October 2021

Integrating Bug Deduplication in Software Development and Testing

Carlos Arguelles

Megha Jindal

Babu Prasad Elumalai

Subham Mishra

Richa Gupta

Integrating Bug Deduplication in Software Development and Testing

ABSTRACT

A bug deduplicator identifies independently discovered bugs that have the same underlying cause. Deduplication of bugs reduces toil for the software team by reducing the number of bugs that developers need to examine. However, if a bug deduplicator incorrectly classifies a bug as a duplicate, human developers might ignore the bug, allowing it to escape to production. A tradeoff exists between toil reduction and risk tolerance. This disclosure describes techniques that enable a software team to trade off the effort to remove bugs (e.g., auto-close bugs so that humans save toil and time) against the risk of errors in a bug deduplicator. Custom settings and a confidence level that a bug is a duplicate are used to determine whether to log a particular bug, to log it with comments, etc. The techniques enable the embedding of a bug deduplicator at suitable locations within a software development toolchain. The performance of the bug deduplicator can be fine-tuned in real-time by an analysis of its true negative and false positive metrics.

KEYWORDS

- Software bug
- Test execution
- Software testing
- Test log repository
- Bug deduplication
- Bug repository
- Software development
- Development toolchain
- Developer effort

BACKGROUND

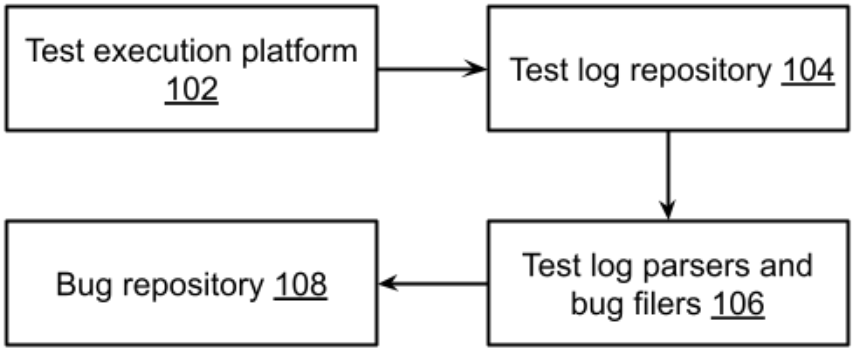

Fig. 1: Software testing stack

A software testing stack comprises a number of subsystems. An example is shown in Fig. 1. A test execution platform (102) runs unit and integration tests. After the tests are run, test logs are stored in a test log repository (104). Test log parsers and bug filers (106) analyze the logs, parse for failures, and log a bug per failure. The bug is stored in a bug repository (108). Storing a bug in a repository is also known as logging a bug.

A bug deduplicator (not shown) identifies independently discovered bugs that likely have the same underlying cause. The bug deduplicator also outputs a metric that reflects the confidence that the bug is a true duplicate. The confidence metric can be used to deduplicate a class of bugs and represent bugs in the class as a single entity. A bug deduplicator can be located in various places in the stack of Fig. 1.

Accurate deduplication of bugs enables increased productivity, or reduced toil, for a software development team, by reducing the number of bugs that developers have to examine and fix. However, there is a risk that the bug deduplicator incorrectly classifies a new bug as a duplicate of an existing bug. This can cause the bug to appear in production code. A tradeoff exists between toil reduction and risk tolerance.

Position of bug deduplicator	Toil reduction	Risk tolerance
At the bug filer (auto-closing the bug at the level of the bug-filer)	High	Low
At the bug repository (adding a comment to the bug at the bug-repository level for a human to make a final decision)	Low	High

Table 1: Tradeoff between toil reduction and risk tolerance

Table 1 illustrates an example tradeoff between toil reduction and risk tolerance. If the bug deduplicator is placed at the bug filer, e.g., the bug filer calls the bug deduplicator prior to filing a bug, then the toil reduction is high, since a bug is not filed (bug entry not created in the repository) if the bug is found to be a duplicate of an existing bug. Developers will not spend time looking at the bug because it doesn't exist in the repository. For the same reason, e.g., the lack of human oversight, the risk tolerance is low (risk is high) that an incorrectly deduplicated bug makes it to production code.

If the bug deduplicator runs against the bug repository, e.g., after the bug is filed, then it is not automatically closed. Rather, a comment is added pointing it to an existing bug that is a likely duplicate. The developer gets a notification, analyzes the bug, and closes it as appropriate. Since a human developer is involved, the toil reduction is low (although toil-reducing information relating to the bug and its possible duplicates is made available). Owing to the involvement of the developer in deciding if the bug is a true duplicate, the risk that the bug makes it to production is low (the risk tolerance is high).

DESCRIPTION

This disclosure describes techniques that enable a software development team to trade off toil reduction against risk tolerance while using a bug deduplication engine. Further, the techniques enable the tuning of the actions of the bug deduplicator.

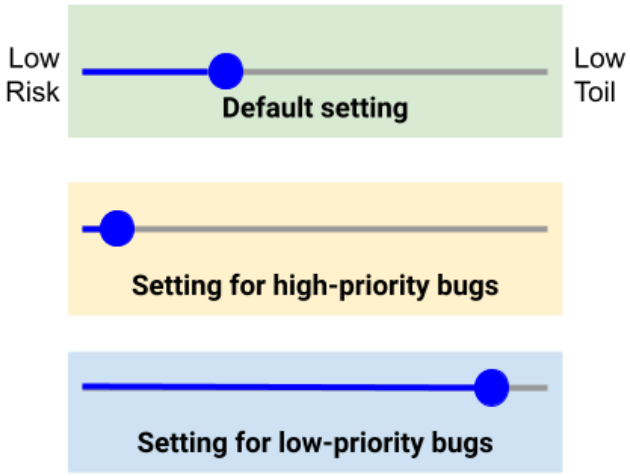

Fig. 2: Sliders to trade off risk reduction against toil reduction

Per the techniques, illustrated in Fig. 2, a customizable set of sliders is provided such that the software team can set an appropriate tradeoff between risk reduction and toil reduction. Bug priority can be taken into consideration in setting the tradeoff. For example, high priority bugs (which tend to be fewer in number) can be set to low risk. Conversely, low priority bugs (which tend to be more numerous) can be set to low toil.

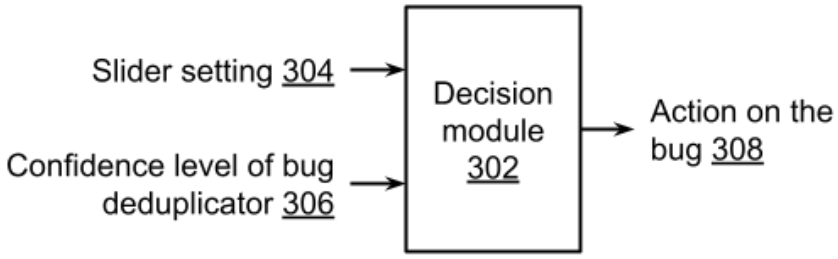

Fig. 3: Automatic actions taken on the bug

As illustrated in Fig. 3, based on the slider setting (304) and the confidence (output by the bug deduplicator) that the bug is a true duplicate (306), a decision module (302) takes one or more actions on the bug (308). Some of the actions taken on a bug include automatically closing the bug (representing it by an existing duplicate bug), adding a comment to the bug for a developer to triage and make the final decision, etc. As explained earlier, auto-closing the bug

reduces or eliminates the human toil of having to sift through duplicates, but has the risk of letting genuine bugs escape to production if the bug deduplication incorrectly identifies a non-duplicate as a duplicate. Adding a comment can reduce, but not eliminate, human toil, since a human does have to make a final decision, but it reduces the risk of letting legitimate problems escape to production because the ultimate decision is up to the human.

For example, if the confidence level is low and the slider indicates a preference for risk reduction, the action is to provide a comment on the bug and leave it open. If the confidence level is high and the slider indicates a preference towards toil reduction, the action is to automatically prevent the bug from being opened, e.g., by preventing the bug from being entered in the bug repository.

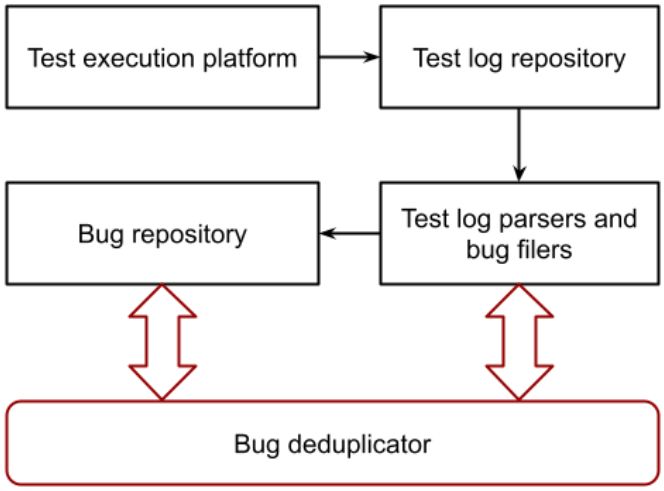

Fig. 4: Positioning a bug deduplicator in the software testing stack

As illustrated in Fig. 4, the techniques enable the positioning of the bug deduplicator (shown in red) in the software testing stack. For example, the test log parser/ bug filer can pass a bug to the bug deduplicator to determine if it is a duplicate. If the bug is a duplicate with high confidence and the slider is set to low toil, then the bug isn't logged and is not entered in the bug repository. As another example, a logged bug in the bug repository can be tested to determine if

it has a duplicate. If it is found to be a duplicate, a comment is generated with a link to the duplicate.

Fine-tuning the actions of the bug deduplicator

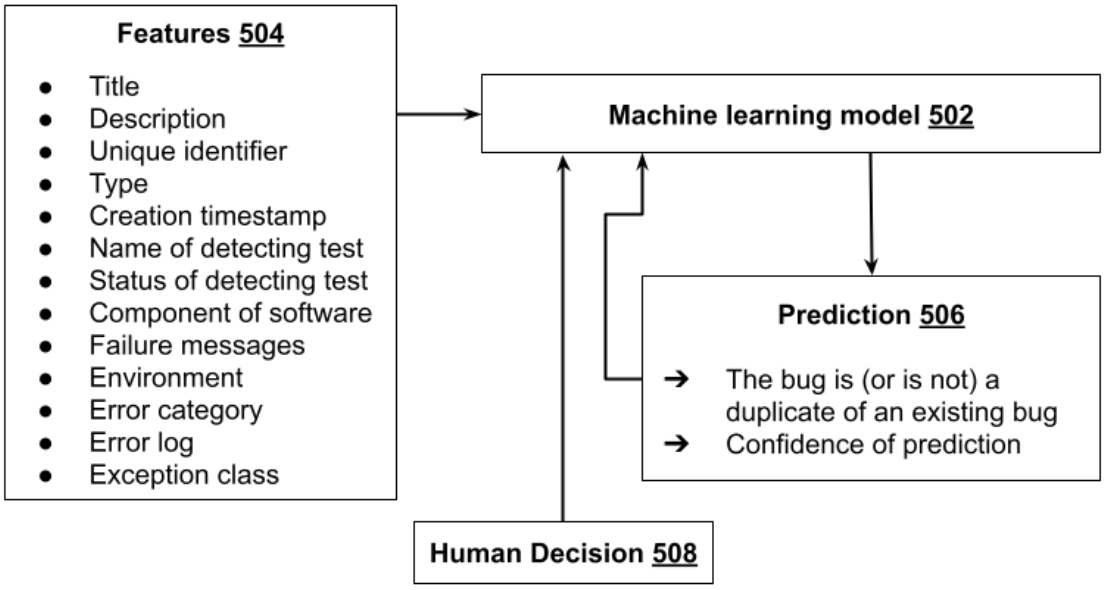

Fig. 5: Fine-tuning the bug deduplicator

As illustrated in Fig. 5, the precision and recall performance of the bug deduplicator can be fine-tuned in real-time as follows. A user newly onboarded to the bug deduplicator is started with a gradual (risk-sensitive) approach, e.g., all detected duplicated bugs are commented to be followed up by a human. Using features of the bug (504), the bug deduplicator, which can be implemented as a machine learning model (502), analyzes bugs in real-time as they're filed and adds the comment before a human sees the bug. Once analyzed, the bug is added to a database along with a prediction (506, duplicate or not). When the bug is resolved by the human, a ground-truth human decision becomes available. The prediction and the human decision (508) can be used to train the bug deduplicator to improve its precision and recall.

	Human decision	ML decision	Comment
True positive (TP)	Duplicate	Duplicate	The ML model correctly identifies a duplicate and saves engineering toil. **Positive impact**
True negative (TN)	Not duplicate	Not duplicate	The ML model correctly identifies lack of duplicates, but there are no savings in toil. **No impact.**
False negative (FN)	Duplicate	Not duplicate	A real duplicate is missed, e.g., a missed opportunity to reduce toil, but not worse sans bug deduplicator. **No impact.**
False positive (FP)	Not duplicate	duplicate	The ML model missed a true duplicate, letting it escape to production. **Negative impact.**

Table 2: Four combinations of human decision and ML prediction

As illustrated in Table 2, there are four combinations of the (human decision, ML prediction) pair. Of the four combinations, true positives (TP) represent the reduction in toil provided by the bug deduplicator (positive impact), and false positives (FP) represent cases where a real bug can potentially escape to production (negative impact). Table 2 aligns with the earlier described tradeoff between reducing toil and reducing risk. (The other two combinations in Table 2, true negatives and false negatives, have neither positive nor negative impact on toil reduction or risk mitigation.)

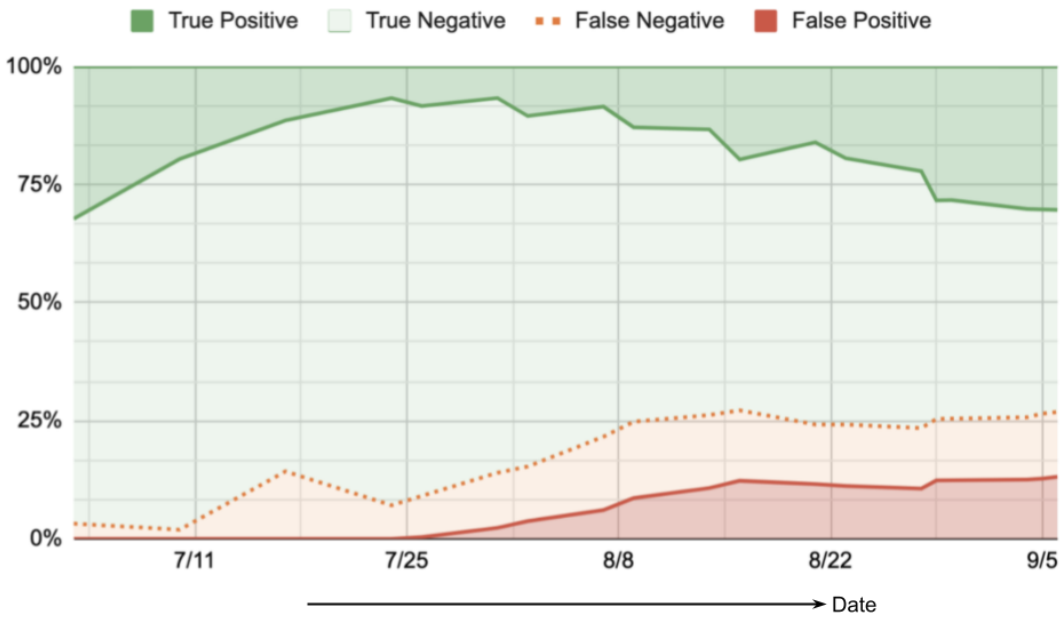

Fig. 6: TP, TN, FP, FN performance with time

Based on the observed, periodic (e.g., daily) performance of the bug deduplicator (in terms of the four parameters TP, TN, FP, FN), a graph, such as the one illustrated in Fig. 6, can be surfaced to the user. Users can inspect the graph to understand how much risk they are taking and how much potential savings in toil they may gain.

For example, the user can be started off with the default slider setting (Fig. 2) at a position of low risk, e.g., bugs are rarely (or never) auto-closed; rather, they are only commented upon for final resolution by a human. If the user moves the slider towards ‘low toil,’ a gradual ramp-down in toil (ramp-up in risk) procedure can be started that works as follows:

- Gather TP/TN/FP/FN metrics for a predetermined number of days.
- Present the TP/TN/FP/FN metrics to the user for inspection, e.g., in the form of Fig. 6. The user can choose to revert to low risk if the metrics aren’t to their satisfaction, in which case the slider is moved back to the previous setting. Alternatively, the user can choose to continue, in which case the bug deduplicator enters a loop, as follows.

    - Of all predicted duplicates, a certain fraction, e.g., 80%, is commented upon and the remaining 20% auto-closed. The TP/TN/FP/FN metrics from the 80% are used to continue assessing the bug-deduplicator performance. Run for a predetermined number of days.
    - If the user approves a greater toil reduction, a lower fraction, e.g., 60%, of the bugs are commented upon and the remaining 40% auto-closed. Repeat for a predetermined number of days.
    - If the user approves a still greater toil reduction, an even lower fraction, e.g., 40%, of the bugs are commented upon and the remaining 60% auto-closed. Repeat for a predetermined number of days.
    - If the user approves a still greater toil reduction, an even lower fraction, e.g., 20%, of the bugs are commented upon and the remaining 80% auto-closed.
- At this point, the bug-deduplicator has gradually ramped up nearly fully to auto-closing bugs while mitigating risk. At any time, if the TP/TN/FP/FN metrics fall below acceptability, the bug deduplicator can be configured to automatically stop and revert to a position of lower risk (fewer auto-closings and more commenting for final resolution by human).

From a user interface perspective, a tab can be surfaced in the bug repository that includes insights that indicate the bugs that are likely to be duplicates, sorted by confidence level. This aids the bug triager to determine that a bug is a duplicate and quickly resolve the bug, e.g., by marking it as a duplicate of the original bug. Additionally, since each bug includes metadata indicating its state of being a duplicate, a bug master can create a search that shows all bugs identified as likely duplicates.

In this manner, the techniques of this disclosure enable the embedding of a bug deduplicator in one or more suitable locations in a software development toolchain. A set of sliders (or similar user interface elements) is provided that enables the owner of a product module to choose a gradient between toil reduction and risk reduction. Optionally, multiple sliders can be used based on bug priority. A combination of the slider setting (which indicates the gradient between toil and risk) and the confidence that a bug is a duplicate is used to decide whether to log the bug after analyzing test failure results, to log it with comments, etc. The bug repository is visually enhanced to surface a tab that shows duplicates, sorted by confidence. A one-click mode is provided within the tab that enables a bug triager to resolve a bug, e.g., with automatically pre-populated fields for a duplicate bug. Views can be created in the bug repository to display duplicate bugs grouped in clusters, for quicker batch resolution by a bug master. The performance of the bug deduplicator can be fine-tuned in real-time by an analysis of its true negative and false positive metrics.

CONCLUSION

This disclosure describes techniques that enable a software team to trade off the effort to remove bugs (e.g., auto-close bugs so that humans save toil and time) against the risk of errors in a bug deduplicator. Custom settings and a confidence level that a bug is a duplicate are used to determine whether to log a particular bug, to log it with comments, etc. The techniques enable the embedding of a bug deduplicator at suitable locations within a software development toolchain. The performance of the bug deduplicator can be fine-tuned in real-time by an analysis of its true negative and false positive metrics.