An Empirical Study of Delays in the Integration of Addressed Issues

Daniel Alencar da Costa*, Surafel Lemma Abebe†, Shane McIntosh†, Uirá Kulesza* and Ahmed E. Hassan†
*Department of Informatics and Applied Mathematics (DIMAp)
Federal University of Rio Grande do Norte - UFRN, Brazil
danielcosta@ppgsc.ufrn.br, uira@dimap.ufrn.br
†Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's Univeristy, Canada
{surafel,mcintosh,ahmed}@cs.queensu.ca

***Abstract*—Predicting the time required to address an issue (*i.e.*, a feature, bug fix, or enhancement) has long been the goal of many software engineering researchers. However, after an issue has been addressed, it must be integrated into an official release to become visible to users. In theory, issues should be integrated into releases soon after they are addressed. Yet in practice, the integration of an addressed issue might be delayed. For instance, an addressed issue might be delayed in order to assess the impact that it may have on the system as a whole. While one can often speculate, it is not always clear why some addressed issues are integrated immediately, while others are delayed. In this paper, we empirically study the integration of 20,995 addressed issues from the ArgoUML, Eclipse, and Firefox projects. Our results indicate that: (i) despite being addressed well before the release date, the integration of 34% to 60% of addressed issues in systems with traditional release cycle, and 98% of addressed issues in systems with rapid release cycle were delayed by one or more releases; (ii) using information derived from the addressed issues, we are able to accurately predict the release in which an addressed issue will be integrated, achieving a Receiver Operator Curve (ROC) area of above 0.74; and (iii) the workload of integrators is the most influential factor in our integration delay models. Our results indicate that integration can introduce non-negligible delays that prevent addressed issues from being delivered to users. Thus, solely focusing on the time to address an issue is not enough to truly assess how long it takes for users to see that the issue has been addressed in the software system.**

I. Introduction

Prior studies have explored several approaches to help developers to estimate the time needed to address issues (features, enhancements, and bug fixes) [1–7]. Such studies are useful for project managers who need to allocate development resources effectively in order to deliver releases on time without exceeding budgets.

On the other hand, users and contributors care most about when an official release of a software system will include an addressed issue. Although an issue may have been addressed, it may not be integrated into an official release for some time. Jiang *et al.* find that after a change has taken 1-3 months to complete the code review process, it takes an additional 1-3 months for that change to be integrated into the Linux kernel [8]. In this paper, we refer to the time between when an issue is addressed and when it is integrated into an official release as *integration delay*.

Although one can often speculate, it is not always clear why an addressed issue would not be integrated into an upcoming release. When the reasons for these integration delays are unclear, users and contributors may become frustrated. For example, on a recent Firefox issue, a stakeholder asked: *"So when does this stuff get added? Will it be applied to the next FF23 beta? A 22.01 release? Otherwise?"* [9].

To investigate why the integration of some addressed issues is delayed, we perform an empirical study of 20,995 issues collected from the ArgoUML, Eclipse, and Firefox projects. We investigate how much delay addressed issues typically have before integration. Furthermore, we investigate how often addressed issues are integrated into the next upcoming release and how often they are delayed. We then build prediction models with two goals: (1) to accurately predict the integration delay of an addressed issue, and (2) to understand the reasons of the delayed integration of addressed issues. We address the following three research questions:

RQ1: How long are addressed issues typically delayed by the integration process?

The integration of 34% to 60% of addressed issues in the studied systems with traditional release cycles (ArgoUML and Eclipse) was delayed by at least one release, despite being addressed well before of the official release date. Furthermore, 98% of the addressed issues were delayed by at least one release in the rapidly released Firefox system.

RQ2: Can we accurately predict when an addressed issue will be integrated?

Yes, our models achieve a weighted average precision of 0.59 to 0.88 and a recall of 0.62 to 0.88, with Receiver Operator Curve (ROC) areas above 0.74.

RQ3: What are the most influential attributes for estimating integration delay?

Our models derived more of their explanatory power from attributes that estimate the workload of the integration team at the time when an issue was addressed, rather than from attributes such as the priority or severity of the addressed issues.

TABLE I: The dates and versions of the first and last releases considered in the study, along with the total number of addressed issues and releases.

System	First release		Last release		No. of issues	No. of releases
	Date	version	Date	Version		
ArgoUML	18-08-2003	0.14	15-12-2011	0.34	3121	17
Eclipse (JDT)	03-11-2003	2.1.2	12-02-2007	3.2.2	3344	11
Firefox	05-06-2012	13	04-02-2014	27	14530	15

Paper organization. The remainder of the paper is organized as follows. Section II describes the issue lifecycle. Section III presents our empirical study by describing the studied systems, research questions, and results. Section IV discusses the threats to the validity of our conclusions. Section V positions our work with respect to previous studies. Section VI draws conclusions and proposes avenues for future work.

II. Background & Definitions

One of the main factors that drives software evolution are the issues that is filed by users, developers, and quality assurance personnel. Below we describe what issues are and the major steps involved in addressing and integrating them.

We use the term *issue* to broadly refer to bug reports, enhancements, and feature requests. Issues can be filed by users, developers, or quality assurance personnel. To track development progress, software teams use an Issue Tracking System, such as Bugzilla or JIRA to describe the status of issues.

Each issue in an ITS has a unique identifier, a brief description of the nature of the issue, and a variety of other metadata. Large software projects receive a plenty of issue reports everyday. For example, Mozilla and Eclipse received an average of 170 and 120 new issue reports daily from January to July 2009, respectively [10]. The number of filed issues is usually greater than the size of the development team. After an issue has been filed, project managers and team leaders *triage* them, *i.e.*, assign them to developers, denoting the urgency of the issue using priority and severity fields [11].

After being triaged, issues are then *addressed*, *i.e.*, solutions to the described issues are provided by developers. Generally speaking, an issue may be in an open or closed status. An issue is marked as open when a solution has not yet been found. We consider UNCONFIRMED, CONFIRMED and IN_PROGRESS as open statuses. An issue is considered closed when a solution has been found. Usually, a *resolution* is provided with a closed issue. For instance, if a developer made code changes to address an issue, the status and resolution combination should be RESOLVED-FIXED. However, if the developer was not able to reproduce the bug, then the status and resolution may be RESOLVED-WORKSFORME [12]. The lifecycle of the issues is available on the Bugzilla website [13].

Finally, addressed issues must be integrated into an official release in order to make them available to users. The releases that contain such addressed issues could be made available every few weeks or months, depending on the project release policy. Releasing every few weeks is typically referred to as a *rapid release* cycle, while releasing monthly or yearly is typically referred to as a *traditional release* cycle [14].

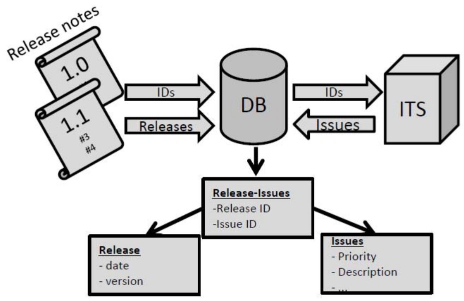


Fig. 1: Database construction overview. First, the issue IDs and releases information are extracted from release notes. Then, the issues are extracted from ITSs. The extracted issues and releases are matched, and the respective information are stored into a relational database.

III. Empirical study

In this section, we describe the studied systems, explain how the data was collected, and present the results of our empirical study with respect to our three research questions.

A. Studied Systems

To study when addressed issues are integrated into releases, we analyze one rapidly-releasing (Firefox) and two traditionally-releasing open source systems (ArgoUML and Eclipse). ArgoUML [15] is a UML modeling tool that includes support for all standard UML 1.4 diagrams. Eclipse [16] is a popular open-source IDE, of which we study the JDT core subsystem. Firefox is a popular web browser [17].

Table I shows the total number of collected issues and the considered releases in our empirical study. We focus our study on the releases for which we could recover a list of issue IDs from the release notes. We collected a total of 20,995 issue reports from the three studied systems. Each issue report corresponds to an issue that was addressed and could be mapped directly to a release.

In addition, we wish to compare a rapidly-releasing system to traditionally-releasing systems. Hence, we study Firefox, which has followed a rapid, 6-week release cycle since March 2011. Although ArgoUML and Eclipse both follow traditional release cycles, there is a longer interval between ArgoUML releases (median of 26 weeks) than Eclipse releases (median of 16 weeks).

B. Database Construction

Figure 1 provides an overview of our database construction approach. We create a relational database describing the integration of addressed issues in the studied systems. To do so, we collect data from two sources. We briefly describe each source, and the data that we collect from it below.

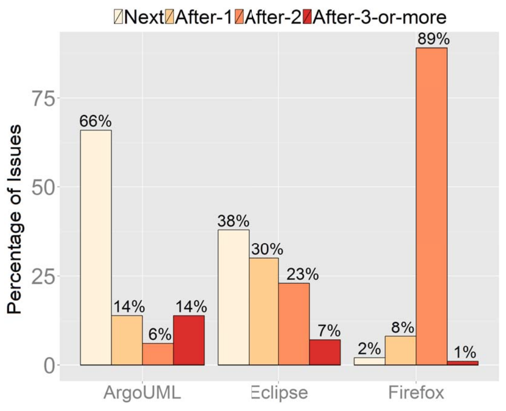


Fig. 2: Distribution of issues for each project

Release notes. In order to identify the release that an addressed issue was integrated into, we first analyze the release notes of the studied systems. A release note is a document that describes the content of a release. For instance, a release note might provide information about the improvements included in a release (with respect to prior releases), the fixed issues, and the known problems. Eclipse, ArgoUML, and Firefox publish their release notes on their respective websites [18–20].

Unfortunately, release notes may not mention all of the addressed issues that have been integrated into a release. This limitation hinders the investigation of issues that were addressed but have not been integrated, because we cannot claim that, an issue not listed in a release note, was not integrated. However, the issues that are listed in a release note have certainly been integrated. Thus, we chose to use release notes despite their incompleteness to identify integrated issues in order to reduce the noise in our dataset, i.e., the release where we claim an issue has been integrated, is almost certainly correct.

To retrieve the list of addressed issues that have been integrated into Eclipse and Firefox, we wrote a script to extract the listed issue IDs from the release notes and insert them into our database. The retrieved issue IDs are then used to collect issue reports from the corresponding ITSs. In our database, we also stored the dates and versions of the releases.

Issue Tracking Systems. Not all release notes from ArgoUML list the issues that were addressed in that official release, and when they do, only a few issues are listed (*e.g.*, 1-4) [21]. Hence, we rely on the ITS in order to map addressed issues to releases. We used the milestone field indicated in the issue reports to approximate the release that an issue was integrated into. Development milestones are counted towards the next official releases. For instance, the development milestones 0.33.7 [22] is counted towards the official release 0.34 [18].

Integration delay classification. In our study, we divide the collected issue reports into four classes: *next*, *after-1*, *after-2*, and *after-3-or-more*. The *next* class contains addressed issues that are integrated immediately. The *after-1*, *after-2*, and *after-3-or-more* classes contain addressed issues whose integration was delayed by one, two, or three or more releases, respectively. Figure 2 shows the distributions of the addressed issues among the classes for each studied system. Figure 2 shows that ArgoUML has the highest percentage of addressed issues that fall into the *next* class (71%), whereas *next* accounts for only 2% and 38% of addressed issues in Firefox and Eclipse, respectively. We chose the last change to the RESOLVED-FIXED status of an issue as the moment when the issue was addressed. For instance, in case an issue changed from RESOLVED-FIXED to REOPENED, and changed to the RESOLVED-FIXED status again, we say that the issue was addressed at the last change to the RESOLVED-FIXED status. Also, we use the RESOLVED-FIXED status rather than the VERIFIED-FIXED status because we found that all of the issues that are mapped to releases went through RESOLVED-FIXED before being integrated, while only a small percentage went through VERIFIED-FIXED. For instance, only 17% of addressed issues in Firefox went through the VERIFIED-FIXED status. We focus on issues that were resolved as FIXED because they involve changes to the source and/or test code [12].

C. Results

RQ1: How long are addressed issues typically delayed by the integration process?

Motivation. Users and contributors care most about when an addressed issue will be integrated in an official release rather than when it is initially addressed. We observed that some addressed issues are integrated in the next release, while others are delayed. It is not clear why some addressed issues take more time to be integrated than others. In RQ1, we investigate the delay between when an issue is addressed and when it is integrated. In order to better understand integration delays in each studied system, we also investigate if the delays differ between rapid and traditional release cycles. The analysis of RQ1 is our first step toward understanding why integration delays differs among addressed issues.

Approach. We compute the *integration delay* of an addressed issue as shown in Figure 3. We first collect the time when the resolution status of each issue changed to *RESOLVED-FIXED* from the ITS. To determine the moment of integration, we analyze the release notes of each project. Finally, we count the number of releases that occurred between the time when an issue status changed to the *RESOLVED-FIXED* and the release that it was integrated into.

Results. ***Addressed issues are usually delayed in a rapid release cycle.*** Figure 4 shows the difference between the studied systems regarding the time interval between their releases. The median time in days for Firefox (42 days) is approximately $\frac{1}{4}$ that of ArgoUML (180 days) and $\frac{1}{3}$ that of Eclipse (112 days). Unlike for Eclipse and Firefox, the distribution for ArgoUML is skewed. In addition, Figure 2 shows that the vast majority of addressed issues for Firefox are integrated *after-2* releases, whereas for Eclipse and ArgoUML, the majority are integrated in the *next* release. The reason for the difference may be the release policies followed in each

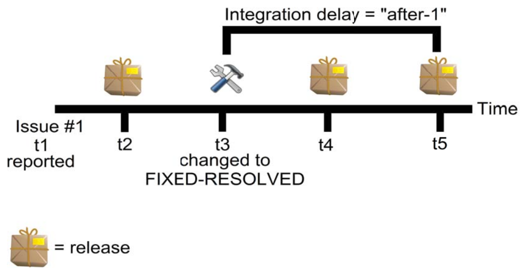


Fig. 3: Integration delay is computed by counting the releases that occur between when an issue status changes to RESOLVED-FIXED and the the date of the release note that lists that issue.

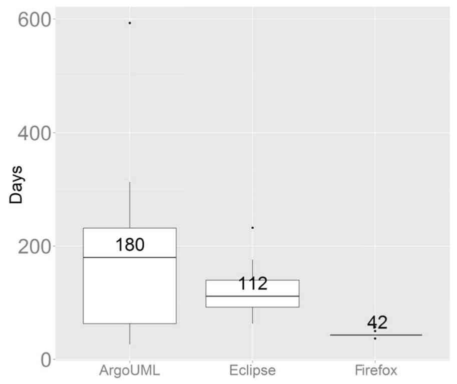


Fig. 4: Delays in days between releases of ArgoUML, Eclipse, and Firefox. The number shown over each boxplot is the median interval

project. For example, Figure 4, shows that Firefox releases consistently every 42 days (six weeks), whereas the times between ArgoUML releases vary from 50 to 220 days. The consistency of Firefox releases may lead to more delayed issues, since they rigidly adhere to a six-week release schedule despite accumulating issues that could not be integrated.

34% to 60% of addressed issues in the traditional release cycle systems were delayed by one or more releases. Figure 2 shows that 98% of the addressed issues in Firefox are delayed by one or more releases. Firefox is expected to have delayed issues due its rapid release cycles. However, 98% is still a considerably large percentage. Furthermore, even for the systems that adopt a more traditional release cycle, 34% (ArgoUML) to 60% (Eclipse) of the addressed issues are delayed by one or more releases. This result indicates that even though an issue is addressed, integration could be delayed by one or more releases.

Many delayed issues were addressed well before releases from which they were omitted. Addressed issues could be delayed from integration because they were addressed late in the release cycle, *e.g.*, one day or one week before the upcoming release date. In order to compare the rapid and traditional release cycles regarding whether delayed issues are addressed late in the release schedule, we computed the

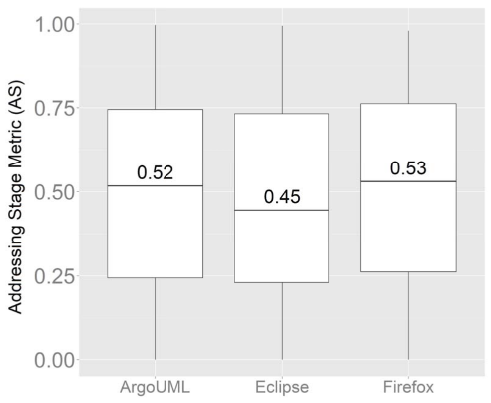


Fig. 5: Distribution of days between when an issue was addressed and the next missed release divided by the release window time.

Addressing Stage metric (*AS*) for each issue. The *AS* metric is calculated using the following equation: $\frac{days\ to\ next\ release}{release\ window}$, where *days to next release* is the number of days when an issue is addressed before the next release (*e.g.*, the time between *t3* to *t4* in Figure 3), and the *release window* is the time in days between the next upcoming release and the respective previous release (*e.g.*, *t4* to *t2*). An *AS* value close to 1 means that an issue was addressed too close to the next release, whereas a value close to 0 means that an issue was addressed at the beginning of a release cycle. Figure 5 shows the distribution of the *AS* metric for each project. The smallest *AS* median is observed for Eclipse, which is 0.45. For ArgoUML and Firefox, the median is 0.52 and 0.53, respectively. The *AS* medians are roughly in the middle of the release. Moreover, the boxes extend to cover between 0.25 and 0.75. The result suggests that, in the studied projects, delayed issues are usually addressed $\frac{1}{4}$ to $\frac{3}{4}$ of the way through a release. Hence, it is unlikely that most addressed issues miss the next release solely because they were addressed too close to an upcoming release date.

The integration of 34% to 60% of the addressed issues in the traditionally releasing systems and 98% in the rapidly releasing system were delayed by one or more releases. Furthermore, we find that many delayed issues were addressed well before releases from which they were omitted from.

RQ2: Can we accurately predict when an addressed issue will be integrated?

Motivation. Several studies proposed approaches to investigate the time required to address an issue [2–7]. These studies could help to estimate when an issue will be addressed. However, we find that integration delays when an addressed issue will be delivered to users. Even though several issues are addressed well before the next release date, their integration is delayed. For users and contributors, however, knowing the release in which an addressed issue will be integrated is of great

TABLE II: Attributes used to predict the integrated release of an addressed

Dimension	Attributes	Value	Definition (d)|Rationale (r)
Reporter	Experience	Numeric	**d:** Experience in filing reports for the project. It is measured by the number of previously reported issues of a reporter. **r:** An issue reported by an experienced reporter might be integrated quickly.
	Delay of previously addressed issues	Numeric	**d:** Measured by the median of the integration delays of previous issues that were reported. **r:** If previously addressed issues were integrated quickly for a reporter, future issues reported by the same reporter may also be integrated quickly.
Issue	Component	Nominal	**d:** The component specified in the issue report. **r:** Issues related to a given component (*e.g.*, authentication) might be more important, and thus, might be integrated prior to issues in less important components.
	Platform	Nominal	**d:** The platform specified in the issue report. **r:** Issues regarding one platform (*e.g.*, MS Windows) might be integrated prior to issues in less important platforms.
	Severity	Nominal	**d:** The severity of the issue. **r:** Issues with higher severity levels (*e.g.*, blocking) might be integrated faster than other issues. Panjer observed that the severity of an issue has a large effect on its lifetime for Eclipse project [23].
	Priority	Nominal	**d:** The priority of the issue. **r:** Higher priority issues will likely be integrated before lower priority issues.
	Description Size	Numeric	**d:** Description of the issue measured by the number of the words. **r:** Issues that are well-described might be more easy to integrate than issues that are difficult to understand.
Project	Integration Workload	Numeric	**d:** The number of issues in the RESOLVED-FIXED state at a given time. **r:** Having a large number of addressed issues at a given time might create a high workload on integrators, and may affect the number of addressed issues that are integrated.
Process	Number of Impacted Files	Numeric	**d:** The number of files linked to an issue report. **r:** An integration delay might be related to a high number of impacted files because more effort would be required to properly integrate the modifications [8].
	Number of Activities	Numeric	**d:** An activity is an entry in the issue's history. **r:** A high number of activities might indicate that much work was necessary to address the issue, which can impact the integration of the issue into a release. [8].
	Number of Comments	Numeric	**d:** The number of comments of an issue report. **r:** A large number of comments might indicate the importance of an issue or the difficulty to understand an it [3], which might impact the integration delay. [8].
	Number of Tosses	Numeric	**d:** The number of times the assignee has changed. **r:** The number of changes in the issue assignee might indicate a complex issue to address or a difficulty in understanding the issue, which can impact the integration delay. One of the reasons for changing the assigned developer is because additional expertise may be required to address the issue [8, 24].
	Comment Interval	Numeric	**d:** The sum of all of the time intervals between comments (measured in hours) divided by the total number of comments. **r:** A short comment time interval indicates that an active discussion took place, which suggests that the issue is important. [8].
	Churn	Numeric	**d:** The sum of the added lines and removed lines in the code repository. **r:** A higher churn suggests that a great amount of work was required to address the issue, and hence, verifying the impact of integrating the modifications may also be difficult [8, 25].

interest. In RQ2, we investigate if we can accurately predict the release in which an addressed issue will be integrated. The prediction could estimate for users and contributors when an addressed issue will likely be integrated.

Approach. In order to predict when an addressed issue will be integrated, we collected information from both ITSs and VCSs of the studied systems. We build models using metrics from four dimensions: *reporter*, *issue*, *project*, and *history*. We briefly describe each dimension below.

- **Reporter dimension** refers to information related to the reputation of an issue reporter. Issues reported by a reporter who is known to report important issues may receive more attention from the integration team.
- **Issue dimension** refers to the information provided about reported issues. Project team use this information to triage, address, and integrate issues. For example, integrators may not be able to properly assess the importance and impact of poorly described issues, which may, in turn, lead to integration delays.
- **Project dimension** refers to the status of the project when a specific issue is addressed. If the project team has a heavy integration workload *i.e.*, many addressed issues waiting to be integrated, the integration of newly addressed issues may be delayed.
- **Process dimension** refers to information related to the process of addressing an issue. An addressed issue that involved a complex process (*e.g.*, long comment threads, large code changes) could be difficult to understand and integrate.

Table II describes the information that we collected in each dimension. Henceforth, we refer to the collected information as attributes. For each attribute, Table II presents the type and the rationale behind its use in our models.

Prediction technique. We train our models using the *random forest* technique [26], which is known to have a good overall prediction accuracy and to be robust to outliers as well as noisy data. Model robustness is important for our study because the data in the ITSs are filed with subjective criteria and tend to be noisy [27]. In our study, we use the *random forest* implementation provided by the *bigrf* R package [28]. To build and test the prediction model, we use a 10-fold cross-validation and 100 trees in each forest.

Evaluation metrics. We use *precision*, *recall*, *F-measure*, and *ROC area* to evaluate our models. We describe each metric below.

Precision (P) measures the correctness of our models in predicting the release delay of an addressed issue. A prediction is considered correct if the predicted integration delay is the same as the actual integration delay it had. Precision is computed as the proportion of correctly predicted integration delays for each class (*e.g.*, next, after-1).

Recall (R) measures the completeness of a model. A model is considered complete if all of the addressed issues that were integrated in a given release r are predicted to appear in r. Recall is computed as the proportion of issues that actually appear in a release r that were correctly predicted as such.

F-measure (F) is the harmonic mean of precision and recall, *i.e.*, ($\frac{2 \times P \times R}{P + R}$). F-measure combines the inversely related precision and recall values into a single descriptive statistic.

ROC area is used to evaluate the degree of discrimination achieved by the model. The ROC area is the area below the curve plotting the true positive rate against false positive rate. The value of ROC area ranges between 0 (worst) and 1 (best). An area greater than 0.5 indicates that the prediction model outperforms a random predictor. We computed the ROC area for a given class c (*e.g.*, *next*) on a binary basis. In other words, the probabilities of the instances were analyzed as pertaining to a given class c or not for each class. Therefore, each class has its own ROC area value.

Results. ***Our prediction models achieve a weighted average precision between 0.59 to 0.88 and a recall between 0.62 to 0.88, with ROC areas of above 0.74.*** Figure 6 shows the precision, recall, F-measure, and ROC area of the prediction models. The boxplots represent the distributions of the 10 folds that were run for each class. Note that the highest precisions for ArgoUML and Eclipse were for *next* class (median of 0.97 and 0.69 respectively), whereas *after-2* had the highest precision in Firefox (median of 0.99). The classes with the highest precision values are also the ones with the majority of instances. In Firefox, the vast majority of the instances are in the *after-2* class, which may explain the low precisions for the other classes. On the other hand, recall values are higher for classes whose instances are a minority. For example, the highest medians for recall in ArgoUML are for *after-2* and *after-3-or-more*, whereas in Firefox the highest medians are for *next* and *after-3-or-more*. Eclipse models also have a relatively high recall for *next* (0.66). ROC area weighted averages are above 0.74 in all of the studied systems, which indicate that our model predictions are better than random guessing (ROC of 0.5). In summary, the models obtained a weighted average precision of 0.59 to 0.88 and a recall of 0.62 to 0.88. Although there is a room for improvement, our models provide a sound starting point for predicting the release that an addressed issue will be integrated into.

Our models achieve better F-measure values than Zero-R. We compared our models to Zero-R models as a baseline. For all test instances, Zero-R selects the class that contains the majority of the instances. Hence, the recall for the class containing the majority of instances is 1.0. We compared the F-measure of our models to the F-measure of Zero-R models. We chose to compare to the F-measure values because precision and recall are very skewed for Zero-R. For Firefox, Zero-R has an F-measure of 0.95 for the class *after-2*, which was equal to our model. For Eclipse, Zero-R always selects *next* and achieves an F-measure of 0.58 while our model achieves 0.68. Finally, for ArgoUML, Zero-R selects always *next* with 0.84, whereas our model achieves 0.91. The results show that our models yield better F-measure values than naive techniques like Zero-R or random prediction (ROC = 0.5) in the majority of cases.

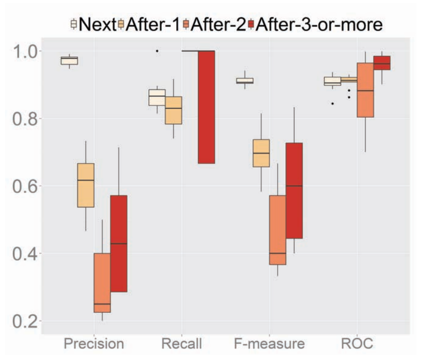
Our models outperform naive techniques such as Zero-R and random prediction, achieving an ROC area of at least 0.74.

RQ3: *What are the most influential attributes for estimating integration delay?*

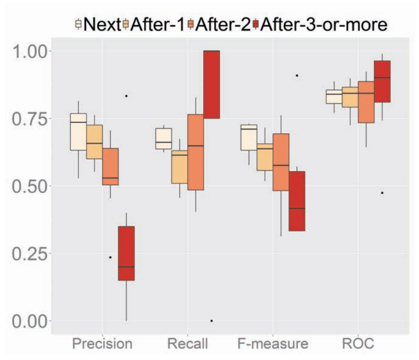
Motivation. In RQ2, we found that our models can accurately predict the integration delay of addressed issues. To build the models, we use attributes collected from ITSs and VCSs. As described in Table II, the attributes measure different dimensions related to the addressed issues. In RQ3, we investigate which attributes are influential in determining whether the integration of an addressed issue will be delayed.

Approach. To identify the most influential attributes that estimate integration delay of an addressed issue, we compute *variable importance* for each attribute in our models. The *variable importance* implementation we use in our study is from the *bigrf* package available in R [28]. This implementation computes the importance of an attribute based on Out Of the Bag (OOB) estimates. Each attribute of the dataset is randomly permuted in the OOB data. Then, the average a of the differences between the votes for the correct class in the permuted OOB and the original OOB is computed. The result of a is the importance of an attribute. The final output of the variable importance is a rank of the attributes indicating their importance for the model. Hence, if a specific attribute has the highest rank, then it is the most influential attribute that the prediction model is using to model integration delay.

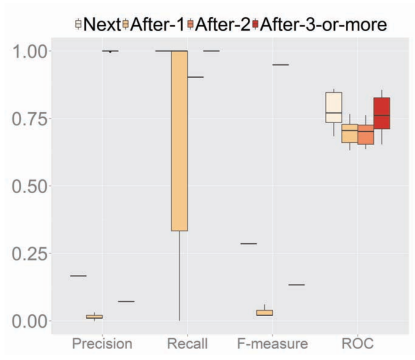
Results. ***The integrator workload has a bigger influence on integrator delay than the other attributes.*** By *integrators* we refer to team members that are responsible for integration tasks [8, 29, 30]. Figure 7 shows the distribution of the variable



(a) ArgoUML

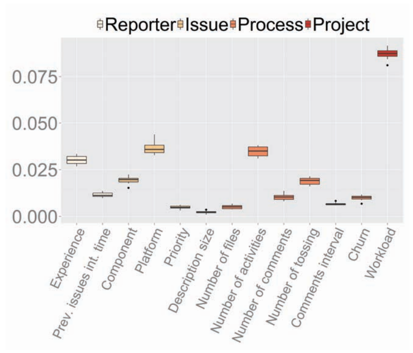


(b) Eclipse

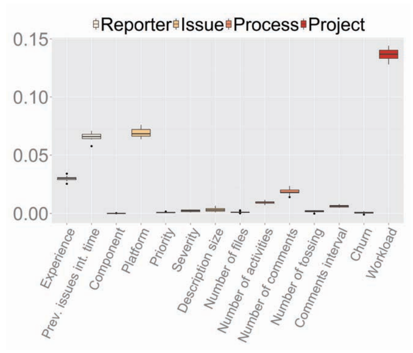


(c) FireFox

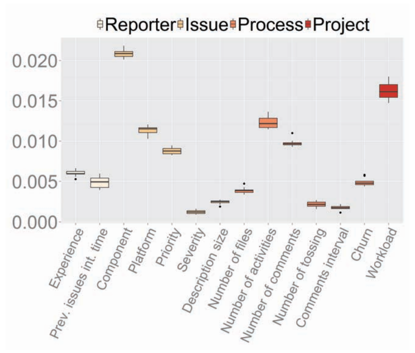
Fig. 6: The performance of our random forest classifiers.



(a) ArgoUML

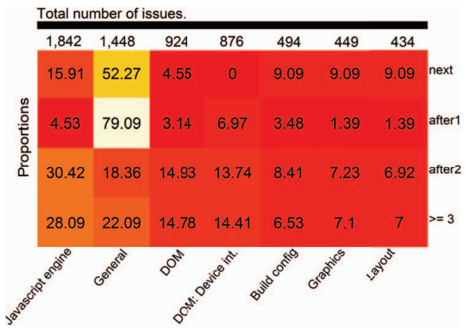


(b) Eclipse



(c) Firefox

Fig. 7: Distributions of variable importance values computed for the 10 folds that were run to train the models.



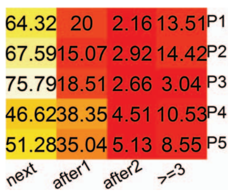
	Javascript engine	General	DOM	DOM: Device int.	Build config	Graphics	Layout
	1,842	1,448	924	876	494	449	434
next	15.91	52.27	4.55	0	9.09	9.09	9.09
after1	4.53	79.09	3.14	6.97	3.48	1.39	1.39
after2	30.42	18.36	14.93	13.74	8.41	7.23	6.92
>= 3	28.09	22.09	14.78	14.41	6.53	7.1	7

Fig. 8: The spread of issues among Firefox components. The darker the colors, the smaller the proportion of issues that impact that component.

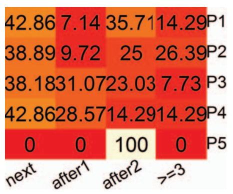
importance values computed for the 10-folds of our models. The result shows that *workload* is one of the most influential attributes for estimating the integration delay of an addressed issue. This finding holds for both the rapid and the traditional release cycles. This result confirms the intuition that issues that were addressed during periods of high integration activity are more likely to be delayed.

In Firefox, *component* is the most influential attribute. To better understand why *component* is considered influential, we counted how many addressed issues that are in each component. Figure 8 shows the top 7 Firefox components, each having more than 400 addressed issues. We analyzed the proportion of delayed integration in these top 7 components. Figure 8 shows that, for classes *next* and *after-1* the majority of issues are related to the *General component*, whereas for *after-2* and *after-3-or-more* the majority are related to the *Javascript engine* component. Addressed issues related to the *General* component may be easy to integrate, whereas issues related to the *Javascript Engine* may require more careful analysis before integration.

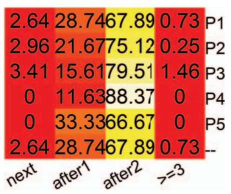
Severity and priority have little influence on issue integration delay. Users and contributors of software projects can denote the importance of an issue using the *priority* and *severity* fields. Previous studies have shown that priority and severity have little influence on bug fixing time [27, 31]. For example, while an issue might be severe or of high priority, it might be complex and would take a long time to fix. However, in the integration context, we expect that priority and severity would play a bigger role, since the issue has already been addressed. One would expect that integrators would try to



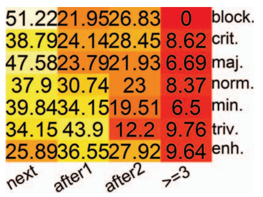
(a) ArgoUML Priority



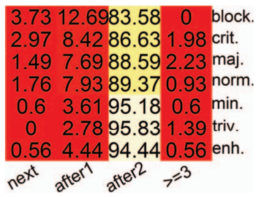
(b) Eclipse Priority



(c) Firefox Priority



(d) Eclipse Severity



(e) Firefox Severity

Fig. 9: The percentage of priority and severity levels in each issue class. We expect to see light colour in the upper left corner of these graphs, indicating that high priority/severity issues are integrated rapidly. Surprinsingly, we are not seeing such a pattern in our datasets.

fast-track the integration of such high priority and severe issues. For instance, according to Eclipse guidelines for filing issue reports, priority value P1 is used for serious issues and specifies that the existence of a P1 issue should prevent a release from shipping [32]. Hence, it is surprising that priority and severity play such a small role in determining the release in which an issue will appear in.

We performed an additional analysis to investigate how integration delays are related to *priority* and *severity* among the studied projects and why they had such little influence in our models. Figure 9 shows the percentage of issues with a given priority (*y-axis*) in a given delay class (*x-axis*). Note that the integration of 36% to 97% of priority P1 addressed issues were delayed for at least one release, whereas the percentages for P2 were 32% to 96%. In ArgoUML, while the majority of priority P1 issues (64%) were integrated in the *next* release, 36% of them were delayed by at least one release. For Firefox, 97% of the P1 issues and 96% of the *blocker* issues were delayed by at least one release. Finally, for Eclipse, 57% of P1 issues and 49% of blocker issues were delayed by at least one release. Hence, our data shows that, in the context of issue integration, *priority* and *severity* have little influence on integration delay.

The integrator workload plays a major role in estimating the integration delay in the three studied projects. Priority and severity have little influence in estimating integration delay. Indeed, 36% to 97% of top priority (P1) addressed issues were delayed by at least one release.

IV. Threats to Validity

Interval validity. The internal threat to validity is concerned with the ability to draw conclusions from the relation between the independent and dependent variables. The main threat in this regard is the representativeness of the data. Although Firefox and Eclipse report the list of addressed issues in their release notes, we do not know how complete this list truly is. Moreover, the method that we use to map the addressed issues to releases in ArgoUML is based on the *target_milestone* which may contain noise. Nonetheless, our Firefox and Eclipse results are based on addressed issues that we are sure have been integrated in the denoted releases.

We segmented the dependent variable into four classes: *next*, *after-1*, *after-2*, and *after-3-or-more*. Although, we found it to be a reasonable classification, a different classification may yield different results. Also, the attributes that we considered in our prediction models are not exhaustive. The addition of other attributes would likely improve model performance. Nonetheless, our models performed well compared to random prediction and Zero-R models with the current set of attributes and dependent variable segmentation.

External Validity. External threats are concerned with our ability to generalize our results. In our work, we investigated three open source projects. Although the projects that we considered in our study are of different sizes and domains, and prescribing to different release policies, our findings may not generalize to other systems. Replication of this work in a large set of systems is required in order to arrive at more general conclusions.

V. Related Work

Estimating the effort and time required to address an issue has become an important project planning activity. To assist developers and project managers in this regard, several studies have proposed different approaches to estimate effort and time required to triage and to address issues [1–7, 33]. In each of the following subsections, we describe prior work about triaging, addressing, and integrating issues.

A. Triaging Issues

Triaging issues is the process of deciding which issues have to be addressed, and assigning the appropriate developer to them [11]. This decision depends of several factors, such as the impact of the issue on the software, or how much effort is required to address the issue. Projects usually receive a high number of issue reports. Issue reports come from a diverse audience that is usually larger than the developer team. Hence, effective triaging of issue reports is an important means of keeping up with user demands.

Hooimeijer and Weimer [33] built a model to classify whether or not an issue report will be "cheap" or "expensive" to triage by measuring the quality of the report. Based on their findings, the authors state that the effort required to maintain a software system could be reduced by filtering out reports that are "expensive" to triage. Saha *et al.* [34] studied long lived issues, *i.e.*, issues that were not addressed for more than one year. They found that the time to assign a developer and address such issues is approximately two years. Our work complements these prior studies by investigating the time to integrate issues once they are addressed.

B. Addressing Issues

Once an issue is properly triaged, the assigned developer starts to address it. To estimate the time required to address issues, some approaches used the similarity of an issue to existing issues [6, 7], while others built prediction models using different machine learning techniques [2, 3, 5, 23]. Kim and Whitehead [4] computed the time taken to address issues in ArgoUML and PostgreSQL. They found that the median issue-fix time is about 200 days. Guo *et al.* [10] used logistic regression model to predict the probability that an new issue will be fixed. The authors trained the model on Windows Vista issues and achieved a precision of 0.68 and recall of 0.64 when predicting Windows 7 issue reports. These approaches focus on estimating the time required to address an issue. In our study, however, we investigated in which release an addressed issue will be integrated.

Recent empirical studies assess the relationship between the attributes used to build models for estimating bug fix time. Bhattacharya and Neamtiu [35] performed univariate and multivariate regression analyses to capture the significance of four features in issue reports. Their results indicate that more independent variables are required to build better prediction models. Herraiz *et al.* [27] studied the mean time to close issues reported in Eclipse, and how the severity and priority levels of the issues affect this time. In their study, the authors used one way analysis of variance to group the different priority and severity levels used in Eclipse. Based on their result, the authors suggest to reduce the severity and priority options to three levels. Zhang *et al.* [36] investigated the delays incurred by developers in the issue addressing process. To do such analyses, they extract the beginning and ending time of an issue addressing activity from interaction logs. Using the collected information they analyzed delays in the issue addressing process. In their analysis, they investigated the impact of three dimensions related to issues: issue reports, source code involved in the issue, and code changes that are required to address the bug. They found that metrics such as severity, operating system, description of the issue, and comments are likely to impact the delays in starting to address the issue and changing the status to RESOLVED. Similar to Zhang *et al.* [36], we used attributes related to issue reports to build prediction models in order to understand which attributes play an important role in the prediction. In addition, we investigate why severity and priority levels are not relevant to distinguish issue reports that are addressed and integrated in a release prior to others.

C. Integrating Issues

Jiang *et al.* [8] studied attributes that could determine the acceptance and integration of a patch into the Linux kernel. A patch is a record of changes that is applied to a software system to address an issue. To identify such attributes, the authors built decision tree models and conducted top node analysis. Among the attributes studied, developer experience, patch maturity, and prior subsystem are found to play a major role in patch acceptance and integration time. Similar to Jiang *et al.* [8], we also investigate the integration of addressed issues. However, we focus on the integration delay of issues that have been addressed.

VI. Conclusion and Future Works

Once an issue is addressed, what users and code contributors most care about is when the software is going to reflect the addressed issue, *i.e.*, when the integration occurs. However, we observed that the integration of several addressed issues was delayed for several releases. In this context, it is not clear why certain addressed issues take longer to be integrated than others. Hence, we performed an empirical study of 20,995 issues from the ArgoUML, Eclipse and Firefox projects. In our study, we:

- found that despite being addressed well before of an upcoming release, 34% to 60% of the addressed issues were delayed by more than one release in ArgoUML and Eclipse. Furthermore, 98% of Firefox issues were delayed by at least one release.
- built models to predict the integration delay of an addressed issue. Our models achieved a weighted average ROC area of at least 0.74. Our models outperform baseline random and Zero-R models.
- computed the variable importance to understand what attributes are the most important in our models of integration delay. The integrator workload is found to be the most important attribute when estimating integration delay. Surprisingly, we found that *Priority* and *severity* have little impact on our models. Indeed, 36% to 97% of priority P1 addressed issues were delayed by at least one release.

Our work provides some initial insight as to why some addressed issues are integrated prior to others. Our results suggest that more work is needed to understand and facilitate the activities of integration teams. For instance, the workload of the integrators is an important indicator of integration delay. It is important to improve the integration step of a release cycle, since the availability of an addressed issue in a release is what users and contributors care most about. Our models could also be used to give an estimation for users and contributors about when an addressed issue will be integrated from the time when it is addressed.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 35–39. [Online]. Available: http://doi.acm.org/10.1145/1117696.1117704

[2] P. Anbalagan and M. Vouk, "On predicting the time taken to correct bug reports in open source projects," in *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, ser. ICSM '09, Sept 2009, pp. 523–526.

[3] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 52–56.

[4] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 173–174. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138027

[5] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, ser. Promise '11. New York, NY, USA: ACM, 2011, pp. 11:1–11:8.

[6] C. Weib, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–.

[7] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1042–1051.

[8] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast?: Case study on the linux kernel," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 101–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487111

[9] Firefox, Bug 883554, http://goo.gl/owzss5. Accessed: 17-04-2014.

[10] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 495–504. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806871

[11] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[12] Bugzilla fields, http://goo.gl/jdbkwd. Accessed: 20-04-2014.

[13] Life cycle of a Bug, http://goo.gl/nbmhaf. Accessed: 20-04-2014.

[14] M. V. Mantyla, F. Khomh, B. Adams, E. Engstrom, and K. Petersen, "On rapid releases and software testing," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 20–29.

[15] ArgoUML, http://argouml.tigris.org/. Accessed: 17-04-2014.

[16] Eclipse, https://www.eclipse.org/. Accessed: 17-04-2014.

[17] Mozilla Firefox, http://goo.gl/qyizd2. Accessed: 17-04-2014.

[18] ArgoUML Release Notes, http://goo.gl/rjggvd. Accessed: 20-04-2014.

[19] Eclipse Release Notes, http://goo.gl/arjj0u. Accessed: 20-04-2014.

[20] Firefox Release Notes, http://goo.gl/x8htzm. Accessed: 20-04-2014.

[21] ArgoUML Release Notes, http://goo.gl/nbdxp7. Accessed: 20-04-2014.

[22] ArgoUML, Bug 4914, http://goo.gl/iok56r. Accessed: 17-04-2014.

[23] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 29–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.25

[24] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 111–120.

[25] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 284–292.

[26] L. Breiman, "Random forests," in *Machine Learning*, ser. Springer Journal no. 10994, 2001, pp. 5–32.

[27] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "Towards a simplification of the bug report form in eclipse," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 145–148. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370786

[28] Random Forest R Package, http://goo.gl/fyyd33. Accessed: 20-04-2014.

[29] Eclipse Commiters, Contributors and Councils, http://goo.gl/q4vzzl. Accessed: 14-07-2014.

[30] Firefox Release Engineering, http://goo.gl/gbvxsk. Accessed: 14-07-2014.

[31] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/567793.567795

[32] Eclipse How to use BugZilla, http://goo.gl/xur43g. Accessed: 17-04-2014.

[33] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 34–43. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321639

[34] R. Saha, S. Khurshid, and D. Perry, "An empirical study of long lived bugs," in *2014 Software Evolution Week - IEEE Conference onSoftware Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, Feb 2014, pp. 144–153.

[35] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: Can we do better?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 207–210. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985472

[36] F. Zhang, F. Khomh, Y. Zou, and A. Hassan, "An empirical study on factors impacting bug fixing time," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, Oct 2012, pp. 225–234.