to proactively think and improve the way on-the-job learnings can be put back as feedback to existing processes.
    - Learn from knowledge or process variants of more efficient ppl and incorporate that to others by changing defined process.
    - Room for continuous improvements - mostly people try to handle the routine job without thinking of a way to come out of the monotonous activity. This is very difficult to decipher unless the resource voluntarily comes out and say about his work nature to the project manager.

- Identify most efficient variants and transfer that knowledge to other team members
- Though process brings standardization but people involved. Use data to connect dots and optimize testing continuously.
- Processes are more important with the bigger team and should be introspected more for continuous improvement.

9. [M] Improve effectiveness of CODE REVIEW PROCESS AND STANDARDIZATION by redesigning check list and updating code analyzers based on the defects reported during testing.

   - Effective code review and standardization. Create proper automated code analyzers and update check list based on the defects reported during testing and after release.
   - Code review checklist validation automation as currently ppl do it manually.
   - Can we improve code review and ensure that it is done more efficiently. How to reduce rework caused because of delayed defect detection? Identify the efficient set of process practices to improve code review. May be redesign checklist based on learning from current defects.
   - High quality code review process.
   - Mechanism to preempt use of FLOSS code if that can lead to problems. May be redesign the checklist to take care of code from open source and prevent later mess
   - Analyse review cycle variants for the cases where FLOSS code is used and learn from it to modify the process such that violation of licence issues can be preempted
   - Address the rework by coming up with a seamless mechanism for project code creation and tracking.

10. Facilitate BETTER INTEGRATION between different silos by reconstructing the process thus, reduce rework happening due to differences in understanding.

    - Process brings consistency as we know what to expect and what to handover to other person. Better Hand-off understanding. Process compliance checking is more important for team with members having variation (in terms of skills, language, culture)

- Is there a collaborating way to continuously integrate and check if it is going towards right direction. A reconstructed process to facilitate better integration between different silos.
- Process oriented projects end up throwing very good results. We need to understand the current coordination between various silos and hence improve the overall coordination pattern
- No coordination among teams (reporting, testing etc.)
- Better handshake between multiple parties like infrastructure and application also.
- Typical problem in development is to coordinate the DevOps activities. Eg. At time of initiating the project, a large no. of teams like Analysts, Architects, Testing, training, release mngmt. Etc. need to come together need to understand the interaction requirements and sign off. It would be interesting to look at the particular development teams operating environment and clearly defined and optimize the process and interactions. Aim should be two fold: 1. Define process that can be done in parallel rather than sequential, 2. Reduce the amount of interactions that are required at the project initiations, and get these started with a delay of around 4 weeks.

11. [M] Handle CHANGING TEAMS seamlessly by analyzing interaction pattern between team members and team dynamics.

    - On long run in maintenance projects, chances are there as to too many changes in the team, spread across states and hence no full or uniform disseminate of information shared by client and how to achieve that. Though we have process in place, still continuously changing team poses a challenge.
    - Maintenance Phase: Lack of experience in support related tasks. Effective Knowledge Management mechanism. Lights on coverage.

12. Design a technique to TRACE ADHERENCE WITH REQUIREMENTS and adapt process automatically with changing requirements.

    - Inability to trace requirements in an automated manner
    - Check if there are any deviations from the requirements

13. PEOPLE VS PROCESS: Identify which factor contributed to what extent towards the success and failure of project.

    - Identify exactly what contributed towards the success and failure of projects. Not make it always that team is responsible. Sometimes we think that process effect is diluted if we have high skilled people instead find best practices which compensate low skilled people.
    - It is very interesting to know people vs process. Where process adds value and where doesn't. Need a prescriptive process not descriptive

14. [M] Simplify tracking of the whole CODE REVIEW PROCESS to identify inefficiencies quickly.

    - Code review tool is useful but tracking the whole review process is difficult
    - Code review effectiveness needs improvement
    - We do initial rapid prototyping and then incremental. Better impact analysis and that too with high quality. From prev. code review if we can find out where we missed out in the impact assessment to correct next time.

15. [M] ♣*During issue resolution, detection and analysis of PING-PONG patterns due to bug tossing between developers to reduce resolution time. [Chapter 3]*

    - Many times an issue is reported as defect however it keeps tossing as developers say its not defect or belongs to some other component. All lot of efforts go waste in this. Can we improve process to reduce such cases
    - Unnecessary delays and blame game: specially when defects are reported, it often leads to delay and blame game. Understand those patterns with cause to avoid such tossing.
    - Ping Pong patterns between various teams when an issue is reported.

16. Improve PROJECT PLANNING AND ESTIMATION by complimenting it with the insights derived from event log mining of similar projects done in the past.

    - Project planning for complex projects with short durations, more resources, more stake holders and less duration is always challenging. Using PM, automate and give a strong recommendation with some inputs from our side.

- Effort estimation requires more accountability
- Reduce dependency on people. Facilitate quick analysis of tons of logs, provide reco along with the stats for better estimation

17. [*M*] ♣ *Investigate the LEAD TIME for issue resolution by analyzing issue resolution process from TIME PERSPECTIVE and thus increase timely resolutions. [Chapter 4]*
    - The support systems are clumsy to use and it is difficult to plan any task by considering the lead time for the resolution of issue. Reduce lead time in solving repeated issues for both the users and the support executives by designing an expert system (may be using machine learning).
    - Lead time in delivering to QA/Testing
18. Design of more meaningful QUALITY METRICS by understanding run time process practices to precisely identify the scope of improvement.
    - Lack of effective project management tools. Have an automated tool using PM with meaningful metrics to reflect discrete information w.r.t. project performance.
    - Not all the data caught during development process is set into the bug tracking system due to 1. difficulty in using the system, 2. tight coupling with the quality management audits. Need to design more meaningful quality metrics by properly understanding the process practices.
    - Automate the task of report generation showing scope of improvement in process with some metrics
19. [*M*] Equip novices with the KNOWLEDGE OF EXPERIENCED PRACTITIONERS by associating efficiency of adopted process with the experience of practitioners.
    - Understand Experience association with the process adopted, efficiency. What sequence of actions actually lead to resolution.
    - Understand the practices of long term associated people with the project vs those who participated for some time and then left as attrition is one of the concern. If it has affect on process practices and therefore overall quality.

- Freshers take lot more time to on-board than before from what I have observed. Can you understand what artifacts they use, how can we better facilitate self-learning.
- Maintenance Phase: Lack of experience in support related tasks. Effective Knowledge Management mechanism. Lights on coverage.

20. [M] ♣ *Facilitate in-depth understanding of point where things went wrong by deriving and understanding actual process at a MORE GRANULAR LEVEL. [Chapter 5]*

    - If we can have process at more granular level it makes easy to find where it has gone wrong.

21. Continuous check on SCHEDULE ADHERENCE is a complex task. Design an automated way to track and preempt if any deviations.

    - Schedule adherence becomes a complex task sometimes

22. [M] Relate bugs with the ACTUAL STAGE OF INCEPTION by understanding issue resolution life cycle along with other relevant attributes.

    - Bug tracking to relate with the actual area the bug was introduced
    - The Project team supporting an application having to sift through bunch of error messages/event logs in order to track the problem and possible root cause
    - Though we have ticket tracking tools in place, we have to maintain the root causes and solutions separately. This becomes very inconsistent as the resources keep on changing. If PM can pull the log, analyse and the problem if it is recurrent, provide its stats, root cause, best solution reco, prevention.

23. [M] ♣ *Uncover DEVIATIONS between the actual process followed by the team and the defined process, their cause, impact on overall outcome and identify the set of people exhibiting more deviations. [Chapter 3]*

    - Adherence to process is not high. Conviction that if I follow the process, it will help is missing.
    - No clear dashboard data that gives an overall view of the process followed/missed out etc. and suggestions for improvement

- PM needs to be done to find the adherence to process
- Based on my experience, most of the times it is process. I find it difficult to understand where the process went wrong exactly. Understand the process deviation.
- Risk mngmt. Process ignored at the very project conception and startup stage eventually leads to greater risk taking leading to deviations of the project constraints during the later stage. How can we control risk effectively to mitigate and erase their ill effects at every stage as it progresses.
- Inconsistent use of process templates and PCB values, the prediction on data hampers. This impacts estimation and projects may face risk.
- Audit Controls - Today production systems have numerous audit checks and balances that are put in place for data sampling and identifying any anomalies in the process.
- Process brings consistency as we know what to expect and what to handover to other person. Better Hand-off understanding. Process compliance checking is more important for team with members having variation (in terms of skills, language, culture)

24. [*M*] ♣ *INTEGRATE MULTIPLE STANDALONE SYSTEMS used during SDLC to solve data and process redundancy challenges, and obtain a holistic view. [Chapter 3]*

    - Some clients won't give work in standard tool based system. Instead the work will come in emails and meetings. Need for better "Integration of standalone systems"
    - From a consultants point of view, if a domain has been attended or how a project has been executed. Progress of project overall. Need for integration to have holistic view.
    - Information scattered at multiple places, not in synch. Bring in one place to capture all the info in one go.
    - Continuous integration mechanism by using a system where we have all the SDLC phase integrated with each other. Not achieved with existing TFS

25. [*M*] Analyze code review life cycle to identify developers who are not reviewing their code properly before they submit it for external review and the deviations from defined checklist. It will help take corrective actions and reduce defects during testing phase.

    - Can we check who is internally doing checklist for the sake of doing only. Find pattern with the comments which developer gets more review comments. Checklist ->

Comments -> Review -> Rework

- Is there an automated way to assess quality in terms of compliance to the checklist.

26. [*M*] Mechanism to manage and keep track of SVN check-ins process that is, activity sequence for merging and branching as it is very important and can help take informed decisions.

    - SVN tools check-ins process for merging and branching is a situation which is difficult to manage sometime

27. [*M*] ♣ *Capture the ACTUAL STATUS (reality) of project or any task by discovering run time process from event logs instead of current manual practice. [Chapter 3, and 4]*

    - Work assignment happens formally. However, the current manual practice to get status on each job is not reliable. Technically determine the correct status of project.

28. [*M*] *Trace the complete flow and understand WHICH ISSUE LEADS TO WHICH CODE CHANGE by analyzing event logs for issue resolution in combination with the code modified in VCS. [Chapter 3]*

    - Is there a way to create heat map of application. Look at issue logs to identify vulnerable areas of application by mapping those issue logs with the code modified in VCS along time dimension. Previous issue logs -> code change

29. [*M*] ♣ *Perform COMPARATIVE ANALYSIS OF TICKETS along dimensions such as component, owner (analyst), reporter, type such as performance, regression and security, final resolution such as duplicate, invalid and fixed, and turnaround time to derive useful insights for improvement.*

    - Tickets and its analysis on where, how, who and time taken
    - Improve performance/issue analysis, it saves a lot of time and will help to meet SLA's better
    - Understand the phenomena of defects raised due to change or fix in shared library especially in agile development.
    - Process mine privacy and security issues resolution process
    - Regression issues are many. What is the cause and how they can be fixed.

- Duplicate issues reporting one of the major challenge. Identify the group of ppl doing it the most and other characteristics
- Inability to trace history of defects

30. [*M*] *Identify the group of ACTIVE VS INACTIVE CONTRIBUTORS, GENERALIST VS SPECIALIST by analyzing performance of individuals participating in the process.*

    - Finding out the weakest link in the project in terms of low or even wrong performing members
    - Performance analysis by bringing skills, learning, other activities, project work together

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Reducing User Input Requests in Ticket Resolution

Activities of process model (refer to Figure B-1) along with the description:

- *INIT*: End user logs a ticket in ticketing system
- *AutoAssign*: Request is assigned to an analyst automatically
- *PendingForDMA*: Waiting for Delivery Manager (DM) approval
- *AutoAssignFailed*: Auto assignment failed so manager assigns ticket manually
- *DM Approved*: DM approves request for further progress
- *FLD*: Ticket properties such as impact, request area are updated
- *LOG*: A comment logged by user
- *ESC*: Priority is changed
- *ATTACHTDOC*: Required document is attached by the user
- *ACK*: Analyst changes the status to Acknowledge
- *Transfer*: Ticket transfer to other analyst
- *Call Back*: Return call request to end user
- *Awaiting User Inputs*: User input request by analyst

- *User IP Received*: Inputs updated by end user
- *RE*: Ticket resolved by analyst
- *Non-RE*: User marks ticket not resolved before ticket closes
- *AUI-Autoclosure*: Auto close if no inputs received within 10 business days
- *Autoclosure*: Auto close if resolution not confirmed by user within 2 days
- *Closed*: Ticket service closed
- *Reopen (RO)*: User dissatisfied with ticket closure, thus, reopens

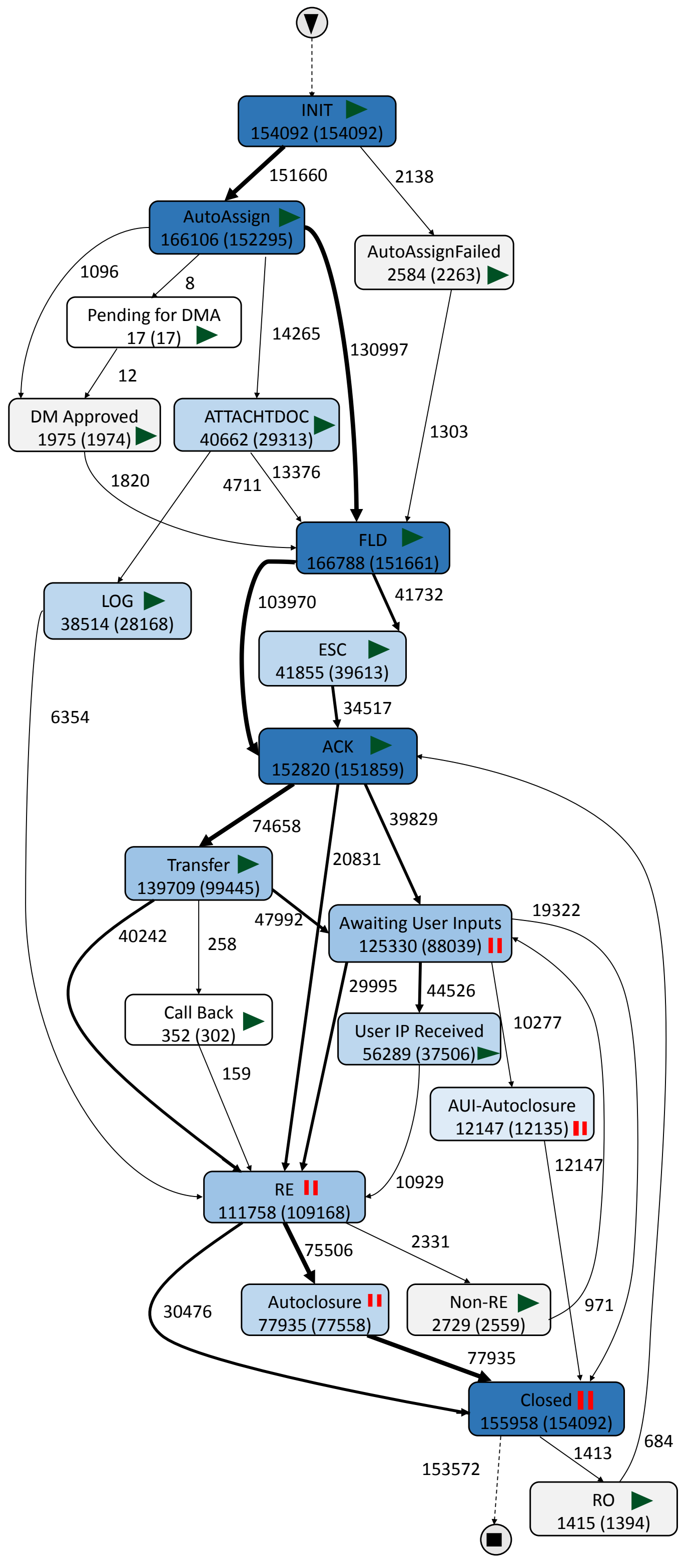


Figure B-1: Process map for IT support process of large global IT company where edges and nodes are labeled with absolute frequency. Thickness of edge and shade of node corresponds to absolute frequency. SLA clock state is indicated using pause/play icons.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix C

Identifying the Changes in Runtime Behavior of a New Release

As shown in Figure C-1, we identified two diff regions between the execution behavior model of two Nutch versions.

- **Region 1** (highlighted in blue): We observe deletion of a set of vertices $T1103 - T1109$ (represented as dashed) from class *Fetcher.java* and addition of a set of new vertices ($EXT0-EXT5$) (represented as double circled and bold edges) from apparently third party library (prefixed with EXT).

  We manually investigated this set of differences and found from the commit[1] that the code lines from class *Fetcher* corresponding to these templates are moved to another class *FetchItemQueue*. Therefore, we investigate the source code for class *FetchItemQueue* and found an issue with the logger of the class that is, used *FetchItemQueue**s*** as logger instead of *FetchItemQueue*. Consequently, the log messages from class *FetchItemQueue* had wrong classname thus, not mapped to the corresponding source code logging statement instead are treated as print from third party library. This explains that the deletion of source code vertices and addition of vertices from apparently third party library in execution behavior model is because of wrong logger being used in the newly created class after the commit. This issue was introduced in Nutch version 1.11 and was fixed in Nutch version 1.13 after we reported this in Nutch issue tracking system with issue ID, *[NUTCH-2345]*[2].

[1]http://svn.apache.org/viewvc?view=revision&revision=1678281
[2]https://issues.apache.org/jira/browse/NUTCH-2345

- **Region 2** (highlighted in green): Key observations in the region include: (i) deletion of vertex corresponding to template "Using queue mode : byHost" (that is, $T1111$, though present in source code of both the versions), (ii) addition of vertex corresponding to template "* redirect skipped: * to same url* ($T869$), and (ii) significant change in distribution of a vertex such that transition to vertex corresponding to template "- redirect count exceeded *" is increased significantly, i.e, edge from $T1135 \rightarrow T1131$ which happened only 2 times in EBM_1 has happened as much as 601 times in EBM_2 causing a significant distribution anomaly. All these observations are from class $FetcherThread.java$ which is investigated manually and a bug is identified in the way URLs are redirected. Instead of following the correct redirect link, the code was following the same link over and over again. After maximum retries is exceeded further processing of the URL stopped with the message $T1131$ (- redirect count exceeded *) thus, increased frequency. This bug is already reported as NUTCH-2124[3] and is caused due to patch committed in bug [NUTCH-1934]

Diff between the two models can sometimes be complex and difficult to analyze. Therefore, as part of future work, the identified diff regions will be automatically classified as potential anomaly. Thus, help developers drill down to the root cause for subset of diff regions.

[3]https://issues.apache.org/jira/browse/NUTCH-2124

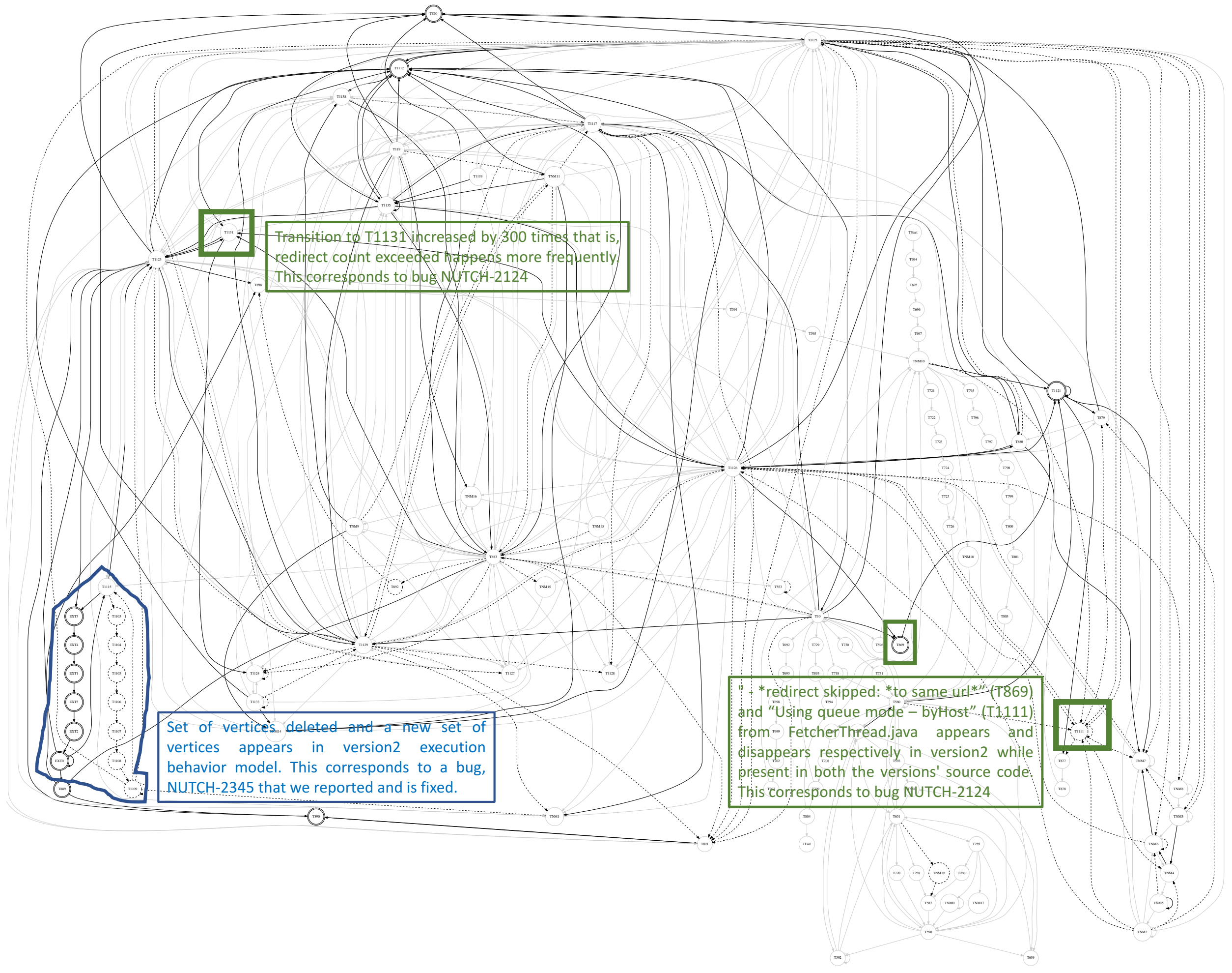

Transition to T1131 increased by 300 times that is, redirect count exceeded happens more frequently. This corresponds to bug NUTCH-2124
Set of vertices deleted and a new set of vertices appears in version2 execution behavior model. This corresponds to a bug, NUTCH-2345 that we reported and is fixed.
" - *redirect skipped: *to same url*" (T869) and "Using queue mode – byHost" (T1111) from FetcherThread.java appears and disappears respectively in version2 while present in both the versions' source code. This corresponds to bug NUTCH-2124

Figure C-1: Execution behavior model differences for Nutch, a graphical representation where dashed corresponds to deleted in new version (version2), double encircled/bold corresponds to added in new version, and grey(light) corresponds to graph common in new and deployed version (version1). Execution behavior model differences are annotated to highlight the potential bugs.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Alfonso Fuggetta. Software process: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 25–34. ACM, 2000. 1

[2] Walt Scacchi. Process models in software engineering. *Encyclopedia of software engineering*, 2001. 1

[3] CMMI Product Team. *CMMI for Services Version 1.3*. Lulu. com, 2010. 1

[4] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE software*, (1):12–21, 1996. 1

[5] Fredrik Pettersson, Martin Ivarsson, Tony Gorschek, and Peter Öhman. A practitioner's guide to light weight software process assessment and improvement planning. *Journal of Systems and Software*, 81(6):972–995, 2008. 1

[6] Silvia T Acuna, Natalia Juristo, Ana Maria Moreno, and Alicia Mon. *A Software Process Model Handbook for Incorporating People's Capabilities*. Springer Science & Business Media, 2006. 1

[7] Tony Gorschek. *Requirements Engineering supporting technical product management*. 2006. 1

[8] CMMI Product Team. Cmmi for development, version 1.2. 2006. 1

[9] Khaled El Emam, Walcelio Melo, and Jean-Normand Drouin. *SPICE: The theory and practice of software process improvement and capability determination*. IEEE Computer Society Press, 1997. 1

[10] C Iso et al. {ISO/IEC} 15504 family: Information technology-process assessment (15504-1 to 15504-5). 2006. 1

[11] Victor R Basili. Quantitative evaluation of software methodology. Technical report, DTIC Document, 1985. 1

[12] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000. 1, 3

[13] Tom Mens. Introduction and roadmap: History and challenges of software evolution. In *Software evolution*, pages 1–11. Springer, 2008. 1

[14] Bennett P Lientz and E Burton Swanson. *Software maintenance management*. Addison-Wesley Longman Publishing Co., Inc., 1980. 1

[15] Alain Abran, Pierre Bourque, Robert Dupuis, and James W Moore. *Guide to the software engineering body of knowledge-SWEBOK*. IEEE Press, 2001. 2

[16] CMMI Product Team. Capability maturity model® integration (cmmi sm), version 1.1. *CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1. 1)*, 2002. 2

[17] Alain April, Jane Huffman Hayes, Alain Abran, and Reiner Dumke. Software maintenance maturity model (SMmm): the software maintenance process model. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(3):197–223, 2005. 2

[18] Ahmed E Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 161–166. ACM, 2010. 2

[19] Tao Xie, Jian Pei, and Ahmed E Hassan. Mining software engineering data. In *Software Engineering-Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 172–173. IEEE, 2007. 2

[20] Michael W Godfrey, Ahmed E Hassan, James Herbsleb, Gail C Murphy, Martin Robillard, Prem Devanbu, Audris Mockus, Dewayne E Perry, and David Notkin. Future of mining software archives: A roundtable. *Software, IEEE*, 26(1):67–70, 2009. 2

[21] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. Data mining for software engineering. *Computer*, (8):55–62, 2009. 2, 3, 4

[22] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008. 2, 3

[23] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007. 2, 3, 4

[24] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470. ACM, 2008. 3, 4, 61, 87

[25] Anh Tuan Nguyen, Tung Thanh Nguyen, Tuan N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 70–79. IEEE, 2012. 3, 4, 61, 87

[26] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009. 3, 4, 5, 65, 86

[27] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering*, pages 361–370. ACM, 2006. 3, 4, 8, 86, 97, 105

[28] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 365–375. ACM, 2011. 3, 4, 8, 65, 86

[29] Tao Zhang, Jiachi Chen, Geunseok Yang, Byungjeong Lee, and Xiapu Luo. Towards more accurate severity prediction and fixer recommendation of software bugs. *Journal of Systems and Software*, 117:166–184, 2016. 3

[30] Meng Yan, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. A component recommender for bug reports using discriminative probability latent semantic analysis. *Information and Software Technology*, 73:37–51, 2016. 3

[31] Ashish Sureka. Learning to classify bug reports into components. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 288–303. Springer, 2012. 3

[32] Shirin Akbarinasaji, Bora Caglayan, and Ayse Bener. Predicting bug-fixing time: A replication study using an open source software project. *journal of Systems and Software*, 136: 173–186, 2018. 3

[33] Mayy Habayeb, Syed Shariyar Murtaza, Andriy Miranskyy, and Ayse Basar Bener. On the use of hidden markov model to predict the time to fix bugs. *IEEE Transactions on Software Engineering*, 44(12):1224–1244, 2018. 3

[34] Qing Mi, Jacky Keung, Yuqi Huo, and Solomon Mensah. Not all bug reopens are negative: A case study on eclipse bug reports. *Information and Software Technology*, 99:93–97, 2018. 3

[35] Thomas Zimmermann, Nachiappan Nagappan, Philip J Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *International Conference on Software Engineering (ICSE)*, pages 1074–1083. IEEE, 2012. 3, 7, 50, 64

[36] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, pages 1–38, 2012. 3, 7, 38, 50, 51, 64, 86

[37] Xin Xia, David Lo, Emad Shihab, and Xinyu Wang. Automated bug report field reassignment and refinement prediction. *IEEE Transactions on Reliability*, 65(3):1094–1113, 2016. 4

[38] Thomas Shippey, David Bowes, and Tracy Hall. Automatically identifying code features for software defect prediction: Using ast n-grams. *Information and Software Technology*, 106: 142–160, 2019. 4

[39] Chao Liu, Dan Yang, Xin Xia, Meng Yan, and Xiaohong Zhang. A two-phase transfer learning model for cross-project defect prediction. *Information and Software Technology*, 107:125–136, 2019. 4

[40] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010. 4

[41] Tao Zhang, He Jiang, Xiapu Luo, and Alvin TS Chan. A literature review of research in bug resolution: Tasks, challenges and future directions. *The Computer Journal*, 59(5):741–773, 2016. 5

[42] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. Summarizing software artifacts: a case study of bug reports. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 505–514. IEEE, 2010. 6

[43] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, 2014. 6

[44] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 11. ACM, 2012. 6

[45] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the âĂŸhurriedâĂŹbug report reading process to summarize bug reports. *Empirical Software Engineering*, 20(2): 516–548, 2015. 6

[46] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering*, pages 499–510. IEEE Computer Society, 2007. 6

[47] Ashish Sureka and Pankaj Jalote. Detecting duplicate bug report using character n-gram-based features. In *2010 Asia Pacific Software Engineering Conference*, pages 366–374. IEEE, 2010. 6

[48] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society, 2011. 7

[49] Anahita Alipour, Abram Hindle, and Eleni Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 183–192. IEEE, 2013. 7

[50] Karan Aggarwal, Finbarr Timbers, Tanner Rutgers, Abram Hindle, Eleni Stroulia, and Russell Greiner. Detecting duplicate bug reports with software engineering domain knowledge. *Journal of Software: Evolution and Process*, 29(3):e1821, 2017. 7

[51] Jaweria Kanwal and Onaiza Maqbool. Bug prioritization to facilitate bug report triage. *Journal of Computer Science and Technology*, 27(2):397–412, 2012. 7

[52] Yuan Tian, David Lo, and Chengnian Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *2013 IEEE International Conference on Software Maintenance*, pages 200–209. IEEE, 2013. 7

[53] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 419–429. IEEE, 2012. 7

[54] Yuan Tian, David Lo, and Chengnian Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *2012 19th Working Conference on Reverse Engineering*, pages 215–224. IEEE, 2012. 7

[55] Geunseok Yang, Tao Zhang, and Byungjeong Lee. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 97–106. IEEE, 2014. 7

[56] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Bo Zhou. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, 22(1):75–109, 2015. 7, 38, 51, 86

[57] John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10, 2011. 8

[58] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *2009 6th IEEE international working conference on mining software repositories*, pages 131–140. IEEE, 2009. 8

[59] Francisco Servant and James A Jones. Whosefault: automatic developer-to-fault assignment through fault localization. In *2012 34th International conference on software engineering (ICSE)*, pages 36–46. IEEE, 2012. 8

[60] Pamela Bhattacharya, Iulian Neamtiu, and Christian R Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10):2275–2292, 2012. 8

[61] Wenjin Wu, Wen Zhang, Ye Yang, and Qing Wang. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 389–396. IEEE, 2011. 8

[62] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. Dretom: Developer recommendation based on topic models for bug resolution. In *Proceedings of the 8th international conference on predictive models in software engineering*, pages 19–28. ACM, 2012. 8

[63] Tao Zhang, Geunseok Yang, Byungjeong Lee, and Eng Keong Lua. A novel developer ranking algorithm for automatic bug triage using topic model and developer relations. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 223–230. IEEE, 2014. 8

[64] Weiqin Zou, Yan Hu, Jifeng Xuan, and He Jiang. Towards training set reduction for bug triage. In *2011 IEEE 35th Annual Computer Software and Applications Conference*, pages 576–581. IEEE, 2011. 8

[65] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Accurate developer recommendation for bug resolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 72–81. IEEE, 2013. 8

[66] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012. 8

[67] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11): 1597–1610, 2013. 8

[68] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014. 8

[69] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013. 8

[70] Shaowei Wang, David Lo, and Julia Lawall. Compositional vector space models for improved bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 171–180. IEEE, 2014. 8

[71] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38 (1):54–72, 2011. 8

[72] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013. 8

[73] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 282–291. IEEE, 2013. 8

[74] AJMM Weijters, Wil van der Aalst, and AK Alves De Medeiros. Process mining with the heuristics miner-algorithm. *Technische Universiteit Eindhoven, Technical Report WP*, 166, 2006. 9, 14

[75] Wil van der Aalst, Hajo A. Reijers, Ton Weijters, Boudewijn F. van Dongen, Ana Karla Alves de Medeiros, Minseok Song, and Eric Verbeek. Business process mining: An industrial application. *Information Systems*, 32(5):713–732, 2007. 9

[76] Wil MP van der Aalst. Process mining - discovery, conformance and enhancement of business processes, 2011. 9, 10, 13

[77] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9): 1128–1142, 2004. 9, 13, 14

[78] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281–1294, 2008. 13

[79] Wil van der Aalst, Kees M van Hee, Arthur HM ter Hofstede, Natalia Sidorova, HMW Verbeek, Marc Voorhoeve, and Moe T Wynn. Soundness of workflow nets with reset arcs. In *Transactions on Petri Nets and Other Models of Concurrency III*, pages 50–70. Springer, 2009. 13

[80] Wil van der Aalst. Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10):639–650, 1999. 13

[81] Arthur HM Ter Hofstede, Wil van der Aalst, Michael Adams, and Nick Russell. *Modern Business Process Automation: YAWL and its support environment*. Springer Science & Business Media, 2009. 13

[82] Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. Causal nets: a modeling language tailored towards process discovery. In *CONCUR 2011–Concurrency Theory*, pages 28–42. Springer, 2011. 13

[83] Christian W Günther and Wil MP van der Aalst. Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In *Business Process Management*, pages 328–343. Springer, 2007. 14, 15, 48, 59

[84] Jonathan E Cook and Alexander L Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998. 14

[85] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. *Mining process models from workflow logs*. Springer, 1998. 14

[86] Anindya Datta. Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches. *Information Systems Research*, 9(3):275–301, 1998. 14

[87] Jochen De Weerdt, Manu De Backer, Jan Vanthienen, and Bart Baesens. A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. *Information Systems*, 37(7):654–676, 2012. 14

[88] AK Alves de Medeiros, Boudewijn F van Dongen, Wil MP van der Aalst, and AJMM Weijters. Process mining: Extending the α-algorithm to mine short loops. *Eindhoven University of Technology, Eindhoven*, 19:24, 2004. 14

[89] Lijie Wen, Wil van der Aalst, Jianmin Wang, and Jiaguang Sun. Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery*, 15(2):145–180, 2007. 14

[90] Jana Samalikova, Rob J. Kusters, Jos J. M. Trienekens, and Ton Weijters. Process mining support for Capability Maturity Model Integration-based software process assessment, in principle and in practice. *Journal of Software: Evolution and Process*, 26(7):714–728, 2014. 15

[91] Kwanghoon Pio Kim. Mining workflow processes from distributed workflow enactment event logs. *Knowledge Management & E-Learning: An International Journal (KM&EL)*, 4(4): 528–553, 2013. 15, 16

[92] Wikan Sunindyo, Thomas Moser, Dietmar Winkler, and Deepak Dhungana. Improving open source software process quality based on defect data mining. In *Software Quality. Process Automation in Software Development*, pages 84–102. Springer, 2012. 15, 16, 54

[93] Patrick Knab, Martin Pinzger, and Harald C Gall. Visual patterns in issue tracking data. In *New Modeling Concepts for TodayâĂŹs Software Processes*, pages 222–233. Springer, 2010. 15, 16

[94] Ashish Sureka, Atul Kumar, and Shrinath Gupta. Ahaan: Software process intelligence: Mining software process data for extracting actionable information. In *Proceedings of the 8th India Software Engineering Conference*, pages 198–199. ACM, 2015. 15

[95] Wouter Poncin, Alexander Serebrenik, and Mark G. J. van den Brand. Process mining software repositories. In *European Conference on Software Maintenance and Reengineering*, pages 5–14, 2011. 15, 16, 46

[96] Wouter Poncin, Alexander Serebrenik, and Mark G. J. van den Brand. Mining student capstone projects with FRASR and ProM. In *International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 87–96. ACM, 2011. 16

[97] Jinliang Song, Tiejian Luo, and Su Chen. Behavior pattern mining: Apply process mining technology to common event logs of information systems. In *Networking, Sensing and Control, 2008. ICNSC 2008. IEEE International Conference on*, pages 1800–1805. IEEE, 2008. 16

[98] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *ICSE*, pages 12–13, 2014. 23, 26, 27, 30

[99] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. Information needs for integration decisions in the release process of large-scale parallel development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1371–1380. ACM, 2012. 23

[100] Thomas Fritz and Gail C Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 175–184. ACM, 2010. 24

[101] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006. 24

[102] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34. ACM, 2006. 24

[103] Monika Gupta, Ashish Sureka, Srinivas Padmanabhuni, and Allahbaksh Mohammedali Asadullah. Identifying software process management challenges: Survey of practitioners in a large global it company. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 346–356. IEEE, 2015. 24, 29

[104] Tim Menzies, Laurie Williams, and Thomas Zimmermann. *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann, 2016. 28, 31

[105] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to Advanced Empirical Software Engineering*, volume 93. Springer, 2008. 28

[106] Monika Gupta and Ashish Sureka. Process cube for software defect resolution. In *APSEC*, volume 14, pages 262–269, 2014. 37, 78

[107] Guillermo Calderón-Ruiz and Marcos Sepúlveda. Automatic discovery of failures in business processes using process mining techniques. In *Anais do IX Simpósio Brasileiro de Sistemas de Informação*, pages 439–450. SBC, 2013. 38

[108] Wil van der Aalst, Hajo A Reijers, and Minseok Song. Discovering social networks from event logs. *Computer Supported Cooperative Work (CSCW)*, 14(6):549–593, 2005. 38, 56, 57

[109] HMW Verbeek, JCAM Buijs, BF van Dongen, and Wil MP van der Aalst. Prom 6: The process mining toolkit. *Proc. of BPM Demonstration Track*, 615:34–39, 2010. 47

[110] Christian W Günther and Anne Rozinat. Disco: Discover your processes. *BPM (Demos)*, 940:40–44, 2012. 47, 51

[111] Francisco Augusto Alcaraz Garcia. Tests to identify outliers in data series. *Pontifical Catholic University of Rio de Janeiro, Industrial Engineering Department, Rio de Janeiro, Brazil*, 2012. 50

[112] Christine A Halverson, Jason B Ellis, Catalina Danis, and Wendy A Kellogg. Designing task visualizations to support the coordination of work in software development. In *Proceedings of Computer supported cooperative work*, pages 39–48. ACM, 2006. 51

[113] Rami-Habib Eid-Sabbagh, Remco Dijkman, and Mathias Weske. Business process architecture: use and correctness. In *Business Process Management*, pages 65–81. Springer, 2012. 51

[114] Rob Addy. *Effective IT service management: to ITIL and beyond!* Springer-Verlag New York, Inc., 2007. 55, 78, 89, 91, 92

[115] Boudewijn F. van Dongen, Barbara Weber, Diogo R. Ferreira, and Jochen De Weerdt. Business process intelligence challenge, 2013. 55, 89

[116] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Not my bug! and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 395–404. ACM, 2011. 59

[117] Wil van der Aalst. Process cubes: Slicing, dicing, rolling up and drilling down event data for process mining. In *Asia Pacific Business Process Management*, pages 1–22. Springer, 2013. 78

[118] Gilad Barash, Claudio Bartolini, and Liya Wu. Measuring and improving the performance of an IT support organization in managing service incidents. In *International Workshop on Business-Driven IT Management*, pages 11–18. IEEE, 2007. 78

[119] Christian Bartsch, Marco Mevius, and Andreas Oberweis. Simulation of IT service processes with Petri-nets. In *International Conference on Service-Oriented Computing*, pages 53–65. Springer, 2008. 78

[120] Link to publicly available artifact. `https://github.com/Mining-multiple-repos-data/TicketExperimentalDataset`. 80, 81, 94, 95, 103, 107, 108, 150

[121] Chang Jae Kang, Young Sik Kang, Yeong Shin Lee, Seonkyu Noh, Hyeong Cheol Kim, Woo Cheol Lim, Juhee Kim, and Regina Hong. Process mining-based understanding and analysis of Volvo IT's incident and problem management processes. In *BPIC@ BPM*, 2013. 85

[122] Nor Shahida Mohamad Yusop, John Grundy, and Rajesh Vasa. Reporting usability defects: do reporters report what software developers need? In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 38. ACM, 2016. 89, 90

[123] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 301–310. ACM, 2010. 89, 90

[124] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *International Symposium on Foundations of Software Engineering*, pages 308–318. ACM, 2008. 89, 90

[125] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407. ACM, 2017. 90

[126] Andrew J Ko, Brad A Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 127–134. IEEE, 2006. 90

[127] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and emotion: sentiment analysis of security discussions on github. In *Proceedings of the 11th working conference on mining software repositories*, pages 348–351. ACM, 2014. 94, 127

[128] Martin P Robillard, Walid Maalej, Robert J Walker, and Thomas Zimmermann. *Recommendation systems in software engineering*. Springer, 2014. 95, 96

[129] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980. 95, 103

[130] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. *Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010. 96

[131] Sam Scott and Stan Matwin. Feature engineering for text classification. In *International Conference on Machine Learning*, volume 99, pages 379–388. Citeseer, 1999. 96

[132] Christopher M. Bishop. Pattern recognition. *Machine Learning*, 128, 2006. 96

[133] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002. 96

[134] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27, 2011. 97, 104

[135] Karim O. Elish and Mahmoud O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008. 97, 105

[136] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European Conference on Machine Learning*, pages 137–142. Springer, 1998. 97, 105

[137] Ana Rocío Cárdenas Maita, Lucas Correa Martins, Carlos Ramón López Paz, Sarajane Marques Peres, Marcelo Fantinato, and Majed Al-Mashari. Process mining through artificial neural networks and support vector machines: a systematic literature review. *Business Process Management Journal*, 21(6), 2015. 97, 105

[138] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *32nd International Conference on Software Engineering-Volume 1*, pages 45–54. ACM, 2010. 97, 105

[139] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. Developer prioritization in bug repositories. In *34th International Conference on Software Engineering*, pages 25–35. IEEE, 2012. 97, 105

[140] Jeff Anderson, Saeed Salem, and Hyunsook Do. Striving for failure: an industrial case study about test failure prediction. In *37th International Conference on Software Engineering*, volume 2, pages 49–58. IEEE, 2015. 97

[141] Frank E Harrell. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis.* Springer Science & Business Media, 2013. 97

[142] Tin Kam Ho. The random subspace method for constructing decision forests. *Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998. 97

[143] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, pages 321–332. ACM, 2016. 97

[144] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. 2003. 97, 104

[145] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. 97, 104

[146] Patricia S. Crowther and Robert J. Cox. A method for optimal division of data sets for use in neural networks. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 1–7. Springer, 2005. 97

[147] Nathalie Japkowicz. The class imbalance problem: Significance and strategies. In *International Conference on Artificial Intelligence*, 2000. 97

[148] Qing-Song Xu and Yi-Zeng Liang. Monte Carlo cross validation. *Chemometrics and Intelligent Laboratory Systems*, 56(1):1–11, 2001. 98

[149] Sotiris Kotsiantis, Dimitris Kanellopoulos, Panayiotis Pintelas, et al. Handling imbalanced datasets: A review. *GESTS International Transactions on Computer Science and Engineering*, 30(1):25–36, 2006. 98

[150] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009. 103

[151] Monika Gupta, Allahbaksh Asadullah, Srinivas Padmanabhuni, and Alexander Serebrenik. Reducing user input requests to improve it support ticket resolution process. *Empirical Software Engineering*, pages 1–40, 2017. 117

[152] Peter D Turney. Learning algorithms for keyphrase extraction. *Information retrieval*, 2(4): 303–336, 2000. 119

[153] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999. 119

[154] Yongzheng Zhang, Nur Zincir-Heywood, and Evangelos Milios. World wide web site summarization. *Web Intelligence and Agent Systems: An International Journal*, 2(1):39–53, 2004. 119

[155] Anette Hulth and Beáta B Megyesi. A study on automatically extracted keywords in text categorization. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 537–544. Association for Computational Linguistics, 2006. 119

[156] Kazi Saidul Hasan and Vincent Ng. Automatic keyphrase extraction: A survey of the state of the art. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1262–1273, 2014. 119

[157] Zhiyuan Liu, Wenyi Huang, Yabin Zheng, and Maosong Sun. Automatic keyphrase extraction via topic decomposition. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 366–376. Association for Computational Linguistics, 2010. 119

[158] Khaled M Hammouda, Diego N Matute, and Mohamed S Kamel. Corephrase: Keyphrase extraction for document clustering. In *International Workshop on MLDM*, pages 265–274. Springer, 2005. 119, 120, 121

[159] K. M. Hammouda and M. S. Kamel. Efficient phrase-based document indexing for web document clustering. *IEEE TKDE*, 16(10):1279–1296, Oct 2004. ISSN 1041-4347. doi: 10.1109/TKDE.2004.58. 120

[160] Mohammad S Sorower. A literature survey on algorithms for multi-label learning. 123

[161] Min-Ling Zhang and Kun Zhang. Multi-label learning by exploiting label dependency. In *Proceedings of the 16th ACM SIGKDD*, pages 999–1008. ACM, 2010. 123

[162] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010. 132

[163] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013. 132

[164] Alexander Tarvo, Peter F Sweeney, Nick Mitchell, VT Rajan, Matthew Arnold, and Ioana Baldini. Canaryadvisor: a statistical-based tool for canary testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 418–422. ACM, 2015. 132, 133

[165] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at Facebook and OANDA. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 21–30. IEEE, 2016. 132

[166] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015. 132

[167] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Trédan. Experience report: Log mining using natural language processing and application to anomaly detection. In *28th ISSRE 2017*, page 10p, 2017. 132

[168] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 215–224. ACM, 2016. 132, 133, 135, 137, 138, 146

[169] Anas Shatnawi, Hudhaifa Shatnawi, Mohamed Aymen Saied, Zakarea Al Shara, Houari Sahraoui, and Abdelhak Seriai. Identifying software components from object-oriented apis based on dynamic analysis. In *Program Comprehension (ICPC), 2018 IEEE/ACM 26th International Conference on*. IEEE, 2018. 132

[170] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. Component interface identification and behavioral model discovery from software execution data. In *Program Comprehension (ICPC), 2018 IEEE/ACM 26th International Conference on*. IEEE, 2018. 132

[171] Jan Martijn EM Van der Werf, Boudewijn F van Dongen, Cor AJ Hurkens, and Alexander Serebrenik. Process discovery using integer linear programming. In *International conference on applications and theory of petri nets*, pages 368–387. Springer, 2008. 132

[172] Maayan Goldstein, Danny Raz, and Itai Segall. Experience report: Log-based behavioral differencing. In *International Symposium on Software Reliability Engineering*, pages 282–293. IEEE, 2017. 132, 133

[173] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. *WASL*, 8:6–6, 2008. 132

[174] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2012. 132

[175] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277. ACM, 2011. 132

[176] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009. 132, 133, 137

[177] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 178–189. ACM, 2014. 132

[178] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. Synergizing specification miners through model fissions and fusions. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 115–125. IEEE, 2015. 132

[179] Tien-Duy B Le and David Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 106–117. ACM, 2018. 132

[180] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 37–48. ACM, 2016. 133

[181] Tien-Duy B Le, Lingfeng Bao, David Lo, Debin Gao, and Li Li. Towards mining comprehensive android sandboxes. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 51–60. IEEE, 2018. 133

[182] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 141–152. ACM, 2009. 133

[183] Murali Krishna Ramanathan, Ananth Y Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *ASE 2006. 21st IEEE/ACM International Conference on*, pages 241–252. IEEE, 2006. 133

[184] Mohammadreza Ghanavati, Artur Andrzejak, and Zhen Dong. Scalable isolation of failure-inducing changes via version comparison. In *International Symposium on Software Reliability Engineering Workshops*, pages 150–156. IEEE, 2013. 133

[185] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH computer architecture news*, volume 38, pages 143–154. ACM, 2010. 135

[186] Boyuan Chen and Zhen Ming Jack Jiang. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering*, pages 1–45, 2016. 135, 136, 145

[187] Shanshan Li, Xu Niu, Zhouyang Jia, Ji Wang, Haochen He, and Teng Wang. Logtracker: Learning log revision behaviors proactively from software evolution history. In *International Conference on Program Comprehension*. IEEE, 2018. 135, 136

[188] Jeroen Arnoldus, Mark G. J. van den Brand, Alexander Serebrenik, and Jacob Brunekreef. *Code Generation with Templates*. Atlantis Press, 2012. 135

[189] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. A search-based approach for accurate identification of log message formats. In *Program Comprehension (ICPC), 2018 IEEE/ACM 26th International Conference on*. IEEE, 2018. 135

[190] Risto Vaarandi and Mauno Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *Network and Service Management (CNSM), 2015*, pages 1–7. IEEE, 2015. 135

[191] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. Using finite-state models for log differencing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 49–59. ACM, 2018. 139

[192] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 1–6. IEEE, 2009. 139

[193] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30 (1):4, 2012. 146