applied sciences

MDPI

Article

Graph-Driven Exploration of Issue Handling Schemes in Software Projects

Bartosz Dobrzyński and Janusz Sosnowski *

Institute of Computer Science, Warsaw University of Technology, 00-665 Warsaw, Poland; dobrzynski.b@gmail.com
* Correspondence: janusz.sosnowski@pw.edu.pl

Abstract: The Issue Tracking System (ITS) repositories are rich sources of software development documentation that are useful in assessing the status and quality of software projects. An original model is proposed for tracing issue handling activities and their impact on project progress. As opposed to classical data mining of software repositories, we consider fine-grained features of issues which provide a better insight into project evolution. A thorough analysis of repository contents allows us to define useful metrics for characterizing issue handling schemes. These metrics are derived from the introduced graph model and developed original data mining algorithms targeting timing, issue flow progress and project actor activity aspects. This study is associated with issue processing states and their sequences (handling paths), leading to problem resolution. The introduced taxonomy of issue processing schemes facilitates the creation of a pertinent knowledge database and the identification of both bad (anomalies) and good practices. The proposed approach is illustrated with experimental results related to a representative set of ITS project repositories. These results enhance experts' knowledge of the project and can be used for correct decision-making actions. They reveal weak points in project development and possible directions for improvement.

Keywords: software development; exploring software repositories; issue handling patterns; data feature recognition; anomaly detection

Citation: Dobrzyński, B.; Sosnowski, J. Graph-Driven Exploration of Issue Handling Schemes in Software Projects. *Appl. Sci.* **2024**, *14*, 4723. https://doi.org/10.3390/app14114723

Academic Editors: Irina Trubitsyna and Reza Shahbazian

Received: 25 April 2024
Revised: 23 May 2024
Accepted: 27 May 2024
Published: 30 May 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software development and maintenance processes are documented in project repositories. The most interesting repositories are provided by the Issue Tracking (ITS) and Version Control (VCS) Systems, which are used by project stakeholders to manage their activities. The importance and practical usefulness of the ITS and VCS is underlined in [1–3]. Data collected in software repositories are the basis for diverse practical and research studies in the software engineering domain. Here, we can distinguish the following aspects considered in the literature on software repositories: the detection of duplicate issues [4,5], bug prediction and issue classification [6,7], bug triaging [8], and issue report textual analysis [9,10]. Most papers related to software bugs focus on a single selected problem considering coarse-grained data from repositories. Other types of issues also have a high impact on development processes and are underestimated. The issue handling schemes have a significant impact on software reliability. Numerous models have been proposed to address this problem, such as [11]. In [12], we showed the need to investigate fine-grained data in issue reports to obtain a deeper view regarding issue handling processes. Developing a holistic methodology of assessing the effectiveness of issue handling processes enriched with anomaly detection capabilities is a consequence of these studies.

The multitude and diversity of data in the ITS and VCS are challenges in data exploration targeting specific investigation problems. Issue management consists of identifying project requirements, planning development tasks, handling, and explaining appearing problems. All of these activities are documented in ITS repositories that describe diverse

issue features (attributes), processing progress, project actor activities, etc. Having examined many ITS repositories, we observed an abundance of diverse data relevant to project development and maintenance, which are not adequately exploited by project stakeholders. Moreover, our experience with developing and analyzing software projects [12–14] motivated us to formulate two research questions:

Q1—What is the scope of issue specification, information accuracy, statistical properties (static and dynamic), and their impact on handling processes?
Q2—How should we evaluate and improve issue handling processes on the basis of ITS reports?

The main contributions of our study are as follows:

- Introducing an original issue handling model: a processing activity graph targeted at diverse observation perspectives and linked with issue repository databases.
- Developing analysis schemes supported by original metrics and algorithms.
- Verifying the efficiency and usefulness of the proposed approach in experiments with real project repositories.

The issue handling model (IHM) with conceptualization entities and derived property profiles (fine-grained and aggregated) constitutes a body of represented knowledge on project development/maintenance processes. The IHM considers various issue attributes neglected in other approaches which impact the accuracy of the analysis. It enables the tracing of issue handling paths and defining quality analysis criteria related to issue processing phases or scenarios, handling times, and resolution coverage. The presented methodology and algorithms allow us to accumulate experience and facts in the knowledge database on software projects. Discussing the analysis results with project stakeholders facilitates making decisions on improving developed projects, avoiding bad practices, or transferring the identified good practices to new projects.

The organization of the remainder of this paper is as follows. Section 2 outlines the related work and addresses our research scope. Section 3 presents general features of issue tracking repositories and formulates challenges in their analysis. Section 4 introduces a generalized graphical model (IHM) of issue handling processes and issue flow metrics. This is complemented by a set of developed data processing algorithms. Section 5 provides illustrative analysis results for a representative set of software projects. The capabilities and limitations of the proposed approach are discussed in Section 6. Our concluding remarks are summarized in Section 7.

2. Literature Review and Problem Statement

Software repositories are used in numerous studies on diverse software engineering aspects such as the classification of issues, bug localization (diagnostics), task allocation, issue lifecycle tracing, and predicting specified issue or project features. Reported issues have different levels of importance specified by priority or severity attributes. They can be fixed by issue reporters, modified by other project actors, or derived from issue descriptions. In [15], a two-stage approach is proposed to predict the goal of opening an issue and its priority. This approach uses feature engineering methods and text classifiers. A survey on issue severity predictions based on issue descriptions is given in [16], confirming the usefulness of text mining (machine learning) supported by unstructured text features. Security-related issues need special attention due to their criticality. Security bug reports are predicted based on constructed knowledge graphs and finding security-related words in bug reports [17]. Issues reporting security vulnerabilities can be identified by checking for the presence of security-related keywords with appropriate filtering [18] to avoid mislabeling. Issue importance (criticality) impacts the order of issue resolution, so its correct specification is compelling.

ITS repositories comprise diverse issue types, e.g., bugs, new functionalities, performance improvements, code merging, and system reconfiguration, and thus require need appropriate handling services. In practice, reported issues may not be labelled or may be

labelled incorrectly. Hence, automatic classification is still helpful in issue management, and it uses text mining techniques applied to the issue title and description. The fundamental classification discriminates between bug and non-bug issues [6]. Issues of the bug type with a higher priority should be resolved prior to the release of the code. Some issues do not need fixing (will not fix issues [19]) due to the ability to postpone for another code version or having an acceptably negligible impact.

In complex projects involving many actors, a significant percentage of duplicate issue reports may appear. Their identification can significantly reduce redundant activities and avoid wasting developer resources. There exist a wide variety of approaches to automatically detect duplicate bug reports, such as text mining their descriptions and using similarity measures and some other statistics [5]. The precision of detecting duplicate bug reports can be improved by converting unstructured textual descriptions into structural data [4].

Numerous papers concentrate on issue classifications related to specified categories. The neural network RoBERT [20] identifies three issue categories: bug, enhancement, and question. Wu et al. [21] developed a prediction model to identify valid, invalid, and performance- and aging-related bug reports. The best results were achieved with the Support Vector Machine (SVM) classifier. In our previous paper [22] (and included references), we discussed some other issue classification schemes. The ITS facilitates characterizing issue types with default or customized labels. This can be enhanced by extracting issue information (report descriptions and comments) and applying text mining classifiers [23]. Correlating ITS and VCS repositories, bug classification can reflect the scope of code and the number of needed fixes [23]. Bug type identification is crucial in bug triage, localization, and fixing processes. Moshin and Shi [24] proposed a robust and effective classification model targeting these aspects. A unified model for bug classification and assignment problems is presented in [7]. It organizes data in a knowledge graph.

Problem localization is important in issue handling. Investigating textual similarities of bug descriptions and source code files helps in localizing bugs [25]. Here, we can also use machine learning techniques. Often, issue descriptions are not clear, so diverse comments are added to ITS reports. Depending upon the project, diverse types of comments and communication functions can be used [26]. The automatic classification of comments was presented in [10,22].

In issue handling, the problem of allocating issues to appropriate persons for resolution arises, referred to as the triaging problem. Some studies propose adopting machine learning and information retrieval techniques to identify suitable fixers for a given bug report. Xie et al. [8] consider the textual content and metadata in the bug reports (e.g., product, component) and the tossing sequence of the bug reports. The selection of the right team for the project is an important problem [27]. The number of project actors involved in a tossing sequence impacts the issue fixing time. Developer rankings are based on developers' competence, contributions, and the achieved average fixing time. A logistic regression classifier including the simple textual and categorical attributes of the bug reports ensured high precision and recall (close to 80%) in bug triaging [28]. The categorical attributes included the product, customer, site, priority, issue reporter, configuration, and project generation. Most bug triaging approaches have focused on static tossing graphs, while Wu et al. [29] considered interactions among developers.

Historical data in software repositories are useful in diverse predictions [30]. Timing features of reported bugs or their frequency (defect arrival) facilitate predicting unrevealed defects or assessing software reliability. For this purpose, various probabilistic models have been proposed [11,31]. Machine learning combined with the ensemble method was used in [31] and ensured high accuracy for 32 projects from three dataset repositories. The predictions were based on product and process metrics. Some approaches use the correlation of code complexity metrics and defect density. Software defect predictions can be performed within a project or across project scopes. Studies at the level of project versions are also possible [32], where a prediction model uses historical data derived by

mining version control and issue tracking systems. The impact of feature selection and sampling techniques on the accuracy of software fault prediction model is discussed in [33]. Software reliability growth models (SRGMs) are used to predict defect appearance during software lifecycles [11]. Defect disclosure and defect resolution models are distinguished in [34].

Many studies focus on predicting which components might be defective [35]. The prediction accuracy depends upon extracting appropriate features from the software repository. Defects may result not only from incorrect code development but also from a mismatch between software behavior and requirements. Other predictions focus on specific problems, e.g., predicting questions raised by developers in an issue report [36], predicting the software component required to resolve an issue [35], and predicting bug fixing time [37].

The considered prediction models are based on coarse-grained defect handling schemes, so they neglect many real problems and do not ensure acceptable modeling accuracy. In practice, they restrict studies to bugs and do not consider a wider range of issues, e.g., their types and diverse attributes which have a significant impact on issue handling processes. Moreover, these features may change over time. Typical asymmetric Gaussian shapes of defect resolution in the project lifecycle (e.g., described in [34]) are not consistent with real project behavior and used development technologies (e.g., Scrum). Hence, a deeper and fine-grained issue handling analysis is needed.

The presented literature survey showed diverse research studies focused on specific software engineering aspects considered separately. There is a gap in assessing the effectiveness of issue handling schemes covering all activities starting with the issue registration and leading to the final resolution. These activities are distributed over time and allocated to specific processing phases (states) which engage with agreed-upon project actors. They create various issue handling schemes (paths). In real projects, we observed a high diversity of issue handling schemes (patterns), even within a single project. They depend on issue features, project actors' capabilities, and development strategies, which can be derived from issue attributes in ITS repositories (Section 3). Combining such a wide scope of issue features was neglected in the literature. The comprehensive analysis of issue handling requires defining diverse observation perspectives and assessment metrics useful in monitoring project development. The proposed well-ordered approach to this analysis resulted in the creation of an original IHM, which embodies issue processing actions, involving actors and other attributes. This compact model is a backbone for constructing data exploration algorithms targeted at deriving qualitative and quantitative properties of issue handling. These algorithms provide the capability to trace, in a systematic way, issue processing at a fine-grained level, considering various dependencies. This is opposed to classical analysis schemes (coarse-grained) restricted to high-level issue flow checking (e.g., [34] and references therein). The IHM with the introduced set of evaluation concepts, analysis objects and metrics is a structured framework for studying the relevant domain knowledge, namely good and bad practices in project development. It supports project monitoring and refinement.

3. Issue Tracking Space

Issues are registered by reporters in ITS repositories. They are handled by project stakeholders (actors) according to the assumed schemes in the company and the used tools. Issue processing is documented in issue reports, which include diverse attributes updated throughout project's development and maintenance. Issue handling involves a sequence of processing steps, such as issue analysis, problem diagnosis, problem solution, testing, and validation. Most of these steps are performed by assigned project actors; however, some automatization may also appear here. The final step should provide a decisive resolution, e.g., completed, fixed, rejected as not a problem, or identified as being duplicated.

The processing progress is documented in the ITS repository, and it depends upon the type of the issue, developer requirements, company organization, and other aspects. Issue reports comprise diverse attributes, e.g., the issue identifier, type and priority, description

(title, problem summary), status of the performed processing activities and relevant time stamps, actors engaged in these activities, code localizations, and project development stages. They can be supplemented with comments, screenshots, relevant code snippets, used test suits, etc. All issues undergo a triaging process which involves issue understanding and checking its relevance, setting the priority, assigning the responsible maintainer, etc. Diverse models of activity flow are encountered in projects, and typically they adhere to the following sequence: (1) create an issue; (2) backlog; (3) selection for development; (4) to do; (5) in progress; (6) quality assurance; (7) verification; (8) done. The project stages of planning, development, testing, and deployment relate to activities 1–4, 5, 6–7, and 8, respectively.

The issue repository of the ITS is a set IS(P) that comprises issue specifications Ii $\subset$ IS(P) correlated with the considered project P. Each issue is specified by a set of attributes and an additional set of features F_i which can be represented in the following form:

$$I_i = \{a_{i1}, a_{i2}, \ldots a_{ij}, \ldots, a_{in(i)}; F_i\}$$

where a_{ij} is the j-th attribute value of the i-th issue and n(i) is the number of attributes for the i-th issue. The number of issue attributes and their contents (values) depend upon the ITS, the issue type, and project manager adjustments. Some attributes are obligatory (e.g., issue id, issue registration timestamp, issue type, priority, reporter id, etc.), while others can be optional (e.g., expected time of issue resolution). Attribute contents are assigned during issue processing by entitled project actors. Nevertheless, some attributes can be generated automatically by the ITS (e.g., issue id). In general, we can distinguish numerical (e.g., project version, time stamp, etc.), categorical (e.g., type, priority level, processing state, etc.), and textual attributes. Textual attributes describe relevant issue problems, typically the title, description, summary. Within additional issue features F_i, we distinguish historical aspects of issue processing, namely the sequence of processing states and the sequence of generated comments. These are usually specified in the form of lists:

$$L_c = <E_{c1}, E_{c2}, \ldots E_{cj}, \ldots, E_{cn(c)}>;$$

where c denotes the list category, E_{ci} is the i-th element of the list, and n(c) is the number of list elements. List elements depend upon the list category. For example, the state processing list comprises subsequent issue processing states with relevant entry timestamps and responsible project actors. The comment list comprises the comment registration timestamp, the comment author, and the comment text. Both lists can be combined in one history list. Having analyzed a wide scope of ITS repositories [12], we found that depending upon the project, issue attributes and their values may differ partially or significantly. Usually, within the same project, they are stable; however, some fluctuation may appear in time, e.g., due to ITS system upgrades, project organization and development improvements, project actors' fluctuations, etc. In practice, issue reporting deficiencies may also occur, e.g., wrong or lacking attribute specifications.

Details of issue reports rely on the used ITS tools (e.g., Jira, Bugzilla, Mantis, etc.), the project manager guidelines, stakeholders' responsibilities, and competence, etc. Issues registered in ITS include software bugs, requests for new functionalities or performance improvement, development tasks (e.g., code merging), unexpected problems, etc. Issue processing depends upon their types, the accuracy of their description, priorities, and other attribute specifications. This is also influenced by the capabilities of project stakeholders engaged in resolving the considered issue. Hence, in practice, we observe diverse sequences of issue processing actions (phases) which impact handling times. Usually, they relate to details associated with the considered issue. Some issues may have greater urgency than others, due to their priorities or them blocking the processing of other issues. On the other hand, low-urgency issues can be resolved as time permits. Issue description accuracy also impacts resolution time, e.g., requests to provide additional comments and mutual interactions of actors.

Our previous studies on handling software bugs in many projects [13] showed a high diversity of processing schemes (state paths) and handling times. We found the need for deeper studies on assessing issue handling processes. Hence, an appropriate framework is needed to extract useful knowledge in a systematic way. It has been enhanced with metrics adjusted to diverse investigation aspects and combined with introduced analysis algorithms. The backbone of this analysis is the introduced original issue handling model (IHM) enhanced with assessment metrics/profiles adapted to specified observation perspectives, e.g., general issue resolution effectiveness, detailed issue processing states and patterns, timing features, and project actor interactions. Additionally, it provides a graphical visualization of the issue processing flow. The IHM is integrated with the issue database which comprises characteristic data extracted from the ITS repository. Depending upon the project, the number of issue attributes is in the range of a few dozen to more than 100. However, only some of them can be useful in the analysis [12]. The ITS records individual issue features to provide project actors with information on their processing.

Issue data are acquired using appropriate REST APIs of the ITS. These APIs1 usually ensure access to a limited number of issues (e.g., 1000), so they have been extended to complete the specified number of issues (or relevant observation period) and store them in a uniform structure in a separate database (MongoDB) for each project. It is important to derive required data and store them in a structured and compact format adapted to further processing. The developed original analysis algorithms refer to the introduced IHM's graphical objects and their relations. They trace sequences (paths) of issue handling steps, issue timing, and other category features stored in the database. Here, we distinguish explicit issue attribute values (e.g., issue type, priority) and derived features: the number and categories of comments, the size of the issue description, issue looping characteristics, other specified properties, etc. This approach allows us to monitor the issue handling time, check compliance with assumed workflows, and perform diverse statistical analysis. It is supported by introduced original assessment metrics and profiles relevant to the considered observation perspectives and issue filtering capabilities that cover addressed investigation aspects. An important aspect is identifying issue processing deficiencies (bottlenecks, anomalies) and indicating possible improvements.

The developed analysis methodology is universal and consistent with project development schemes. To demonstrate its capabilities, we present and interpret many experimental results relevant to issue repository mining of real projects. Detailed results are presented for the open source projects Mongo DB and Log4j2 and the commercial project P1. MongoDB is a cross-platform, document-oriented database program (NoSQL database) and it utilizes JSON-like documents. Apache Log4j2 is a versatile, industrial-grade Java logging framework composed of an API, its implementation, and components to create logs while running the application. It is actively maintained by a team of several volunteers and supported by a large community. P1 is an e-commerce B2B system developed using Scrum technology. Its database stores the specifications and prices of diverse products and data related to clients. The provided services (transactions) cover many countries and product suppliers. These projects differ in their used development technologies and provide representative analysis results which illustrate the variety of issue handling schemes, properties, or anomalies. Nevertheless, we also include succinct comments on other analyzed projects to provide a wider scope of possible results.

Considering the broad range of analysis perspectives, an important issue is the flexible visualization and presentation of results adjusted to the analyst's (expert's) interests. Hence, the results are presented in cross-sectional files with possible filtering capabilities to facilitate deeper problem-oriented exploration and derive aggregated assessment profiles (useful in creating relevant knowledge databases).

4. Exploring and Assessing Issue Handling Processes

In assessing issue handling processes in software projects, we can deal with general or detailed features. In the first approach (coarse-grained), three metrics are important: the

issue handling time, the scope of unresolved issues, and the distribution of code changes per issue (Section 4.1). The second approach (fine-grained) focuses on tracing issue processing in subsequent phases (states) and various correlation dependencies. For this purpose, a graphical model (IHM) and relevant metrics are introduced (Section 4.2). This model allows the investigation of issue handling schemes (paths—Section 4.3) by processing issue repository data with the developed algorithms.

4.1. Issue Handling Summary

The efficiency of handling issues can be assessed according to the distribution of issue handling times. For each issue, we derive the timestamp of its registration and resolution. The issue resolution needs to be commented upon; theoretically, it should relate to a *closed* state. However, quite often, this state is skipped, so in the case of the last *resolved_x* state (x denotes the resolution method) not being succeeded by the *closed* one, we take its time stamp as the terminal handling time. Another problem relates to issues for which handling processes did not terminate in the considered repository time scope. Hence, deriving timing distributions, we obtain two numbers, namely that related only to the terminated issues (suffix A) and that considering the remaining issues as terminated at the last time stamp of the analysed repository (suffix B). Some illustrative statistics are shown in Table 1: the number of considered issues (N) and the average (AVG), maximal (MAX), and Q2 and Q3 quartiles of issue handling times. For each project, these statistics are calculated considering two cases of issue processing termination (suffix A or B). The results for the open source projects MongoDB and Log4J2 relate to all reported issue types: *Bugs*, *New Feature, Improvement*. In the case of the commercial project P1, *New Feature, User Story, and Task* issue types were considered.

Table 1. General statistics of issue handling times (in days) for three projects.

Project	N	AVG	MAX	Q2	Q3
$MongoDB_A$	11,675	45.2	358.2	28.4	52.3
$MongoDB_B$	1760	242.5	499.6	232.2	326.1
$P1_A$	2059	45.4	226.2	30.0	48.9
$P1_B$	319	52.0	80.2	48.8	59.9
$Log4J2_A$	2343	153.5	1721.3	40.7	125.7
$Log4J2_B$	764	7.5	26.2	2.8	7.0

Deeper studies can focus on specified issue types, priorities, resolution methods, etc. More interesting studies relate to the introduced issue handling model (Section 4.2). It is worth noting that the handling time for high-priority issues is significantly lower than for the others (compare Section 5.3). In time statistics, we skip the first quartile (Q1) due to its low practical significance. It relates to a small number of handled issues and in practice is 10 or more times lower than Q2.

Another perspective on issue handling in the project is provided by time plots showing the number of all registered events at a specified moment (starting from the beginning of the repository period) and the number of closed ones. Figure 1 shows the distribution of resolved, postponed, and duplicate issues for the open source project Spark. The postponed issues (not resolved) constitute about 20%. Their absolute number increases with time. Intuitively, it seems that the percentage and absolute values of unresolved issues should increase at the beginning of the project up to the maximum and then decrease, as shown in [34]. This could reflect the systematic engagement of actors in project development, gaining experience, and problem knowledge over time. This can refer to stable projects which achieve a maturity level. In practice, most projects include continuous code improvements, upgrades, and adding new functionalities which result in the appearance of new issues needing resolution. The increasing number of project users over time may also cause an increase in the number of generated issues. On the other hand, the fluctuation of the number of project stakeholders in time has an additional impact. Nevertheless,

a decreasing ratio of unresolved issues is required. This can be achieved by optimizing the number of issue handling actors and improving the efficiency of this process.

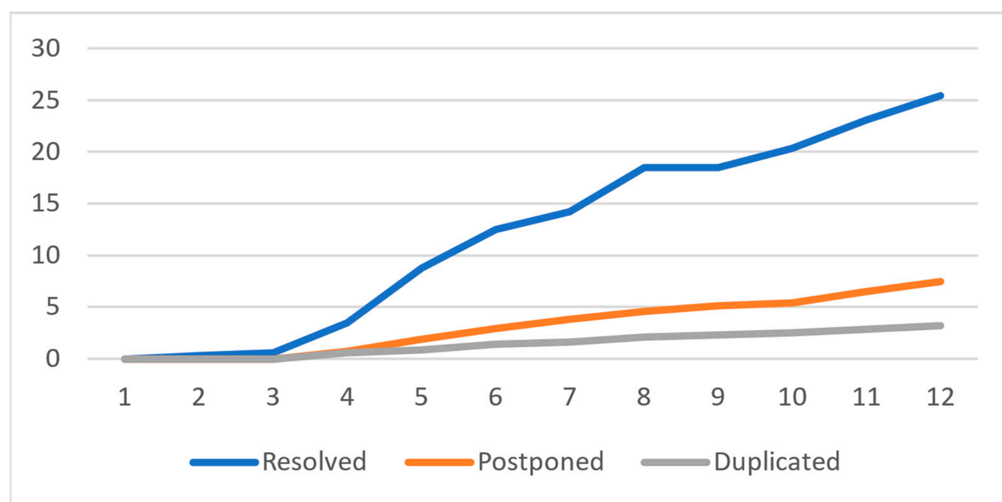

Figure 1. Distribution of processed issues for the Spark project (y-axis scaled in thousands) within the subsequent 10 years (x-axis).

The presented shape of the plots in Figure 1 was similar to many other projects; however, the rate of unresolved issues (e.g., postponed) fluctuated in the range of a few percent up to 40%, and for some projects, a low percentage was observed in the initial phase of the project and then increased or fluctuated. In the developed commercial project P1, the rate of unresolved issues fluctuated around 5%. In practice, issues have diverse impacts on project quality and usability. Hence, it is important to rapidly resolve the most critical ones. In the analysed P1 project, they constituted below 1% of the total, and absolute values reached one to five issues per year. This was simple to ensure due to the low number of high-priority issues. In the analysed open source projects Casandra, Flink, Spark and Mozilla, we observed fewer than 10, 50, 80, and 100 of unresolved issues of the two highest priorities, which constituted less than 1% of the total. Moreover, the handling times of these issues were lower than the others.

On the other hand, we can observe postponing the resolution of low-level priority (e.g., cosmetic) issues. This may cause the so-called effect of bug debt [22,38]. This leads to overlooking important problems due to the false qualification of issue priority and the combined impact of many postponed issues, which may also be triggered in correlation with code upgrades or system configuration changes. Dealing with this problem, we introduced an algorithm which detects a significant increase in such postponed issues and initiates their investigation, targeted at searching for suspicious ones to check their criticality [22]. This analysis is performed with issue description text mining based on machine learning algorithms. The output of this analysis is the set of suspicious issues needing deeper analysis. Some experiments with several projects showed that the algorithm filtered out only several percent of postponed issues as suspected, deeper analysis was restricted only to these issues, and it confirmed that 80–90% of them were critical.

Resolving issues triggers the activities of diverse project contributors (e.g., testers, analyst, developers). It is able to control their workload and efficiency. This can be traced in correlation with the IHM discussed in Section 4.2. Nevertheless, we can derive general insights regarding this problem by presenting the distribution of performed code changes which have a significant impact on testers' and developers' workflow. As an illustration, Table 2 shows code change statistics (CCS) for the MongoDB, P1 and Log4J2 projects, i.e., the percentage of relevant issues involving the specified number of performed code changes.

Table 2. Code change statistics (CCS) for MongoDB and P1 projects.

Project	1	2	3	4	5	6	7	8	9
MongoDB	48.5%	22.1%	17.7%	7.3%	2.6%	0.3%	0.3%	2.3%	0.3%
P1	84.0%	11.3%	4.1%	0%	0.3%	0%	0%	0%	0%
Log4J2	87.1%	10.9%	2.0%	0%	0%	0%	0%	0%	0%

In [14], we generated statistics of the number of changed files and code lines in relation to the number of performed commits. Higher numbers of changes or their scope may reveal the need for better issue partitioning. We observed this problem in long-term projects with fluctuating actors. To deal with this problem, we introduced parameter ACC, defined as follows:

$$ACC = \sum_{i \ni I_c} \frac{p_i}{c_i \cdot |I_c|} \quad (1)$$

where I_c denotes the set of issues (of cardinality $|I_c|$) needing code changes, p_i denotes the number of programmers involved in code changes within the i-th issue, and c_i is the number of code changes during i-th issue handling. Low values of ACC may relate to the incorrect granularity of project tasks (too complex) which result from deficiencies in issue description and problems with testing the introduced code changes. This leads to delaying issue resolution. In the considered commercial project P1, the parameter ACC was low during its initial phase (0.38–0.65). The gained development experience (as the project progressed) allowed us to improve this parameter in the range of 0.8–0.9, which contributed to decreasing issue handling times.

We should also note that issue resolution may not need code changes, e.g., unconfirmed problem, duplicated or negligible issues, etc. Moreover, sometimes changing the environment configuration is sufficient. Nevertheless, the resolution of issues in this way needs the activities of appropriate project actors (e.g., analysts, testers, etc.). In P1 Scrum projects, only 49.8% of the reported issues needed code changes/extensions, while many issues needed configuration refinement. We can also trace the number of code line or file changes over time; quite often, they decrease as the project stabilizes or becomes mature. In the P1 project, we achieved a reduction in the number of code change operations (and modified files) by 30% within 3 years, which resulted in lowering the workload of programmers. We have also introduced some metrics of project actors' activities [12]. The presented coarse-grained project development features provide a general overview. Deeper investigations need to introduce a backbone model mapping issue handling processes in well-structured schemes.

4.2. Issue Handling Model (IHM)

We introduced the Issue Handling Model (IHM), which is a significant extension of our previous studies [13]. In [13], only bugs are considered (neglecting issue attributes). Furthermore, they are traced with ad hoc scripts (not published) that are adapted to the Bugzilla repository. The IHM is a generalized model that is adjusted to deeper studies involving advanced data exploration algorithms covering a wide range of investigated issue properties and handling processes. Issue reports can refer to bug detection, correction, perfective and adaptive maintenance, adding new functionalities, the discussion of encountered problems, and other tasks. The IHM is supported by original algorithms which consider a wide range of issue types/features derived from the statistical analysis (Sections 3 and 4.1) and provide deeper insights into issue processing. It facilitates deriving categories of dominant or anomalous handling schemes, correlating them with relevant issue features, etc. The IHM covers diverse types of issues, correlated attributes, and other features. The derived issue processing features are presented in a concise form of introduced handling patterns, diverse processing profiles, effectiveness metrices, etc. This is a form of mapping the bulk of software repository data into a compact image of project development processes. It supports the assessment of these processes by project

managers and software engineering experts. The IHM is built around the original database comprising extracted features of issues from the ITS repository as well as those derived using the set of original analysis algorithms. The issue handling model (IHM) for project P is defined as

$$IHM(P) = \{V(P), E(P), \delta, \gamma, ID(P)AL\} \mid \mid R \quad (2)$$

where V(P) is the set of graph nodes describing states (phases) of issue handling and E(P) is the set of directed edges specifying state changes. The functions δ and γ map various features for nodes (states) and edges, respectively. They can relate to issue handling flow, timing, or other parameters. $ID(P)_{AL}$ is a database comprising derived issue properties using A_L algorithms linked with the issue handling graph. The range of modelling is trimmed by restrictions R imposed on the considered issues. Typically, the functions δ and γ specify issue flow, e.g.,

$$\begin{array}{c} \bigvee_{e_{ij}\epsilon E}\ \gamma_1(e_{ij}) = n_{ij} \bigvee_{v_j \epsilon V}\ \delta_1(v_j) = n_j = \sum_{e_{ij}\in in(v_j)} n_{ij} \\ \bigvee_{e_{ij}\epsilon E}\ \gamma_2(e_{ij}) = \frac{n_{ij}}{N}\ \delta_2(v_j) = \frac{n_j}{N} \\ \bigvee_{e_{ij}\epsilon E}\ \gamma_3(e_{ij}) = \frac{n_{ij}}{\sum_{e_{ij}\epsilon\ out(V_i)} n_{ij}} \end{array} \quad (3)$$

where n_{ij} is the number of issues correlated with edge e_{ij}, i.e., leaving node v_i and entering the succeeding node $v_{j,}$; v_i, $v_j \ \epsilon \ V(P)$; $e_{ij} \ \epsilon \ E(P)$; N is the number of all considered issues; and $out(v_j)$ is the set of all edges leaving the j-th node. The function γ_1 specifies the explicit number of issues (absolute throughput), while the functions γ_2 and γ_3 are relative metrics which specify the ratio of handled issues in relation to all issues N (global throughput) and issues attributed to the initial node of the considered edge (local throughput), respectively. Similarly, the functions δ_1 and δ_2 specify the absolute and relative values of issues handled in the considered state, while $in(v_j)$ is the set of all edges entering node v_j. We can also introduce external graph edges related to the direct delivery of issues by reporters to a specified state. They can be modelled by adding an external state v_{ext} or a set of such states, e.g., corresponding to specified reporters. Depending upon the analysis goal, we can assume various restrictions on considered issues:

Issue classes—issues of a specified type, priority, or severity, generated by a subset of reporters, etc.;

Issue handling times—issues reported after a fixed time moment, handled within a specified time, Scrum sprint, or program version, etc.

We can also generate reduced graphs/models by deleting specified states (e.g., related to a lower number of handled issues, beyond the interest of studies, etc.), merging/aggregating states or edges (e.g., merging diverse resolution states into one), and eliminating edges with flows below some threshold (e.g., 5%). With different forms of these functions, we can concentrate on different aspects of graph visualization (global or local perspective). These can characterize issue flow capacity, timing features, project actor activities, issue commenting patterns, issue processing patterns (path structures), etc. The IHM facilitates correlating issue report details stored in the database with structural features of the IHM. The observation granularity can be adjusted to global (aggregated) or local perspectives. The introduced IHM and the analysis methodology support finding desirable issue handling patterns and identifying anomalous behaviours. These studies can be formalized using regular expressions, defined using quality profiles, and supported by appropriate algorithms/tools.

As an illustration, we present two IHM graphs. Graph nodes represent issue handling states, and the specified numbers denote the numbers of handled issues in the states and edges. Figure 2 presents an IHM graph relevant to bug handling in the commercial project P1. Issue processing starts in the *New* state, and most issues are further moved to the *In Progress* (1315 issues) or *To be Tested* states (245), etc. The final state is *VerifiedClosed* (1595). This is a relatively complex graph with 12 nodes, and it involves some processing loops (e.g., *New*, *To be Tested*, *New*). The reduced graph that skips edges handling less than 10% of

the issues comprises seven states (*New*, *In progress*, *Review*, *Need Info*, *To be Tested*, *Reopened*, *Verified/closed*). For the issues *New Feature*, *Task*, *User Story*, the IHM graph is more complex (21 states) and reduces to nine states by eliminating edges handling less than 10% of the issues. For comparison, Figure 3 presents an IHM graph for the MongoDB project covering only bug issues with major priority and nodes covering over 20% of the issues. It shows a dominant issue handling path. The full graph is more complex, involving 20 states. Graph analysis including issue handling paths are discussed in Sections 4.3 and 5.

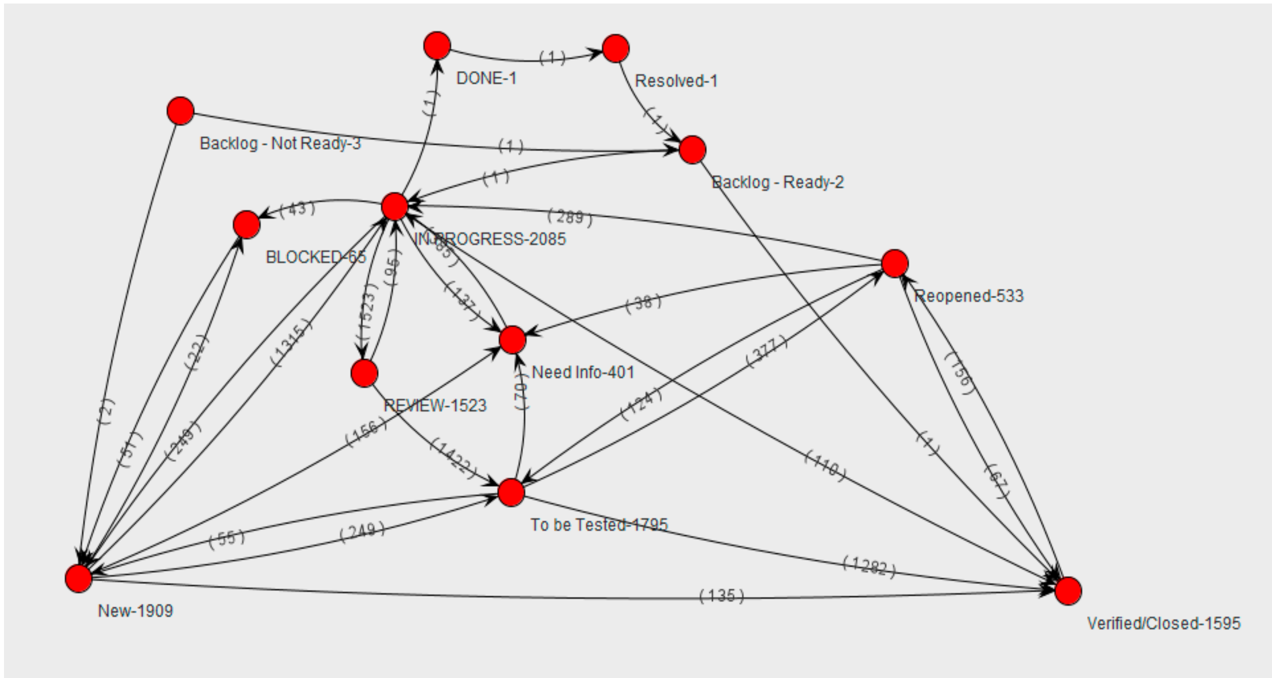

Figure 2. IHM graph for handling software bugs in the commercial project P1.

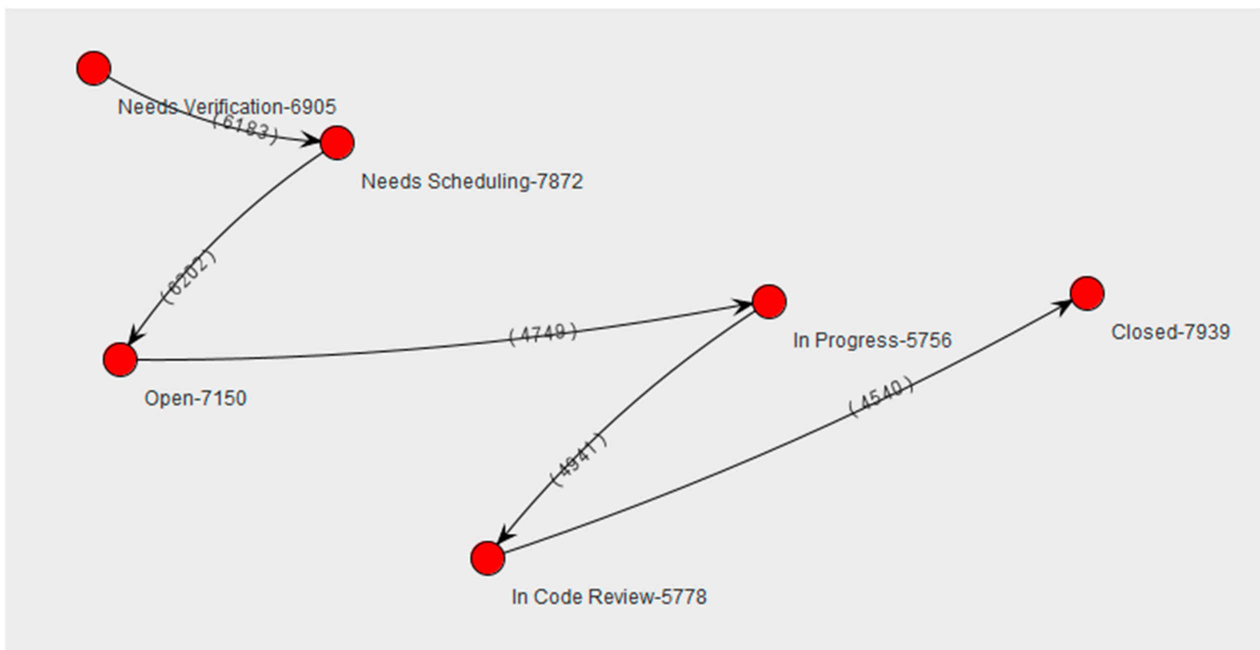

Figure 3. IHM graph of the MongoDB project covering major bug issues reduced to state nodes covering more than 20% of issues.

Algorithms for IHM graph analysis are adapted to their complexity, characterized by the following parameters: |V|—the number of nodes; |E|—the number of edges (graph size); $d(G) = |E|/(|V|*(|V| - 1))$—graph density; and $indeg(V)$ and $outdeg(V)$—the number of entering and leaving edges of the node, respectively. Having generated IHMs for many open source and commercial projects, we found that the graph complexity was relatively low: in most cases, the graph size was in the range of 25–48, $indeg(V)$ was 1–3, $outdeg(V)$ was 1–6, and graph density was 0.15–0.43. For one complex commercial project C [13] (telephone billing system), we obtained a graph comprising 26 states and 280 edges (graph density, 0.43). These values significantly reduce if we reject edges related to a low number of issues (restriction R). The IHM graph of the Mongo DB project (open source) covering issues with major priority (7488) comprises 17 nodes and 66 edges (d(G) = 0.24, indeg(V) = 0–11, outdeg(V) = 0–9)), and in case of issues with a higher priority blocker (54), it is reduced to 9 nodes and 25 edges (d(G) = 0.35, indeg(V) = 0–7, outdeg(V) = 1–5). As a consequence, algorithms tracing issue handling (e.g., finding all possible paths and loops) are simpler than classical ones.

Issues can be processed sequentially according to IHM states (nodes). The state S_j payload can be measured as the issue inflow $IF(S_j)$, i.e., the number of all issues entering state S_j:

$$IF(S_j) = \sum_{i \in I} (n_{ij}) + dp(S_j) \tag{4}$$

where n_{ij} is the number of issues passing the edge e_{ij} from state S_i to S_j, I = {i: < S_i, S_j > ∍ E}, and $dp(S_j)$ is the direct payload, i.e., the number of issues introduced directly to state S_j by reporters. In practice, this appears in the initial IHM state, while for other states, it is observed very rarely (Section 5.3). Issue inflow can also be conditioned by specified issue restriction R, as in IHM specification (2). The significance of states in issue handling can be characterized by the state load (*sl*), state weight (*sw*) and state redundancy (*sr*) profiles defined as follows:

$$\begin{aligned} sl(S_i) &= IF(S_j)/N \\ sw(S_i) &= \textstyle\sum_{P_j \in P} n(S_i, P_j)/|P| \\ sr(S_i) &= [IF(S_i) - UI(S_i)]/IF(S_i) \end{aligned} \tag{5}$$

where P_j denotes the issue handling path from the set of all possible paths P, defined as the sequence of states involved in processing issues (Section 5.2); |P| is the cardinality of set P; $n(S_i, P_j)$ denotes the number of state S_i appearances in path P_j; and $UI(S_i)$ is the number of unique issues handled in state S_i. These profiles can be adapted to issue restrictions R of the IHM graph. The *sw* profile reflects the impact of state S_i activity in all paths, while *sl* shows the ratio of issues handled in state S_i. The profile *sr* identifies issue looping ($sr(S_i) > 0$—issue processing returned to state S_i). We can also use profile vectors (over states) or aggregated state profiles defined as

$$ASx = \sum_{S_i \in S} sx(S_i) \tag{6}$$

where x specifies profile type (5), with l, w and r corresponding to state load, weight and redundancy, respectively. Deeper analysis combines detailed issue parameters from the ITS repository with IHM features using the developed algorithms described in Section 4.3.

4.3. IHM Analysis Algorithms

ITS repositories comprise a multitude of issue features and processing information that are hard to interpret directly. Complexity, non-uniform data structures, and ambiguous semantics create additional difficulties in processing software repositories. Hence, the problems of data normalization, aggregation, and problem-oriented exploration arise. Arranging issue reports in the IHM facilitates developing systematic analysis algorithms

referring to the introduced objects in the IHM, deriving issue handling states, patterns, and statistical features from diverse observation perspectives (e.g., specified by issue restriction conditions R). They can support the identification of good practices, possible threats, bad decisions, etc. (as discussed in Section 5).

Data for the IHM are downloaded into the database *ID*(*P*) from the project's ITS repository according to its API requests (Section 3) specifying issue type, priority, time stamps, lists of issue comments, etc. They are presented in JSON format. Issue handing analysis in relation to the proposed IHM is supported by specially developed algorithms grouped into three classes: preprocessing (A1), aggregation (A2), and supplementary (A3). These algorithms are outlined in points followed by detailed comments related to their specifications in the form of pseudocodes (using object-oriented dot notation, as in Python). They are used in repository analysis covering open source and commercial projects. Representative results of the performed studies targeted at diverse qualitative and quantitative issue handling aspects are presented in Section 5.

4.3.1. Issue Preprocessing Algorithm (A1)

Algorithm 1 is focused on processing single issues considering the attributes and history of reports. It derives issue handling paths (state sequences with relevant timestamps) and basic issue statistical features. Here, we can process selected issues according to specified restrictions (R), e.g., issue type, priority. For each issue I_i in the considered issue set (IS) within the repository, Algorithm 1 extracts relevant data and performs two actions:

- The creation of the issue handling path P_i complemented by the handling time $T(P_i)$, state set S_i, and relevant processing times within states ($T(S_i)$);
- The creation of a comment statistics set C_i (the number of involved actors, timing, comment size, etc.) attributed to the considered issue.

Algorithm 1 uses two auxiliary lists: a list of created issue states (CNL) and a list of state change timestamps (CTL). The list of state changes (*I.history*) is sorted by time stamp (line 10). State changes are detected by comparing the field of relevant state modification with the contents of the field "status" (lines 23–31). For each issue, the initial handling state is derived (line 12–19). The lists CNL and CTL are created in lines 21–31. The list of subsequent issue processing states, SL (with appended processing times, TS), is created in lines 33–39 and it constitutes the derived issue handling path object P.SL (line 40). The handling time of the last state in the path is assumed to be 0 (lines 34–38). Statistics on comments correlated with the considered issue are derived in lines 41–46 (word count, the number of unique words, etc.). The issue list I comprises all issue attributes, while the relevant history and comments are stored as a separate list (compatible with the Jira scheme). The history list comprises the name of the relevant issue field (attribute) which was changed, the previous value of the attribute in string format (*fromString*), the new value (*toString*), the timestamp of the performed change (*time*), and the author of the change. In the case of the state attribute assuming the value *resolved*, there is an additional substate specifying the resolution type (e.g., fixed, rejected). Hence, in lines 23–24, we check for the existence of the resolution state (*i.resolution*). It is concatenated with the resolution state (operator + in line 25). The comment list comprises the author, the time stamp of adding the comment, and the comment text. The set of created paths for all considered issues (relevant to assumed restrictions R) are submitted for further processing in Algorithm 2.

Algorithm 1. Deriving issue handling paths

input: List of JIRA issues with comments and history [I]
output: List of paths objects [P]

```
1 function translate_to_path([I]):
2     result = new empty list
3     FOR I in [I] DO
4         P = new empty Path
```

Algorithm 1. *Cont.*

```
5               SL = new empty List
6               CNL = new empty list
7               CTL = new empty list
8               CrD = I.createdDate
9               CTL.add(CrD)
10              sort I.history by timestamp
11              FS = Null
12              FOR each item in I.history DO
13                  IF (item field equals "status") THEN
14                      FS = item.getFromString()
15                      BREAK
16                  ENDIF
17              IF FS == Null THEN
18                  FS = I.currentStatusName()
19              ENDIF
20              CNL.add(firstStatus)
21              FOR each i in I.history DO
22                  SN = Null
23                  IF (i field == "status") DO
24                          IF i.resolution exists THEN
25                              SN = i.fromString + i.resolution
26                          ELSE
27                              SN = i.fromString
28                  ENDIF
29                      CNL.add(SN)
30                      CTL.add(item.getTime)
31                  ENDIF
32              ENDFOR
33              FOR i starting from 0 to len(CTL) DO
34                  TS = 0
35                  IF(i < len(CTL) − 1) THEN
36                      TS = CTL[i + 1] − CTL[index]
37                  ENDIF
38                  SL.add(new S(CNL[i], TS))
39              ENDFOR
40              P.SL = SL
41              FOR each C in I.comments DO
42                  WC = C.countWords()
43                  UWC = C.countUniqueWords
44                  P.CL add C.author, C.text
45                  P add WC and UWC to proper lists
46              ENDFOR
47              result.add(P)
48          ENDFOR
49          return result
50  endfunction
```

4.3.2. Aggregation of State and Path Features (Algorithm 2)

The issue paths derived by Algorithm 1 can be aggregated considering their unique state sequences. This is provided by Algorithm 2, which tracks state transitions and relevant timing features in all paths. It also determines the IHM graph structure, state statistics, and path-related features, such as issue coverage, structural properties/profiles, timing statistics, etc. Algorithm 2 inspects the sets P_i, $T(P_i)$, S_i, and $(T(S_i))$ for all I_i in IS (provided by Algorithm 1) and includes the following actions:

- The creation of a set of unique paths (PU) and relevant aggregated statistics (handled issues, timing features (e.g., Q1, Q2, Q3 quartiles), comment global statistics (over diverse path types), etc.);
- The creation of a global set of states (SG) and relevant statistics (handled issues, unique issues, processing timing, etc.);
- The creation of an IHM graph with nodes relevant to set SG and edges derived by tracing the PU set.

Algorithm 2 constructs state graph G by subsequently tracing all paths P generated by Algorithm 1. For each state in the path, it creates a list of unique settings and updates relevant vertices of the generated graph G (lines 6–16). For each state in the considered path, we check for its existence in the created graph G; in the positive case, we update data of the relevant graph vertex in the analyzed path. A single path Pi in P relates to a single issue. In line 11, we use the special function *getVertices*(), which when applied to graph G provides all vertices of the currently constructed graph G. Combining paths with the same state sequence (lines 19–20), we generate aggregated statistics, including issue flow (the number of covered issues), timing, and other features. In the opposite case, a new vertex is added (lines 35–39). Next, we check for the existence of an edge between the subsequent path states S(i) and S(i + 1) in the created graph G. In the positive case, it updates the issue flow for this edge (*E.count*), and in the opposite case, a new edge is created (lines 24–32). Having traced all states in the analyzed path of the considered issue, we check whether it is compatible (the same structure) with any previously analyzed paths. In the positive case, we update the statistics of the identified compatible paths. In the opposite case, we create a new unique path for the further analysis (lines 42–47). Algorithm 2 creates sets of unique paths (UP)—lines 39–44. States appearing in the path are mapped as vertices of the graph (V)—line 37. Each vertex comprises two counters: (i) *count* (line 38)—the number of setting the corresponding state by all passing it issues; and (ii) *countunique* (line 37)—the number of unique appearances of the considered state (equivalent to the number of issue paths in which it appears). The directed edges (E) of graph (G) define transitions between states (lines 21–32). Algorithm 2 is supported by the procedure *paintGraph*, which includes graph visualization (line 51). It ensures the capability of generating restricted graphs, e.g., with filtered-out states or edges corresponding to the issue flow below a specified threshold (e.g., 1%). The procedure *generate_state_features* provides the set of all identified states and relevant statistical features, including the issue count and count unique numbers, timing statistics (minimal, maximal, average and quartiles Q1, Q2, Q3), and other profile values. Similarly, the procedure *generate_path_features* provides statistics of paths, including the number of issues processed by the considered path, timing features similar to in case of states but related to passing all states in the path), and other features (e.g., related to comments—discussed farther on).

Algorithm 2. Generation of the IHM

input: List of paths ([P]) with relevant statistics.
output: Graph (G) of the IHM, files with statistics of paths and states

```
1  function make_graph([P]):
2      G = new empty graph
3      UPL = new empty List
4      FOR each P in [P] DO
5          US = new empty List
6          FOR each S in P.SL:
7              IF (S not in US) THEN
8                  US.add(S.name)
9              ENDIF
10         ENDFOR
11         VL = G.getVerticies()
12         FOR each v in VL DO
13             IF (v.name in US) THEN
```

Algorithm 2. *Cont.*

```
14                  v.UVC += 1
15              ENDIF
16          ENDFOR
17          FOR each S in P.SL DO
18              IF (G contains V with name equals to S.name) THEN
19                  V.count += 1
20                  V.TL = S.time
21                  IF (S(i+1) and G contains vertex V(i + 1) with S(i + 1).name
22                  and G contains edge E<V(i),V(i + 1)>) THEN
23                      E.count += 1
24                  ELSEIF (S(i + 1) and G contains vertex V(i + 1) with S(i + 1).name
25                  and G does not contains edge E<V(i),V(i + 1)>) THEN
26                      E = new E<V(i),V(i + 1)> and count = 1
27                      G.addEdge(E)
28                  ELSEIF (S(i+1) and G does not contain vertex V(i + 1)) THEN
29                      V = new V<S(i + 1).name> and count = 0
30                      G.addVertex(V)
31                      E = new E<V(i),V(i + 1)> with count = 1
32                      G.addEdge(E)
33                  ENDIF
34              ELSE
35                  V = create new V(S.name)
36                  V.timeList add(S.time)
37                  V.uniqueCount = 1
38                  V.count += S.count
39                  G.addVertex(V)
40              ENDIF
41          ENDFOR
42          IF (UPL contains P) THEN
43              UPL[P].update(P.statistics)
44          ELSE
45              UP = create new UP(P)
46               UPL.add(UP)
47          ENDIF
48      ENDFOR
49      generatePathsFile(UPL)
50      generateVertexFile(G)
51      paintGraph(G)
52      return G
53  endfunction
```

4.3.3. Supplementary Algorithms (A3)

The set of identified paths provided by Algorithm 2 can be quite large:, reaching several dozen or hundreds, and sometimes exceeding one thousand. Therefore, to facilitate their analysis, we must introduce some classifications related to state and other features. This is useful in identifying issue processing deficiencies and to compare good and bad practices in the investigated projects. The developed supplementary algorithms are used to derive specific path features:

- Deriving paths with loops (PL) within the PU set and relevant statistics: the number of handled issues, timing features, path structure (A3_L);
- Deriving specified path classes in the PU set and relevant statistics (A3_P);
- Deriving commenting features (A3_C).

Supplementary algorithms facilitate the identification of anomalies and the evaluation of issue handling processes. They are based on the results from Algorithms 1 and 2.

Algorithm 3 identifies state looping in paths. It provides lists of detected loops. Moreover, we distinguish single-state or multiple-state loop categories. Single-state loops comprise the repetition of the same state. Single-state loops usually result from deficiencies in reporting processes. In Algorithm 3, state loops of this category are identified within lines 6–10 of the FOR code loop. State loops of the second category are identified within lines 13–20 of the WHILE code loop. Searching for all possible state loops is ensured by the FOR code loop (lines 4–20), which iterates on the parameter LL, running from 1 to the half-length of the analyzed path (maximal possible loop length). The WHILE code loop checks for successive state subsequences in the path showing repetitions. As an illustration, Algorithm 3 applied to an excerpt of the state path [*O*,*O*,*R_F*,*C*,*RE*,*R_F*,*C*,*RE*,*R_F*,*C*] provides two loops: {[*O*:1] and [*C*;*RE*;*R_F*;*C*:4]}. The first one is a single-state loop comprising state O, while the second one is a four-state loop with the starting state C.

Algorithm 3. Identification of loops in issue handling paths (A3_L)

```
input: Path (P) represented as a list of statuses for given issue,
output: List of found loops ([AL])
1 function find_loop(P):
2      AL = new empty List
3      S = P.getStatuses()
4      FOR LL = 1 to (length of list)/2 DO
5          IF (LL == 1) THEN
6              FOR I=1 to len(S) − 1 DO
7                  IF (S[I] == S[index + 1] THEN
8                       AL.add(new A(list[i], 1))
9                  ENDIF
10             ENDFOR
11         ELSE:
12              I = 0
13              WHILE (I + 2 * LL < len(S)) DO
14                  L = new list as sublist of S with beginning in
                    S[I] and end in S[I + LL]
15                  LTC = new list as sublist of S with beginning in
                    S[I + LL] and end in list[(I + 2)*(LL − 1)]
16                  IF (L equals LTC) THEN
17                       AL.add(new A(L, LL))
18                  ENDIF
19                  I += 1
20              ENDWHILE
21          ENDIF
22      ENDFOR
23      return AL
24 endfunction
```

Having analyzed issue handling paths for many projects, we observed a multitude of path structures involving diverse combinations of states. It is therefore reasonable to distinguish path classes according to various criteria: the range of issue flow, timing, and structural features. For this purpose, we can define several profiles and path classes that are targeted at the analyzed problems. The classification process can be supported by regular expressions that define appropriate properties of searched paths, e.g., comprising a specified subsequence of states (algorithm A3_P). Some illustrations are given in Section 5.

Comments registered during issue processing show problems in issue resolution, deficiencies in issue description, misunderstandings of project stakeholders, etc. An interesting aspect is tracing the timing, frequency, and information features of comments. In the performed comment analysis, we use a previously developed classifier [22] which distinguishes four comment classes: question, response, information, and issue fixing confirmation. For each issue, the introduced algorithm A3_C derives the number of generated

comments, text size, category, structure of issue sequences, correlations with handling states, and timing features. Timing features include the delay (SD) to the first comment after the issue registration and the duration of the comment sequence (CD). Deeper analysis is targeted at comment sequence distribution in issues. Here, we use aggregated profiles of comment sequences from three perspectives (global, state, and path), defined as follows:

$$APC = \{[nc_i; <ct_{i1}, ct_{i2}, \ldots ct_{ij}>; ns_i]: i \ni C^*\} \mid R, \tag{7a}$$

$$APC(S_r) = \{[nc_i; <ct_{i1}, ct_{i2}, \ldots ct_{ij}>;ns_i]: i \ni C(S_r)^*\} \mid R, \tag{7b}$$

$$APC(P_s) = \{[nc_i; <ct_{i1}, ct_{i2}, \ldots ct_{ij}>; ns_i\}: i \ni C(P_s)^*\} \mid R. \tag{7c}$$

where nc_i is the number of comments in the i-th sequence, $<ct_{i1}, ct_{i2}, \ldots ct_{ij}>$ is the i-th sequence pattern, with ct_{ik} denoting comment type ($0 < k < j$), ns_i denotes the number of issues comprising i-th sequence type, and restriction R specifies the considered issue types and observation time range. C, $C(S_r)$, and $C(P_s)$ denote the set of comment sequences correlated with all considered issues, the sequences generated during issue processing in state S_r, and the sequences within the whole issue processing path P_s, respectively. The sets C^*, $C(S_r)^*$ and $C(P_s)^*$ comprise indices which label elements of relevant sets of comment sequences. We can focus on some profile features, e.g., dominant sequences, the distribution of comment categories in sequences, etc. Analyzing comment sequences, it is reasonable to use general comment categories (e.g., questions, information, response) to limit the number of possible structures of sequences. Some illustrations are given in Section 5.3.

The considered issue set IS needs to be specified by the project repository and additional filtering features (if needed) related to issue types, priorities, timing ranges, etc. The results of the algorithms are presented in an explicit form (derived sets, graphs, etc.) and multicolumn Excel tables comprising many detailed features which can be used by the analyst for deeper studies based on diverse filtering and correlations. Moreover, the available links to individual issues provide capabilities for deeper studies regarding selected paths to drill down and find explanations for strange processing results, diverse deficiencies in issue documentation, etc.

Algorithm 1 provides rich data on issue handling in a well-structured way, and it is the basis for further exploration with Algorithms 2 and 3. The number of considered parameters can be adapted to the further analysis goals. The basic list of parameters involves handling times within states and paths, issue features (type, priority, comment characteristics, reporter activity), etc. This list can be extended to cover other details and adapt to the needs of the considered project and the expert's interests. Diverse aggregated statistics are derived for states and paths derived in Algorithm 2. They cover all considered issues (specified by condition R). Issue flow is characterized by the introduced profiles of state or path load, while the issue handling time statistics cover the minimal, maximal, Q1–Q3 quartiles, and distribution of time ranges correlated with the number of relevant issues. Special attention is devoted to issue processing loops, assessed using Algorithm 3, which provides the loop parameters of loop structure, issue coverage, and timing features. The introduced taxonomy of issue handling paths supported by regular expressions and correlated with statistic features facilitates the identification of project deficiencies, anomalies, and good/bad practices (algorithm A3_P). Path exploration is enhanced with issue comment analysis (A3_C). The capabilities and usefulness of the algorithms are illustrated in relation to the provided analysis results in Section 5.

5. Experimental Results and Analysis

Based on the developed IHM, relevant algorithms, and introduced metrics and issue processing profiles, we analyzed both open source and commercial projects. The illustrated results focus on three observation perspectives: issue processing states (Section 5.1), issue handling paths (Section 5.2), and issue commenting activities (Section 5.3). We also discuss how these perspectives complement each other.

5.1. Statistics Related to Processing States

The basic IHM state parameters are the state entry and exit times of processed issues. Here, the problem of issues still being processed in the considered state needs to be considered. We can either neglect such issues or assume the exit time to be equal to the repository's final time stamp. Another possibility is attributing zero handling time to these issues. Fortunately, when dealing with repositories comprising many issues, such situations rarely appear. Some consideration is required regarding terminal states, e.g., *closed*, which should specify the completion of issue handling, so transitions to another state may not appear. However, sometimes we observed state changes from the *closed* state. The basic state features of IHM can be extended by considering state change profiles (e.g., probabilities of transitions to subsequent states) and actor activities (e.g., exchanging comments), etc. Some of these are commented upon in subsequent subsections.

Issues attributed to a selected IHM state are processed by assigned actors and, depending on the results, are transferred to a subsequent state. Tracing the time statistics of handling issues in IHM states, we can derive relevant time profiles defined by minimal, maximal, average, and Q1–Q3 quartile values. Creating these profiles, we can consider diverse restrictions, e.g., issue types (all issues, bugs, new functionalities), issue priorities, and project time ranges. As an illustration, in Table 3, we present a sample of state time profiles (Q2 and Q3 quartiles specified in days) related to all software bug issues (1896) in the Log4J2 project. It is reasonable to correlate state time profiles with the number of processed issues within these states. Sometimes, an issue is processed in the state more than once (looping), and hence we present two parameters: N_S—the number of processed issues in state S; and U_S—the number of unique issues ($U_S \leq N_S$). In the state R_F, looping covered ($N_S - U_S$) = 67 issues (this problem is analysed in Section 5.2.2). State specifications in the first column of the table are consistent with Jira; moreover, we introduced acronyms which are used in the subsequent text. Maximal handling times ranged from 148 up to several thousand days (typically several hundreds). For the highest-priority bug issues, the parameters Q2 and Q3 were in the ranges of 0.2–2 and 0.2–27 days (mostly below 10), respectively (maximal values ranged from a few days up to 150).

Table 3. State time profiles for the Log4J2 project.

State	Us	Ns	Q2	Q3
O (Open)	1841	1850	3.8	42.7
R_F (Res.fixed)	889	956	0.9	6.6
C (Closed)	823	881	0.25	33.25
InP (In Progress)	148	153	0.9	0.75
R (Reopen)	142	159	0.24	11.55
R_D (Res. Duplicate)	50	51	0.75	3.34
R_NP (Res. not a Problem)	44	44	0.26	2.75
R_WF (Res. won't fix)	24	24	0.1	0.09
R_CR (Res. Cannot reproduce)	18	18	0.5	11.15
R_NB (Res. not a bug)	17	17	0.7	2.6
R_Inv (Res. Invalid)	17	17	3.37	1324
R_Inf (Res. Information prov)	16	17	0.2	0.32

It is worth noting that high values of maximal processing time may relate to postponed bugs considered as having a negligible impact (however, they may contribute to so-called bug debt). They may also have a significant impact on the increased average values, so the Q2 and Q3 parameters can be more representative. The derived statistics can cover all registered issues (as in Table 2) or only the completed ones. In practice, the latter issues provide almost the same values (sometimes with higher maximal values). Typically, higher-priority issues are handled more effectively (lower time values: Q3 for blocker issues of 1–17 days and maximal values of 1–610 days). Parameter Q3, as compared with Q2, is typically 5–10 times higher for states handling a larger number of issues (in case of lower numbers, this is not as regular). Maximal values may exceed Q3 by over 100 times.

Having analysed state time parameters for several open source projects, we also observed diverse values depending upon the considered states. Sometimes, they are much higher than in the Log4J2 project, e.g., for the Exim project for the initial state (referred to as *NEW*), Q2 and Q3 assumed values of 19 and 376 days, respectively. For comparison, in Table 4, we provide state timing statistics for the MongoDB project (states covering up to four issues were skipped). It is worth noting that for the highest-priority issues (53), much lower time parameters were achieved: Q2 and Q3 in the ranges of 0.1–1 and 0.2–3.7 days, respectively (maximal values of 1–100 days). The number of considered issues was 13,435 (8405 bugs and 5030 new functionalities).

Table 4. State time profiles for the MongoDB project.

State	Us	Ns	Q2	Q3
NES (Needs scheduling)	12,644	14,725	0.2	4.9
NEM (Needs Merging)	1728	1765	0.1	0.7
INV (Invalid)	1678	2430	1.1	9
IPR (In progres)	8489	9702	0.2	2.8
OPE (Opened)	11,325	13,573	1.1	23.4
ICR (in Code Review)	8767	9316	1.9	5.8
NED (Needs verification)	12,400	12,466	<1 h	<1 h
C (Closed)	11,811	12,575	1	7.3
NET (Needs Triage)	88	90	0.2	2.8
WFU (Wait for user input)	437	723	1.9	17.9
BAC (Backlog)	2034	2124	17.1	70.2
BLO (Blocked)	298	332	7.1	28.7

Typically, for open source projects, processing times are higher than for commercial ones. For two commercial projects A and B and an *Open* state, we obtained processing times of 0.2 and 10 days, and for other states, the processing time was typically a fraction of a day up to 20 days. Low values were also observed in total path handling times (compare Section 5.2). For states with higher delays, we searched for the reasons for these delays, e.g., not involving enough actors (or their low competence). Similarly, we traced the reasons for maximal values; in practice, they may relate to neglected issues or closed ones without notification in the repository (undisciplined actor activities). In the case of Scrum technology, it is important to ensure that Q3 is lower than the sprint duration. We will also discuss timing issues in relation to issue handling paths (Section 5.2).

5.2. Profiles of Issue Handling Paths

The presented metrics in Section 5.1 are correlated with IHM states. Another observation perspective is targeted at tracing issue handling paths: their structures, issue flow, and timing properties. The developed algorithms derive the distribution of possible paths (sequence of states engaged in handling issues) with the relevant issue load and timing statistics. Here, we can also adapt the analysis to the IHM's restrictions R.

5.2.1. Structural Features

Analysing IHM path structures, we distinguish two taxonomy profiles, namely the path length (PL_P) and path state (PS_P) profiles, defined as follows:

$$PL_P = [l1\ (c1, n1);\ l2\ (c2, n2) \ldots lk\ (ck, nk)] \tag{8a}$$

$$PS_P = \{Pi{:}S1_i, S2_i, \ldots Sm_i \mid cp_i, 0 \le i \le r\} \tag{8b}$$

where $l1, l2, \ldots, lk$, $c1, c2, \ldots, ck$, and $n1, n2, \ldots, nk$ denote the path lengths (the number of consecutive states) in increasing order, the summarized issue flow of these paths, and the number of different paths of the specified length, respectively. PS_P is the set of all paths P_i composed of state sequences $S1_i, \ldots Sm_i$ and cp_i is the issue flow (coverage) of the i-th path. It is reasonable to list the paths in decreasing order of relevant flow coverage.

Issue flow (coverage) can be specified explicitly as the number of handled issues or the percentage of all covered issues. In simplified profiles, issue flow (c_i, cp_i) can be skipped. Other aggregated profiles can also be derived, e.g., considering path groups covering specified ranges of handled issues. These ranges can be specified in absolute or relative scales. In compact profiles, we can use ranges of state lengths. Deriving path profiles, we can also consider issue restrictions, R (issue types (bugs or new functionality), issue priority, issue reporting time, range of covered issues, etc.) or path classes (discussed in Section 5.3). Similarly, we can use generalized states equivalent to some individual ones, e.g., *Resolved_** corresponding to resolved states with diverse individual suffixes (e.g., fixed, rejected, or duplicated).

As an illustration, we present a sample of PL_P and PS_P profiles for some projects. In the commercial Scrum project P1, the ITS repository comprised 2378 issues (related to the year 2022), including 1555 bugs (1434 closed and 121 open) and 823 other cases (425 paths). The distribution of the last group was as follows: *User story*—274 (209 closed); *Task*—342 (282 closed); and *New Feature*—234 (134 closed). The derived PL_P profile was as follows:

PL_P(P1) = 23(4;4)—0.17%; 21(4;3)—0.17%; 20(5;5)—0.21%; 19(9;9)—0.38%; 18(8;8)—0.34%; 17(13;13)—0.55%; 16(12;12)—0.5%; 15(18;15)—0.76%; 14(17;17)—0.71%; 13(33;24)—1.39%; 12(16;14)—0.67%; 11(53;37)—2.23%; 10(49;27)—2.06%; 9(138;45)—5.8%; 8(82;35)—3.45%; 7(160;43)—6.73%; 6(341;27)—14.34%; 5(750;29)—31.54%; 4(164;16)—6.9%; 3(214;20)—9.0%; 2(148;8)—6.22%; 1(140;2)—5.89%;

Limiting IHM to bug issues (restriction, R), we obtained 1555 issues which were handled in 332 unique paths with the following distribution: (12–23)—8.54%; (8–11)—16.01%; (6–7)—14.02%; (5)—41.29%; (1–4)—20.13%. In the brackets here, we give the range of path lengths followed by the relevant issue coverage percentage. Similarly, we can explore the impact of other issue features on path profiles. Issues with the highest priority (32 closed) were handled within 16 unique paths with the following distribution: (7–20)—21.86%; (5–6)—40.62%; (2–3)—37.5%. Independently of the assumed IHM restrictions (R), the highest issue coverage is assured by paths with five and six states. For bug issues covering the period of 2017–2021 (4545 issues, 1070 paths), we obtained paths covering 2–50 states: paths with over 13 states covered 0.02–1% (total 4.8%) of the issues, while 6 state paths (119) covered 40.42% of the issues. A higher number of long paths can be attributed to the higher fluctuation of staff and the lower experience as compared with period of 2022, where the number of states was reduced due to discussions after analyzing the statistical results.

Another observation perspective is the path state (PS_P) profile. In Table 5, we present 10 paths with the highest coverage (N) for project P1, extended by the timing features (in days) discussed in Section 5.3. Specifying path states, we use the following state acronyms: *New* (*N*), *In Progress* (*IN*), *To be Tested* (*TbT*), *Verified/Closed* (*V/C*), *Reopened* (*REO*), *Need Info* (*NIF*), *Review* (*REV*), *Backlog-not ready* (*BN*), *backlog-Ready* (*BR*), and *Done* (*D*). It is worth noting that 68% of the paths cover single issues, resulting in 15% of all handled issues overall. Due to the high diversity of paths, we recommend using some aggregation in the analysis (Section 5.3).

For the open source project MongoDB, the ITS repository (covering 2 years) comprised 13,435 issues (8405 bug issues—7385 closed; 4257 improvement—3155 closed; 773 new feature—625 closed). The derived PL_P profile for the 832 unique paths was as follows:

PL_P(MongoDB) = 28(1;1)—0.01%; 27(1;1)—0.01%; 23(1;1)—0.01%; 22(2;2)—0.01%; 21(8;8)—0.06%; 20(6;6)—0.04%; 19(7;7)—0.05%; 18(9;9)—0.07%; 17(12;12)—0.09%; 16(17;17)—0.13%; 15(26;24)—0.19%; 14(34;30)—0.25%; 13(43;39)—0.32%; 12(102;84)—0.76%; 11(121;91)—0.9%; 10(295;158)—2.2%; 9(400;172)—2.98%; 8(686;157)—5.11%; 7(1948;159)—14.5%; 6(5023;119)—37.39%; 5(1630;71)—12.13%; 4(1617;42)—12.04%; 3(1161;22)—8.64%; 2(283;7)—2.11%; 1(2;2)—0.01%;

Table 5. Sample of bug issue handling paths for the project P1.

N	Path	AVG	MAX	Q2	Q3
576	N,IN,REV,TbT,V/C	25.2	340.2	6.9	21.2
106	N,TbT,V/C	29.8	332.3	3.8	17.3
100	N,V/C	26.4	218.2	5	17
48	N,IN,REV,TbT,Reo,IN,REV,TbT,V/C	60.3	283.7	30.3	65.4
45	N,NIF,IN,REV,TbT,V/C	46.1	212.9	15	49.9
23	N,IN,REV,TbT,Reo,TbT,V/C	30.6	112.1	22.1	36.2
22	N,IN,N,IN,REV,TbT,V/C	45.5	294.2	25.6	74.7
21	N,IN,V/C	30.3	131.3	10.2	41.3
19	N,NIF,IPrS,V/C	58.4	235.4	28.3	80.0
18	N,IN,NIF,IPrS,REV,TbT,V/C	48.2	248.7	27.4	49.3

Limiting IHM to bug issues (restriction R), we obtained 8405 issues (510 open) handled by 832 unique paths with the following distribution: (10–28)—2.45%; (7–9)—21.98%; (6)—41.22%; (2–5)—31.97%. For issues with the highest priority (54 blocker issues, 15 unique paths), the distribution was as follows: (8–10)—9.26%; (6–7)—72.2%; (2–5)—18.52%. For the lower-priority issues, the path length diversity was higher: *Critical* (2–16)—123 issues; *Major* (2–27)—7488 issues; *Minor/Trivial* (2–14)—231 issues. We identified that 69.9% of paths (586) were related to single issues, and they covered only 7% of all issues. The length of most of these paths ranged from 10 to 27 states; however, some short paths (with strange state sequences) occurred also, e.g., 14 paths involving three states. On the other hand, the 10 most populated paths (with a length range of 1–6) covered 67.9% of all bug issues. The number of unique paths (length 1–9) covering 10 or more issues was 56, covering 85% of issues overall. Paths covering more than 10 issues contributed 69.9% of the paths. The dominant path length was 6, relating to 95 paths which covered 40.42% of issues. We can also derive path profiles for issues of specified priorities. Issues with the priorities *blocker* (54), *critical* (121), *major* (7481), and *trivial/minor* (231) were handled by paths with length ranges of 2–10, 2–16, 2–28 and 2–14, respectively. It is worth noting that there are many long paths (high diversity). However, they are linked to a small number of issues; nevertheless, in total, they can cover a significant percentage of issues. The ratio of paths covering only a single issue related to all issues for the considered priorities in the considered project was 0.67, 0.82, 0.68, and 0.66, respectively. For *Improvement/new features* issues (5030 issues including 3780 closed issues and 760 paths), the path lengths ranged from 1 to 27, with dominant coverage for 3-7-state paths, covering 85% of issues.

The path state (PS_P) profile for the 10 most populated paths of the MongoDB project is given in Table 6. The used state acronyms are defined in Table 3.

Table 6. Sample of bug issue handling paths for the MongoDB project.

N		AVE	Max	Q2	Q3
2730	NV,NS,OP,IPR,ICR,CLO	19.4	511.0	5.8	19.0
649	NV,NS,OP,IPR,ICR,NM,CLO	19.0	348.0	5.9	18.7
513	NV,NS,OP,ICR,CL	10.7	372.8	3.3	8.4
370	NV,NS,OP,CLO	47.0	693.6	11.1	39.9
316	NV,NS,CLO	8.7	168.0	2.0	6.8
236	NS,OP,NS,IPR,ICR,CLO	11.9	172.9	4.1	11.3
229	NV,CLO	3.0	56.9	0.8	3.4
209	NV,NS,OP,IPR,CLO	27.7	229.9	9.8	31.2
174	NV,NS,INV,IPR,ICR,CLO	12.6	224.7	4.0	10.2
124	NV,NS,OP,BAC	375.8	825.9	366.2	584.2

A quite different PL_P profile relates to the Log4J2 project (1423 bug issues):

PL_P (Log4J2) = 9 (2, 2)—0.14%; 8 (5, 5)—0.35%; 7 (6, 6)—0.42%; 7 (9, 22)—6.3%; 5 (14, 42)—2.9%; 4 (22, 111)—7.8%; 3 (44, 463)—32.5%; 2 (37, 771)—54.2%

As compared with the presented profiles, PL_P (P1) and PL_P (MongoDB), PL_P (Log4J2) comprises a lower number of paths (99) with a lower number of states (2–9). Two- and three-state paths are dominant (54.2% and 32.5% issue coverage). The 10 most populated paths are presented in Table 7 (PS_P profile), covering 82.6% of bug issues. It is worth noting that most paths were terminated with a closed state (C). On the other hand, the structure of two-state paths was O, C (2.9%) and O, R_x (51.3%), where x corresponds to diverse issue resolution types, e.g., fixed (F), not a problem (NP), invalid (I), cannot reproduce (CnR), rejected (R), duplicated (D), etc. Handling times for the O,CnR path (14 issues) was very high (AVG, MAX, Q2, and Q3: 610.7, 2598.7, 206, and 619.8 days, respectively) due to the high effort required in reproducing the bug, probably resulting from inaccurate issue specification. The duplicate issues (path O, R_D) were detected quite fast (Q2, Q3: 1.5, 14.1 days); however, in case of one issue, over 2000 days was required.

Table 7. Sample of bug issue handling paths for the Log4J2 project.

N		AVE	Max	Q2	Q3
372	O, R_F	148.9	2489.0	14.6	126.9
320	O, R_F, C	53.6	755.2	10.1	47.1
279	O, C	207. 2	2982.1	1.2	66.2
52	O, IPr, R_F, C	40.9	568.8	9.1	29.9
51	O, IPr, R_F	74.2	660.2	21.1	95.1
30	O, R_D	104.6	2123.7	1.5	14.1
21	O, R_NP	105.5	1105.8	5.8	99.3
17	O, R_NP, C	70.6	561.3	13.2	53.9
16	O, R_F, Reo, R_F	192.6	192.6	65.3	219.9
15	O, C, Reo, C	17.8	17.8	0.9	10.4

The presented path profiles provide some perspectives on the path diversity, the impact of restriction (R) on the considered issues, etc. Having analyzed the path profiles for other open source projects, we also found projects with paths involving a lower number of states, e.g., 2–10. In two other commercial projects (A and B), the path structures involved 2–15 states. However, in a complex long-term commercial project, C [13], path lengths resulted in 4-44 states with a total number of paths of over 1500, covering issues at rate ranging from a fraction of a percent up to 10%. So, the dominating paths were not uniquely obvious. This was a long-term project with high fluctuation of actors. Moreover, the large number of used states created possible high path diversity. Higher path diversity is typical in projects involving many states. Issue flow coverage for longer paths is usually lower than for typical ones. The path profiles PL_P and PS_P provide some aggregated and detailed perspectives on issue handling sequences, facilitating the tracing of typical, optimal, and exceptional sequences (e.g., comprising loops) for further and more in-depth studies. In many projects, we observed 20–30 paths covering 50–70% of issues (Pareto principle). On the other hand, a lot of diverse paths covered 1–10 issues only. The existence of a relatively small number of dominating paths (covering many issues) can indicate mature projects. However, the many remaining paths may still cover a significant percentage of issues. The effective assessment and interpretation of so diverse paths are not easy tasks, and hence additional classification is needed considering the specific structural features and related comments. This is studied in the subsequent sections.

5.2.2. Path Classification and Aggregation

Having analysed path profiles for many projects, we observed high diversity in their properties. To systemize these studies, we introduced some path taxonomy at diverse

levels considering issue flow, structural, and other features (e.g., timing parameters). The first criterion results in identifying high (dominating), medium, and low flow paths (discriminated by predefined threshold values). Structural classifications involve path length and state sequences. These specifications can use regular expressions or other formulas. As an illustration, we present some regular expressions and correlated path statistics for the considered MongoDB project:

^[^,\n]*((,[^,\n]*){2}$)-3-state paths (22 paths covering 8.6% issues)

^[^,\n]*((,[^,\n]*){1}$)-2-state paths (7 paths, covering 2.1% issues)

These formulas define sets of aggregated paths for which we can specify the number of comprised paths, the distribution of path lengths, the distribution of issue flow (coverage), and the timing parameters. These specifications relate to the goal of the analysis, e.g., we can trace (i) terminal paths with the final *closed* state, (ii) decisive paths ending with an issue resolution state (e.g., *fixed*, *not a problem*, *duplicated*), and (iii) paths comprising specified states/sequences or their absence. Path analysis is supported by the developed algorithms. Such classification characterizes the development and maintenance processes and facilitates searches for "anomalous paths" showing some deficiencies in problem handling, e.g., paths with a repeated *Closed* state or a bouncing *Resolved_x* state, with *reopen* or the repetition of a specified state, and a percentage of correctly closed paths or resolved ones. Having identified such paths, we can drill down into relevant issue reports, comments, etc. They can also be correlated with engaged reporters, actors, timing properties, or other dependencies. Some examples are discussed later.

Well-organized issue handling paths should start with the project initial state, e.g., *Open*, *New*, etc. Nevertheless, in many projects, beyond the dominant initial state, there are some paths with other initial states, usually reflecting management deficiencies. For example, in the commercial project P1 (period 2017–2021), the initial path state statistics were as follows (percentage of paths/covered issues): *New* (61%/72.6%), *Open* (17.6%/11%), *Created* (19.7%/16%), *To do* (1.7%/0.4%). Here, due to the earlier period of project development, some diversity of the used initial states appeared (ambiguous interpretation by reporters). In the later period (2022), this was unified, with the dominant state being *New* (99.2%/99.7%), followed by *Backlog not ready* (0.8%/0.3%). In the case of MongoDB project, the dominant initial state (for bug handling) was *Needs Verification* (91.7% paths, 84% issues), followed sporadically by other states, namely *Needs Scheduling* (7%, 8.5%), *Needs Triage* (0.8%, 5.4%), and *Open* (0.2%, 1.5%). For *Improvement/New Feature* issues, this statistic was as follows: *Needs Verification* (92.4%, 84%), *Needs Scheduling* (7%, 11.5%), *Needs Triage* (0.3%, 2%), and *Open* (0.3%, 2.5%). This results from some inconsistency within the project team (appearance of new reporters).

The sequence of states in issue handling paths should reflect the related activities of project actors. Unfortunately, quite often, this is not reported accurately in the IST repository, e.g., lacking states with the issue resolution type (comparing Tables 2 and 3). An interesting illustration of this problem is the two-state path *Needs Verification*, *Closed* (NV,CLO) of MongoDB (Table 5). The profile of this path does not reveal how the covered issues were resolved. We can trace this by analyzing issue descriptions and included comments. We discuss this in Section 5.3. On the other hand, we encountered a diversity of long paths (compare Section 5.2); as an illustration, we provide an example from the MongoDB project (long handling time: 635.4 days):

NV,NSch,Inv,NSch,Inv,NSch,O,IPrs,O,IPrs,O,ICR,IPrs,O,ICR,IPrs,ICR,IPrs,O,Back,

Typically, such paths comprise state repetitions which show processing deficiencies (anomaly). The developed Algorithm 3 identifies cycles (loops) in state paths (Section 4.3.3). It provides path structures with underlined cycles and some aggregated statistics (loop length, issue coverage). Some illustrations are given in Tables 8 and 9. For the commercial project P1 (period 2022), loops appeared for paths with at least six states. We identified

66 paths (19.9% of all paths) with loops (three to five states) which covered 83 bug issues (5.34%). For example, the path *New*, *In Progress*, *Review*, *To be Tested*, *Reopened*, *IN PROGRESS*, *Review*, *To be Tested*, *Reopened*, *In Progress*, *Review*, *To be Tested*, *Verified/Closed* comprises five state loops (repeated three times in the state sequence) and covers nine issues. Loops did not appear in paths covering high-priority issues. Most loops covered three states and were initiated by programmers—loops starting with *In_Progress* or *To be Tested* states were initiated by developers, and those starting with *New* or *Reopen* were initiated by testers. Most looped paths related to single issues (58 out of 66). Loops with the *Reopen* state usually correlated with incorrect bug fixing. Loops did not correlate with the number of involved actors. For other issues (*New Feature*, *User Story*, *Task*), we detected seven loops covering seven issues (0.9%). Analyzing loops in the earlier period (2017–2021) of the P1 project, the percentage of bug issues handed in loops (within 23.4% of paths) was higher, at 7.6%. A sample of the paths with loops is presented in Table 8.

Table 8. Anomalous paths in the commercial project P1 (bug issues).

N	Path	AVG	MAX	Q2	Q3
9	N,**IN**,**REV**,**TbT**,**REO**,**IN**,**REV**,**TbT**, **REO**,**IN**,**REV**,**TbT**,V/C	86.5	213.1	32.8	133.2
4	N,**IN**,**NIF**,**IN**,**NIF**,**IN**,REV,TbT,V/C	41.9	67.8	42.6	58.4
2	N,**IN**,**REV**,**TbT**,**REO**,**IN**,**REV**,**TbT**,**REO**,**IN**,**REV**, **TbT**,**REO**,**IN**,**REV**,**TbT**,**REO**,**IN**,**REV**,**TbT**,V/C	154.1	252.2	154.1	252.2
2	**N**,**IN**,**N**,**IN**,**N**,**IN**,REV,TbT,V/C	42.4	74.0	42.4	74.0
2	N,**IN**,**REV**,**IN**,**REV**,**IN**,N,IN,REV,TbT,V/C	156.5	175.0	156.5	175.0

Table 9. Anomalous paths in the MongoDB project (bug issues).

N	Path	AVG	MAX	Q2	Q3
44	NED,NES,OPE,IPR,ICR,**CLO**,**CLO**	43.1	366.2	11.0	37.3
37	NED,**INV**,**WFU**,**INV**,**WFU**,**INV**,CLO,CLO	43.0	240.9	32.3	49.1
12	NED,**INV**,**WFU**,**INV**,**WFU**,**INV**,**WFU**,**INV**,CLO	113.5	545.1	59.4	133.3
11	NES,OPE,**NES**,**NES**,OPE,IPR,ICR,CLO	13.7	35.2	11.0	15.5
9	NED,**INV**,**WFU**,**INV**,**WFU**,**INV**,**WFU**,**INV**,**WFU**,CLO	112.0	281.4	91.2	117.6

Having analyzed some open source projects, we identified that 0.1–6% of issues were handled by paths with loops. In the open source project MongoDB, we detected 174 paths (20.1%) with loops, covering 350 bug issues (4.9%). Most of these paths (142) covered single issues. For the issues *Improvement*, *New Feature*, we found that 17.5% of paths had loops, covering 4.3% of issues. Most loops involved sequences of three states initiated by developers. However, some single-state loops comprised the repetition of the *Closed* state (12 loops) separated by short delays, probably resulting from faults in Jira API usage. Beyond regular loops, we can also trace repeating states in paths. In MongoDB, we observed repetitions of the *Open* state in 44.5% of paths, which covered 7.9% of all bug issues. However, the 10 paths with the highest issue coverage (66%) were free of state repetition anomalies. Among the 100 paths with the highest issue coverage, *Open* state repetition occurred in 22% of paths, covering 3.4% of issues. In the case of single-issue paths (70% paths, 6.9% of all issues), this anomaly was observed for 50% of paths. Most repetitions of the *Open* state followed the *Closed* state (as well as *Needs scheduling* and *Backlog*), indicating reporting and resolution deficiencies.

In Apache Lucene, 2.46% of bug issues and 2.70% of new functionality issues were handled in loops. Most (about 80%) of these loops involved the repetition of one state (e.g., *Open* state) resulting from some deficiencies. In the Arrow project, only a single regular loop was detected within bug handling paths: O,IPrs,**C**,**Reo**,**C**,**Reo**,**C** (covering one issue). Anomalous paths were mostly related to diverse patterns comprising the *Reopen* state or

some randomly repeated states. Among the 10 paths with the highest issue coverage (in total 95.5%), only 2 comprised the *Reopen* state, and they covered less than 1% of all bugs. On the other hand, among the 33 single-issue paths, 85% comprised the *Reopen* state. The repetition of the *Open* state appeared in 7 paths (covering 0.9% issues) among all 78 paths (covering 6047 issues). Other anomalies were related to the repetition of *Resolved_x* states with the same or diverse specification of x (in most cases, *Resolved_fix*, and sometimes the suffix *fix* was replaced by *duplicate*, *not a problem*, or *rejected*). They appeared mostly in low-coverage paths (20.5% paths), thus handling of 1.1% issues.

In the case of the Log4j2 project, only five loops were identified for bug issues—each handled a single issue (in the brackets, we give the handling times in days):

O, C, Reo, C, Reo, C, Reo, C (107.6); O, C, Reo, C, Reo, C (0.3); O,R_F, Reo, R_F, Reo, R_F, C (181.2); O,R_F, Reo, R_F, Reo, R_F (282.6); O, R_NP, Reo, R_NP, Reo, R_NP, Rep, R_F, C (36.1).

Deriving specified path classes, we can extract paths with searched features, comprising rare states or suspicious state transitions which require deeper studies. For example, in the commercial project P1 (period 2022), paths with *Need Info* (3%/1%), *Reopen* (56.9%/24.5%), and *Blocked* (2.7%/1%) were extracted. Repetitions of these states within paths can be considered as concerning and requiring deeper investigation. Similarly, two-state paths terminated with the *Closed* state are ambiguous. In Mongo DB, they constituted about 3% of bug issues, while in other open source projects, they constituted up to 6% of issues. Meanwhile, three-state paths, comprising *Initial state*, *In Progress*, and *Closed*, showed some information deficiencies (7% of bug issues). Studying these issues, we found some additional information in the included comments (e.g., invalid issue). Lacking states specifying issue resolution is not good practice (this state is not encountered in Mongo DB). Handling *New Feature* and *Improvement* issues showed a lower percentage of two-state paths (1%). In some projects, diverse versions of the *Resolved* state appear, e.g., *Resolved_fix*, *Resolved_duplicate*, *Resolved_invalid*. The appearance of diverse *Resolved* states within the same path shows some chaos in the handling process (can be treated as some type of looping), similarly to the *Reopen* state—such situations are worth deeper investigation.

We can also trace odd transition states in issue handing paths or those that confirm good practices. The path analysis algorithm developed provides the distribution of the next states for a specified state. It can only relate to direct successors or all successors until the end of the path. For example, in Mongo DB (bug issues) for the *Closed* state, we identified nine diverse successive states; fortunately, such situations are rare (0.3% of issues). For the *In Code Review* (69% of issues) state, we identified 12 successive states, but the dominant ones were *Needs Merge* (18.3%), *In Progress* (5.8%), *Open* (5.7%), and *In Code Review* (4.4%). For *New Feature/Improvement* issues, these statistics were 22.8%, 8.8%, 8.2%, and 7.4%, respectively. This confirms some imperfections in the code review process; moreover, the code merging process is not clearly documented.

Long paths may result in long handling times or a higher load of project actors (more people engaged in problem resolution). On the other hand, short paths may not reflect all issue processing activities. A high diversity of path structures is not consistent with typical activity sequences recommended in project development (Section 3). Some states are missing (e.g., verification, quality assurance, etc.), while others are repeated. This results from reporting negligence, unrevealed issue dependencies, triaging shortcomings, actor fluctuations, etc. Comparisons of the derived statistics within the considered project life cycle and the different projects provide useful information for experts and project managers.

5.2.3. Path Timing Features

The time parameters of the IHM states provide only local perspectives on the issue processing phases, while the total issue handling time for the whole project presents general statistics (Section 4.1). More detailed insights regarding handling time require tracing it in relation to IHM paths. The processing time view from the state perspective is presented in

Section 5.1. The timing parameters for paths differ, and they result from the aggregation of correlated issues.

Typically, most issues are handled by some dominant paths. This is illustrated in Section 5.2 (Tables 4 and 5). In the MongoDB project, maximal issue handling times ranged from 57 to 826 days (often several hundreds), referring to some individual issues and showing some accidentality. The Q2 and Q3 ranges were 1–11 days and 3.4–40 days, respectively. However, the time parameters of the last path in Table 5 (124 issues) differ significantly (several hundred for Q2 and Q3). In fact, these issues were opened after a long period for further processing. Restricting timing features to the dominant paths is not satisfactory due to the high diversity of paths and many paths covering a low number of issues (typical for many projects—see Section 5.2). Hence, we introduced the path timing profiles defined by three values:

$$TP_i = \{\Delta_i, n_i, i_i\}, \Delta_i, = a_i - b_i \quad (9)$$

where Δ_i denotes the range (in days) of the investigated time profile TP (e.g., Q2, Q3, max), while n_i and i_i denote the number of paths and issue coverage (%) related to the i-th parameter range. The index i runs subsequent time ranges in increasing order. Such profiles are mostly interesting for the dominant paths (with high issue coverage)—they provide some insights regarding the effectiveness of typical issue processing sequences. As an illustration, we present TP profiles (Q2, Q3 quartiles) for the commercial project P1 (period 2017–2021) considering 26 dominant paths:

$$TP(Q2)_1 = \{4–9, 13, 72\%\}, TP(Q2)_2 = \{10–18, 9, 20\%\}, TP(Q2)_3 = \{35–58, 5, 8\%\}$$

$$TP(Q3)_1 = \{13–30, 11, 66\%\}, TP(Q3)_2 = \{35–58, 9, 22\%\}, TP(Q3)_3 = (79–94, 5, 8\%\}$$

The maximal path handling time range was 100–881, with dominant values of 375–881 (minimal value of 39, with 66 related to 2 paths covering 3% of issues). This proves that in the considered project, most issues are handled in up to 30 days (single Scrum sprint).

In case of open source projects, issue handling times are usually higher, e.g., for the Groovy project, we received the following time profiles (related to 26 dominating paths):

$$TP(Q2)_1 = \{20–43, 10, 35\%\}, TP(Q2)_2 = \{50–128, 7, 61\%\}, TP(Q2)_3 = \{144–356, 6, 4\%\}$$

$$TP(Q3)_1 = \{42–150, 8, 42\%\}, TP(Q3)_2 = \{208–457, 6, 55\%\}, TP(Q3)_3 = (500–661, 6, 3\%\}$$

Nevertheless, we identified some anomalies (Q2 = 2210, Q3 = 2369 days) for a path covering 19 issues. The maximal values for most paths ranged between 1722 and 4225 days, while for 12 paths, the maximal value was between 500 and 1500 (5% issues), while it was 147 and 57 days for two paths.

Better results, as compared with Groovy, were obtained for the MongoDB project (8405 bug issues):

$$TP(Q2)_1 = \{1–5, 57, 19, 58\%\}, TP(Q2)_2 = \{5–14, 114, 57, 63\%\}, TP(Q2)_3 = \{15–30, 139, 5, 37\%\}$$

$$TP(Q3)_1 = \{1–6, 70, 9, 92\%\}, TP(Q3)_2 = \{7–14, 59, 14, 77\%\}, TP(Q3)_3 = \{14–30, 132, 46, 19\%\}$$

The maximal handling times for some paths reached up to three years. For a significant percentage of issues (70.88%), Q3 was below 30 days. Nevertheless, for some paths covering a few issues, a high handling times appeared. Similarly, good results were observed in the Arrow project: a Q3 below 30 days for 66.99% of bug issues (a Q2 below 30 days covered 83.56% of issues). On the other hand, poor results were obtained for the Lucene project: a Q3 below 30 days covered 15.16% of issues, while for Q2, this was 22.95%. These results are due to inefficient issue processing, which can be traced in deeper analyses of paths, states, comments, and correlations with other issue documentation or processing features.

Here, we observed 137 paths (with lengths in the range of 1–10) covering a total of 3983 bug issues. The timing characteristics (Q2 and Q3 in days) for some high-coverage (N issues) paths were as follows:

p_1: *Open*, *Resolved_Fixed*, *Closed*: N = 1818; Q2 = 83.7, Q3 = 236.7

p_2: *Open*, *Closed*: N = 208; Q2 = 24, Q3 = 327.4

p_3: *Open*, *Resolved_Fixed*, *Reopen*, *Resolved_Fixed*, *Closed:* N = 157; Q2 = 94, Q3 = 247.8

p_4: *Open*, *Resolved_Invalid*: N = 74; Q2 = 0.4, Q3 = 1.4

Paths involving bug fixing states (p_1, p_3) show high handling times, as opposed to the path with the identification of invalid bugs (p_4). The two-state path p_2 shows the highest resolution time, and this does not reflect the involved activities; they can be derived by analyzing comment sequences. In the case of Scrum projects, the timing effectiveness of issue handling can be measured as the percentage of issues postponed to subsequent sprints. In the considered project P1 (period 2022) for 10 subsequent sprints, this percentage was 6.8%, 6.2%, 15.6%,11%, 10%, 6.8%, 15%, 6.5%, 10%, and 3.1% for all issue types, respectively. However, for bug issues, the postponed issues were negligible. Higher values for *New Function* issues resulted mostly from the fact that they were transferred for handling with some delay (the decision of the project owner). In classical projects, we can check the ratio of issues exceeding the deadline date, but this feature is rarely encountered in repositories and is usually attributed to a fraction of issues.

5.3. Comment Sequences

The issue handling process is supported by comment exchanges between project stakeholders. They facilitate explaining sources of uncertainty, e.g., the imprecise specification of reported problems, their localization, the context of appearance, etc. In [22], we discussed issue comment classification based on machine learning schemes. Combining this classification algorithm with the IHM allowed us to derive comment sequence profiles as defined in Section 4.3.

As an illustration, we present some results referring to the MongoDB project. We identified 722 unique comment sequences which appeared in 7156 issues within the repository, comprising 8405 bug issues, and hence 84.7% of issues comprised comments. In the APC global profile (Equation (7a)), short sequences dominated, with one to five comments. The distribution of comment sequence length was as follows: 1—4555; 2—1103; 3—569; 4—295; 5—176. These instances covered 6698 issues (80%). The longest comment sequence involved 57 comments (26 questions, 17 information and 14 positive) for a single issue; the first comment appeared about 30 min just after its registration (parameter SD), and the last comment appeared after 538 days. The third quartile of comment sequence duration (parameter CD) relevant to the 10 sequences with the highest issue coverage (5551) was in the range of 4–21 days. The APC profile of the 20 comment sequences related to the highest number of issues (restriction R1) was as follows:

APC | R1 = {1, <fix>, 4016; 1, <information>, 366; 2, <information, fix>, 355; 2, <fix, fix>, 348; 3, <fix, fix, fix>, 120; 2, <positive, fix>, 95; 1, <positive>, 95; 2, <information, information>, 78; 1, <question>, 78; 3, <information, information, fix>, 71; 2, <question, fix>, 53; 4, <fix, fix, fix, fix>, 48; 3, <question, information, fix>, 42; 3, <information, fix, fix>, 30; 3, <information, positive, fix> 29; 2, <information, positive>, 27; 2, <question, question>, 22; 4, <information, information, information, fix>, 21; 3, question, information, information>, 21}

The time (Q3 quartile) of the first comment appearance (CS) was in the range of several to 20 days, while sequence duration (CD) was typically in the range of several to a few dozen days. The maximal values of CS reached several years (postponed or neglected issues), the maximal values of CD reached several hundred days (appearing sporadically), while the Q3 quartile of CD correlated with the issue handling times in paths. A reduction in the number of comments requires accurate issue descriptions, better issue triaging, and careful selection of the project team, including systematic training and adapting to project specificities. This problem can also be studied in correlation with the APC(S_r) and APC(P_r) profiles defined in Section 4.3.

Analyzing the comment sequences reported during handling states (APC(S_r)), we found 14 states (5 with low issue coverage among all 19) with comment sequences. As an illustration, we present the APC(S_r) profile for the state *In code review*:

APC(In Code Review) | Ra = {1, <fix>, 4046; 2, <fix, fix>, 261; 3, <fix, fix, fix>, 55; 2, <information, fix>, 54; 1, <information>, 40; 4, <fix, fix, fix, fix> 16; 1, <positive> 14, 2, <fix, information>, 11};

The restriction condition Ra selects comment sequences covering more than 10 issues. We can also use aggregated statistics for states with many sequences:

Needs verification: 173 unique sequences covering 1000 issues, constituting 12% of all (7765) issues handled in this state; Q3(CD)—4 days;

Open: 107 unique sequences covering 952 issues, constituting 12.5% of all 7615 issues; Q3(CD)—14 days;

Investigating: 96 unique sequences covering 658 issues, constituting 49% of all 1334 issues; Q3(CD)—10.6 days.

Most issues were commented upon in the states *Needs merge* (91%), *In code review* (83%), and *Investigating* (49%) due to the high effort related to these activities. The durations of sequences (CD) were reasonable, reaching 20 days for the third quartile (Q3(CD)); nevertheless, some issues involved over several hundred days. Comment sequences in states are merged in issue handling paths. They are characterized by APC(P_r) profiles (7c). In the considered MongoDB project, we found 838 unique paths of handling bugs (with lengths of 1–27 states), among which 251 paths handled 2 or more issues, 20 dominant paths covered 6338 issues out of a total of 8405 (75.4%), and their length range was 1–7. Below, we present a sample of the derived statistics (each specified path is followed by issue coverage):

APC(*Needs Verification,Open,In Progress,In Code Review,Closed*—2752 issues) = {1, <fix>, 1996; 2, <information, fix> 152; 2,<fix,fix>, 141; 2, <positive, fix>, 41; 3, <fix,fix,fix>, 27; 3, <information, information, fix> 25; 2, <question,fix> 20; 3, <question, infor, fix> 16; 1, <information> 16, other sequences (122) with length below 10 and dominating 1–2 issues}—Q3(HT)—19.8 days

APC(*Needs Verification,Open,In Progress,In Code Review,Needs Merge,Closed*—654 issues) = {1, <fix> 437; 2, <fix, fix> 43; 2, <information fix> 38; 2, <positive, fix> 22; 2, <information,fix> 10; others dominating 1 issue, 49 comment seqences}, Q3(HT)—18.9 days

APC(*Needs Verification,Closed*—442 issues) = {1,<information> 51; 1, <fix> 48; 1, <positive> 16; 12 comment sequences}, Q3(HT)—11 days.

The presented distribution of the comment sequences is restricted to the most populated paths, and the remaining paths are summarized. Moreover, we added the Q3 quartile of path handling times (HT). APC(P_r) profiles are the consequence of APC(S_r) profiles (with some concatenation effect) with longer sequences and higher duration times. Nevertheless, short sequences are dominant, and the CD duration is acceptable. In longer sequences, we observed the repetition of *fix* and *information* comments. Repetitions of *fix* comments indicate higher effort in correcting the bug, suggesting that such issues could be partitioned into smaller parts. Long sequences could be analyzed to explain the reasons for these bugs and to suggest improvements in reporting to obtain more precise problem descriptions (which are helpful in diagnosis). Intensive commenting may result from misunderstandings between users and developers. In the case of short issue handling paths, comments can describe more processing details. In the presented profile APC(*Needs Verification,Closed*), the path structure does not reflect processing activities, so it can be considered as being too simplified with information deficit. We can search for this information in the relevant comments. In the considered case, only 53.6% of issues comprised comments (27.8% of these comments related to the fix category, which was the dominant information category). However, a significant percentage of issues were not commented upon (deficiency of reporting). Such analysis allowed us to improve issue handling in a commercial project by presenting and discussing our analysis results with project testers and developers and setting out precise requirements for commenting and reporting by the users.

The proposed profiles facilitate assessing comment activities, their significance, and the impact of issue handling processes. We can identify sources of excessive commenting sequences and their impact on issue resolution processes. They can be correlated with project actor fluctuations, lacking professional competence or experience, and thus trigger possible improvements.

6. Discussion

The presented scheme of analyzing issue handling processes significantly extends the scope of investigations as compared with those proposed in the literature (Section 2). It provides a multidimensional and detailed view of issue handling processes and extends the related knowledge database on software project development which can be used as a supervised expert system. General issue resolution features (Section 4.1) provide a rough assessment of the project progress, which can be expanded upon fine-grained studies with the IHM focused on the detailed investigation of issue processing (Sections 4.2 and 4.3).

Analyzing the effectiveness of issue processing, we consider the diversity of issue types, relevant attributes, and additional features. These data are extracted from ITS repositories with appropriate interfaces. Systematic analysis is ensured by the developed framework constructed around the introduced IHM combined with the relevant database. The included exploration algorithms derive assessment metrics and characterize profiles which provide a concise and intuitive presentation of issue handling properties. When loading data into databases, some normalization can be involved to facilitate result comparison between projects. This relates to defining compatible attribute values, related issue types, priority levels, resolution methods, sets of used handling states, and their meaning. Repository exploration includes higher levels of state or path classes, the specification of anomalous situations, and relevant statistic metrics and profiles (direct or aggregated). It can be adjusted according to issue types and other restricting conditions that specify the analysis' scope.

Issue handling processes can be analyzed based on diverse perspectives: within-project or cross-project studies. The first perspective can be divided into within-version and cross-version studies (depending on whether data are analyzed in the same version or diverse versions of the project). The proposed systemized and holistic study of the ITS repository facilitates the identification of characteristic issue processing schemes depending upon their types and other features, distinguishing between good and bad practices, which is helpful for improving development processes and creating relevant knowledge database. This is especially interesting in cross-project studies. Here, the gained experience is useful in enhancing within-project studies. In long-term projects, actor fluctuations, organizational changes, and introduced reporting refinements (e.g., state redefinition) impact issue handling patterns, which can also be explored with the introduced framework based on the IHM.

Statistics related to issue processing states (Section 5.1) show deficiencies in their usage (low issue coverage—U_S) or excessive handling times (high Q3 values). They may trigger the need for state redefinition/unification or project actor reallocation to processing states (considering their load and competence). More attention is needed to monitor issue paths' structural, timing, and issue flow profiles (Section 5.2). Good development practices result in a low number of well-structured dominating paths covering most issues, low handling times for high-priority issues, and satisfactory values for other issues (e.g., not exceeding the Scrum sprint period).

Some troubling situations relate to an excessive number of paths and their complex and nonuniform structures, e.g., many states, loops, intermingled issue resolution states, a lack of termination states, ambiguous two-state paths (e.g., *Open*, *Closed*), and aberrant state transitions (e.g., *fixed*→*duplicated*). They are identified via the introduced classification schemes. This can be correlated with timing features and tracing comment sequence profiles at the general, state or path levels (Section 5.3). Too long comment sequences or repeated information comments indicate actors' misunderstanding or imprecise reporting.

Many examples of issue handling deficiencies detected in real projects are presented and interpreted in Section 5. Anomalous situations can be analyzed to identify problem sources. They usually result from insufficient competence or irresponsibility of project actors, their fluctuation, negligence in reporting issue processing, deficiencies in issue attribute assignments, etc.

A high diversity of handling paths and their timing profiles may impact the accuracy of classical schemes used to predict issue handling times, resolution needs (fixing, rejecting, duplication), etc. So, these predictions can be arranged separately for specified issue categories and project actor groups. Moreover, issue handling profiles changing over time reveal intended or spurious modifications to development processes.

The derived features and handling profiles relate to fine-grained assessment. A higher level of evaluation can also be included, e.g., consistent with DORA metrics (https://docs.gitlab.com/ee/user/analytics/dora_metrics.html (accessed on 23 May 2024)): the deployment frequency (frequency of successful software releases), the lead time for changes (time needed from code committed to successful running in production), the time to restore service (from a failure), and the change failure rate (the percentage of deployments causing failure in production). The correlation of higher-level metrics with the proposed fine-grained features can also be investigated.

The gained experience and retrospective discussions with project stakeholders can trigger issue handling improvements. Some suggestions can be derived after interpreting the obtained analysis results during project progress and environment fluctuation. Here, a good knowledge of the problems and successes experienced in other projects is helpful. The identified unsatisfactory issue handling schemes (e.g., paths) require further interpretation by mining issue descriptions/comments and correlations with other attributes. We observed insufficient issue reporting in many open source projects. This resulted from imprecise attribute specification by the project manager, project actor fluctuations, reporting/commenting inaccuracies, actor negligence/incompetence, or bad habits. Such analysis of historical data for the project P1 resulted in positive improvements as it progressed further (better granularity of project tasks, issue attribute refinements, team changes/training, etc.). In the case of commercial project assessment, another problem arises due to data access limitations (confidence), usually imposed by project owners or users. This can be mitigated by anonymization techniques, but quite often, such techniques are difficult to include in relevant data access agreements.

ITS analysis can be combined with repositories generated from the VCS using other tools, e.g., BugBuilder [1]. This can facilitate the evaluation of project progress, documentation quality, and bug localization and mitigation, and inspire novel approaches for project development or its improvements. Another aspect is including studies on issue dependencies [3] and project stakeholders' competence/capabilities. Lacking information on issue dependencies may result in lowering issue handling efficiency, e.g., the missing detection of duplicated issues, not reported the closing of such issues (unnoticed duplicates), and an issue blocking another issue's resolution, meaning that it should be resolved sooner.

The presented studies were illustrated with results related to selected real projects, so some comments are needed on the validity of our research.

The construct validity of the results is concerned with the accuracy and representativeness of the introduced IHM. Flexibility in defining states and edges combined with imposed diverse restrictions (R) is consistent with practical issue handling schemes reported in used ITS tools, and only data extraction procedures should be adapted to ITS tool APIs. Our objective was to study the efficiency and anomalies in issue handling processes from three perspectives: the sequences of issue handling states, issue processing flow correlated with relevant state paths, and timing features. The presented experimental results based on rich data covering a set of open source and commercial project repositories demonstrated the usefulness of our approach.

External validity stands for the generalization of our studies. The presented research was based on a broad scope of analyzed issue tracking features derived in our previous

paper [12]. Here, the problem of data set representativeness (IST repositories) arises. This may be critical in some prediction problems. In our studies, an important concern was to ensure the universality of the IHM and its ability to cross-sectionally trace issue handling in relation to diverse attributes (features). The identified examples of good and bad practices (anomalies) do not exhaust all possible cases. Nevertheless, this limitation is mitigated by the flexibility of the analysis procedures. This flexibility includes configurations specified by regular expressions, diverse issue filtering for analysis, and statistical result presentation in selected cross-sectional perspectives.

7. Conclusions

This work presents an advanced model (IHM) for software issue handling analysis, in which a broad scope of issue features are considered. The graphical form of the model reveals issue processing phases and their dependencies resulting from issue flow over time. The IHM, combined with the issue repository database, provides the ability to derive aggregated (generalized) or problem-focused views on issue handling effectiveness. This is supported by qualitative and quantitative characteristics related to diverse observation perspectives (timing, operational, reporter activities, etc.). The qualitative view reflects typical or abnormal issue handling schemes and project stakeholder activities considering diverse issue types, priorities, etc. The quantitative studies are combined with the introduced metrics/profiles targeted at specified aspects providing detailed and systemized assessment. Both views supported by the presented algorithms provide a holistic insight into the project progress and threats. The identified anomalies/deficiencies can be effectively investigated by project stakeholders or experts. Investigating IST repositories, we acquire knowledge on software development useful for project stakeholders. This knowledge can help them to gain new insights into the problems encountered when developing software, deepen their domain knowledge, and stay informed about deficiencies/possible improvements.

The proposed approach was verified based on open source and commercial projects. The presented results revealed diverse issue handling profiles in the considered projects and specific anomalies which can be eliminated by introducing issue handling improvements. Nevertheless, some suspected behaviors may require interactions with project stakeholders, improving reporting accuracy, etc. On the other hand, by analyzing other projects, we can enrich our knowledge of good and bad practices. These can be enhanced by discussing the results with project stakeholders. Further research can include the correlation of the developed model with other project repositories, e.g., software version control, test reports, project recommendation/assessment opinions, etc. The advanced exploration of ITS provides additional information which can deepen the understandability of other software engineering problems, e.g., the detection of duplicated issues, issue triaging, and problem prediction and diagnosis.

Author Contributions: Conceptualization, J.S.; methodology, B.D. and J.S.; software, B.D.; validation, J.S. and B.D.; formal analysis, J.S. and B.D.; investigation, B.D.; writing—original draft preparation, J.S.; writing—review and editing, J.S. and B.D.; visualization, B.D.; supervision, J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Jiang, Y.; Liu, H.; Luo, X.; Zhu, Z.; Chi, X.; Niu, N.; Zhang, Y.; Hu, Y.; Bian, P.; Zhang, L. BugBuilder: An automated approach to building bug repository. *IEEE Trans. Softw. Eng.* **2023**, *49*, 1443–1463. [CrossRef]

2. Ramirez-Mora, S.L.; Oktaba, H.; Gomez-Adorno, H. Descriptions of issues and comments for predicting issue success in software projects. *J. Syst. Softw.* **2020**, *168*, 110663. [CrossRef]
3. Raatikainen, M.; Motger, Q.; Lüders, C.M.; Franch, X.; Myllyaho, L.; Kettunen, E.; Marco, J.; Tiihonen, J.; Halonen, M.; Männistö, T. Improved management of issue dependencies in issue trackers of large collaborative projects. *IEEE Trans. Softw. Eng.* **2023**, *49*, 2128–2148. [CrossRef]
4. Zheng, W.; Li, Y.; Wu, X.; Cheng, J. Duplicate bug report detection using named entity recognition. *Knowl.-Based Syst.* **2024**, *284*, 111258. [CrossRef]
5. Rakha, M.S.; Bezemer, C.-P.; Hassan, A.E. Revisiting the performance of automated approaches for the retrieval of duplicate reports in issue tracking systems that perform just-in-time duplicate retrieval. *Empir. Softw. Eng.* **2018**, *23*, 2597–2621. [CrossRef]
6. Herbold, S.; Trautsch, A.; Trautsch, F. On the feasibility of automated prediction of bug and non-bug issues. *Empir. Softw. Eng.* **2020**, *25*, 5333–5369. [CrossRef]
7. Mohsin, H.; Shi, C.; Hao, S.; Jiang, H. SPAN: A self-paced association augmentation and node embedding-based model for software bug classification and assignment. *Knowl.-Based Syst.* **2022**, *236*, 107711. [CrossRef]
8. Xi, S.-Q.; Yao, Y.; Xiao, X.-S.; Xu, F.; Lv, J. Bug Triaging Based on Tossing Sequence Modeling. *J. Comput. Sci. Technol.* **2019**, *34*, 942–956. [CrossRef]
9. Gupta, S.; Gupta, S.K. An approach to generate the bug report summaries using two-level feature extraction. *Expert Syst. Appl.* **2021**, *176*, 114816. [CrossRef]
10. Arya, D.; Wang, W.; Guo, L.C.; Cheng, J. Analysis and detection of information types of open source software issue discussions. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 454–464. [CrossRef]
11. Hanagal, D.N.; Bhalerao, N. *Software Reliability Growth Models*; Springer: Berlin/Heidelberg, Germany, 2021; ISBN 978-981-16-0025-8.
12. Reszka, Ł.; Sosnowski, J.; Dobrzyński, B. Enhancing software project monitoring with multidimensional data repository mining. *Electronics* **2023**, *12*, 3774. [CrossRef]
13. Sosnowski, J.; Dobrzyński, B.; Janczarek, P. Analysing problem handling schemes in software projects. *Inf. Softw. Technol.* **2017**, *91*, 56–71. [CrossRef]
14. Polaczek, J.; Sosnowski, J. Exploring the software repositories of embedded systems: An industrial experience. *Inf. Softw. Technol.* **2021**, *131*, 106489. [CrossRef]
15. Izadi, M.; Akbari, K.; Heydarnoori, A. Predicting the objective and priority of issue reports in software repositories. *Empir. Softw. Eng.* **2022**, *27*, 50. [CrossRef]
16. Ferreira Gomes, L.A.; da Silva, R.; Torres, M.; Côrtes, L. Bug report severity level prediction in open source software: A survey and research opportunities. *Inf. Softw. Technol.* **2019**, *115*, 58–78. [CrossRef]
17. Zheng, W.; Cheng, J.; Wu, X.; Sun, R.; Wang, X.; Sun, X. Domain knowledge-based security bug reports prediction. *Knowl.-Based Syst.* **2022**, *241*, 108293. [CrossRef]
18. Peters, F.; Tun, T.T.; Yu, Y.; Nuseibeh, B. Text filtering and ranking for security bug report prediction. *IEEE Trans. Softw. Eng.* **2019**, *45*, 615–631. [CrossRef]
19. Panichella, S.; Canfora, G.; Di Sorbo, A. "Won't We Fix this Issue?" Qualitative characterization and automated identification of wontfix issues on GitHub. *Inf. Softw. Technol.* **2021**, *139*, 106665. [CrossRef]
20. Nadeem, A.; Sarwar, M.U.; Malik, M.Z. Automatic issue classifier: A transfer learning framework for classifying issue reports. In Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Wuhan, China, 25–28 October 2021; pp. 421–426. [CrossRef]
21. Wu, X.; Zheng, W.; Pu, M.; Chen, J.; Mu, D. Invalid bug reports complicate the software aging situation. *Softw. Qual. J.* **2020**, *28*, 195–220. [CrossRef]
22. Dobrzyński, B.; Sosnowski, J. Text mining studies of software repository contents. In Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering, Prague, Czech Republic, 24–25 April 2023; Kaindl, H., Mannion, M., Maciaszek, L., Eds.; SciTePress: Setubal, Portugal, 2023; pp. 562–569, ISBN 978-989-758-647-7. [CrossRef]
23. Alonso-Abad, J.M.; López-Nozal, C.; Maudes-Raedo, J.M.; Marticorena-Sánchez, R. Label prediction on issue tracking systems using text mining. *Prog. Artif. Intell.* **2019**, *8*, 325–342. [CrossRef]
24. Mohsin, H.; Shi, C. SPBC: A self-paced learning model for bug classification from historical repositories of open-source software. *Expert Syst. Appl.* **2021**, *167*, 113808. [CrossRef]
25. Rath, M.; Mäder, P. Structured information in bug report descriptions—Influence on IR-based bug localization and developers. *Softw. Qual. J.* **2019**, *27*, 1315–1337. [CrossRef]
26. Ramírez-Mora, S.L.; Oktaba, H.; Gómez-Adorno, H.; Sierra, G. Exploring the communication functions of comments during bug fixing in open source software projects. *Inf. Softw. Technol.* **2021**, *136*, 106584. [CrossRef]
27. Hussain, M.; Khan, H.U.; Khan, A.W.; Khan, S.U. Prioritizing the issues extracted for getting right people on right project in software project management from vendors' perspective. *IEEE Access* **2021**, *9*, 8718–8732. [CrossRef]
28. Sarkar, A.; Rigby, P.C.; Bartalos, B. Improving bug triaging with high confidence predictions at Ericsson. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019; pp. 81–91. [CrossRef]

29. Wu, H.; Ma, Y.; Xiang, Z.; Yang, C.; He, K. A spatial–temporal graph neural network framework for automated software bug triaging. *Knowl.-Based Syst.* **2022**, *241*, 108308. [CrossRef]
30. Elmishali, A.; Sotto-Mayor, B.; Roshanski, I.; Sultan, A.; Kalech, M. BEIRUT: Repository mining for defect prediction. In Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), Wuhan, China, 25–28 October 2021; pp. 47–56. [CrossRef]
31. Sharma, P.P.; Sangal, A.L. Examining the Predictive Capability of Advanced Software Fault Prediction Models—An Experimental Investigation Using Combination Metrics. *E-Inform. Softw. Eng. J.* **2022**, *16*, 220104. [CrossRef]
32. Afric, P.; Vukadin, D.; Silic, M.; Delac, G. Empirical study: How issue classification influences software defect prediction. *IEEE Access* **2023**, *11*, 11732–11748. [CrossRef]
33. Rathi, S.C.; Misra, S.; Colomo-Palacios, R.; Adarsh, R.; Neti, L.B.M.; Kumar, L. Empirical evaluation of the performance of data sampling and feature selection techniques for software fault prediction. *Expert Syst. Appl.* **2023**, *223*, 119806. [CrossRef]
34. Nafreen, M.; Luperon, M.; Fiondella, L.; Nagaraju, V.; Shi, Y.; Wandji, T. Connecting Software Reliability Growth Models to Software Defect Tracking. In Proceedings of the IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), Coimbra, Portugal, 12–15 October 2020; pp. 138–147. [CrossRef]
35. Choetkiertikul, M.; Dam, H.K.; Tran, T.; Pham, T.; Ragkhitwetsagul, C.; Ghose, A. Automatically recommending components for issue reports using deep learning. *Empir. Softw. Eng.* **2021**, *26*, 14. [CrossRef]
36. Huang, Y.; da Costa, D.A.; Zhang, F.; Zou, Y. An empirical study on the issue reports with questions raised during the issue resolving process. *Empir. Softw. Eng.* **2019**, *24*, 718–750. [CrossRef]
37. Akbarinasaji, S.; Caglayan, B.; Bener, A. Predicting bug-fixing time: A replication study using an open source software project. *J. Syst. Softw.* **2018**, *136*, 173–186. [CrossRef]
38. Albuquerque, D.; Guimarães, E.; Tonin, G.; Rodríguezs, P.; Perkusich, M.; Almeida, H.; Perkusich, A.; Chagas, F. Managing technical debt using intelligent techniques—A systematic mapping study. *IEEE Trans. Softw. Eng.* **2023**, *49*, 2202–2220. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.