Technical Disclosure Commons

Defensive Publications Series

December 2020

Use of Ad-hoc Metrics to Identify Code Revision That Introduced Anomalies

Noah A. Brubaker

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation

Brubaker, Noah A., "Use of Ad-hoc Metrics to Identify Code Revision That Introduced Anomalies", Technical Disclosure Commons, (December 02, 2020)
https://www.tdcommons.org/dpubs_series/3848

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Use of Ad-hoc Metrics to Identify Code Revision That Introduced Anomalies

ABSTRACT

Large-scale software codebases are updated frequently by a large number of developers, leading to a large number of code revisions. It is difficult to identify the exact code revision at which an undesirable behavior (bug or culprit) was introduced, since there may be numerous subsequent changes to the codebase prior to detection of the undesirable behavior. This disclosure describes a general-purpose culprit finder that streamlines and automates the detection of a behavioral change to software code. In particular, the described techniques enable quick determination of the exact code revision that introduced a bug. An ad-hoc signal is defined that numerically represents a behavior of the software. Anomalies in the ad-hoc signal that arise between consecutive software revisions are indicative of the presence of a bug. An iterative N-ary search is performed by dividing the code revisions into N parts and testing the corresponding code revisions in parallel until the search interval includes just the one revision where the bug was first introduced.

KEYWORDS

- Software debugging
- Culprit finding
- Anomaly detection
- N-ary search
- Software development
- Code revision
- Execution log

BACKGROUND

Software code development is typically done by teams of programmers, with subsets of individuals working on different modules and independently making changes to the codebase and checking in the updated code. It is not uncommon for undesirable software behaviors (also

known as bugs, anomalies, or regressions) to be introduced during such a development process. For an active codebase in the process of simultaneous modification by several programmers, there may be numerous changes between the revision in which a problem is introduced and the revision in which the problem is first detected.

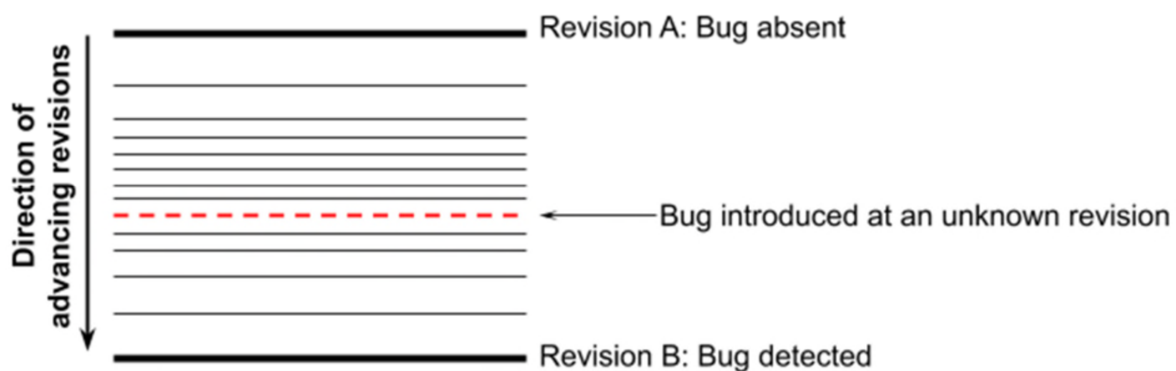

Fig. 1: Introduction of a bug at an unknown revision

Fig. 1 illustrates the introduction of a bug at an unknown software revision. Revision A is confirmed as not having a bug. Revision B, which is more recent than revision A, is where the bug is detected. The bug is determined to have been introduced at an unknown revision between revisions A and B.

It is difficult to identify exactly the revision at which the bug was introduced, or equivalently, the change that caused the undesirable software behavior. Although there are checkpoints where the software is known to be good (revision A), and checkpoints where the software is known to be broken (revision B), there can be many revisions between the checkpoints where the bug might have been introduced.

Culprit finding (CF) is a procedure used to discover the specific revision that introduced the undesirable behavior. Although there are software tools, known as culprit finders, that somewhat automate the culprit finding procedure, culprit finding remains a tedious process.

Also, culprit finders are generally only run when the undesirable behavior introduced by a change is not caught by pre-submit testing.

There are two possibilities for why pre-submit testing doesn't detect bugs - either an existing pre-submit test that could have detected the bug was not run, or no pre-submit test exists that could have detected the bug. Many culprit finders merely assume that an existing pre-submit test wasn't run. These culprit finders search for a test amongst the intermediate revisions that flips from pass to fail, possibly employing a search strategy that divides the search interval into two or more parts.

However, there are situations where no suitable pre-submit test exists, such that running the culprit finder (possibly multiple iterations) fails to identify the revision that introduced the bug. A bug introduced by a bad-but-passing revision requires significant engineering effort to detect, find root cause, and resolve. The downtime between detection and resolution slows down release at best and causes an impact on production systems (where the code is deployed) at worst.

DESCRIPTION

This disclosure describes a general-purpose culprit finder that streamlines and automates the detection of a behavioral change in software. The described culprit finder does not rely on the existence of pre-submit tests designed to detect particular bugs. Rather, an ad-hoc signal is defined that numerically represents a behavior of the software, decided at the time the culprit finder is invoked. For example, an ad-hoc signal can include the following: test status, CPU usage, memory usage, difference between the timestamps of two log entries, any internal state of the software under test, etc. An anomaly that arises in the ad-hoc signal between consecutive software revisions, e.g., a large increase in CPU usage, is indicative of the presence of a bug.

To accelerate the determination of the buggy revision, the described culprit finder uses N-ary search as follows. In a first iteration, the search interval (e.g., between revisions A and B in Fig. 1) is divided into *N* parts, and *N* revisions are simultaneously tested for the presence of bugs, across multiple systems as necessary. At the end of the first iteration, the search interval shrinks into one-*N*th of the original search interval (A, B). At the next iteration, the new (reduced) search interval is once again divided into *N* parts, and tests are run on the revisions that lie within the new search interval. The procedure is repeated until the search interval includes just the one revision where the bug was first introduced.

The described culprit finder has four primary components:

Test data ingestor (TDI): The TDI is an extensible pipeline for collecting test and performance data from automated execution of the software at a specific (test) revision. Test data includes items such as pass/fail signal, test logs, remote procedure calls (RPC) logs, etc. Performance data includes such items as CPU usage, memory usage, miscellaneous benchmarks, etc. The TDI acts as a database-like interface for existing tools that already collect this kind of data. The TDI collects an activity log that can be used to propose as-yet unanalyzed data points or benchmarks that are likely to be important. Queries for the proposed data are generated in collaboration with the signal query handler (data-to-signal transform), described below. The anomaly detection control panel (ADCP), also described below, requests these queries as it actively searches for new anomalies.

Signal query handler (QH): The QH receives metrics and logs from TDI and generates an array of numbers (signals) according to a query. These signals are stored as metrics in a table. For example, suppose the pass/fail signal is labeled `status`, CPU usage is labeled `cpu_time` and memory usage is labeled `max_mem`. The user can write a query as “`SELECT status = PASS,`

`cpu_time, max_mem FROM my_tdi`" for the QH to return, upon every test execution, a signal array of the form `{status: 1.0, cpu_time: 500.0, max_mem: 13.2}`.

Anomaly detection control panel (ADCP): The ADCP provides an extensible suite of anomaly detectors and configuration options to streamline the production of an automated anomaly detection workflow for the suspect signals from QH. The ADCP is used to configure the anomaly detection workflow. A workflow includes *configuring* the test infrastructure (what tests, what hardware, how many shards, etc.); *generating* data k times at $N-1$ revisions by executing the tests (where both k and N are configurable); *transforming* data into signals; *analyzing* whether or not an anomaly is present between two consecutive revisions; etc. The ADCP can be used to set the tests to be run, the number of times to run a test to produce a strong signal of a bug, and other aspects of a test execution strategy. The ADCP is particularly useful because a bug may show up only probabilistically, e.g., once every N runs; by setting the number of runs to a suitably large figure, a strong signal for a bug can be generated. The ADCP can also be used to set a detection procedure to decide whether a bug is present at a certain revision based on the data collected so far. The ADCP can be configured to run continuously to ask for queries from TDI to explore the available signal space; to automatically discover and save queries that produce anomalies; to trigger N-ary search for detected anomalies; etc. The ADCP maintains a database of saved queries and previously detected anomalies, e.g., the aggregated signals for a test run on an affected revision R and its immediate-previous revision R−1.

N-ary searcher, also known as (N-sect): The N-ary searcher accepts a range of revisions and executes the anomaly detection workflow from the ADCP in parallel with a configurable number of worker threads. As explained earlier, the N-ary searcher accelerates the determination of the buggy revision by successively subdividing and testing the revision range.

Example use case: Manual use of the culprit finder to determine a buggy revision

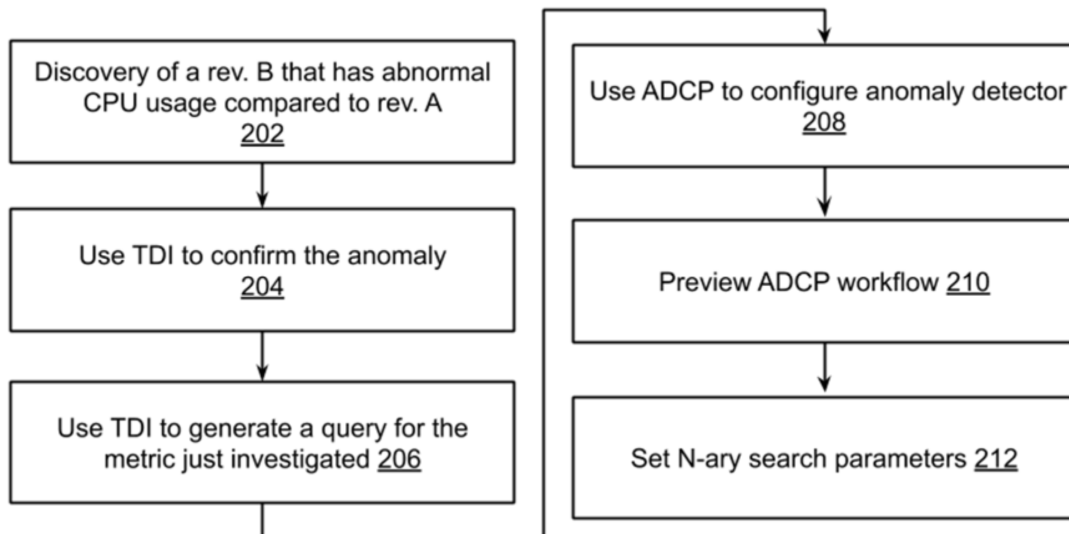

Fig. 2: Workflow using the culprit finder

Fig. 2 illustrates an example of a workflow that a developer can use in order to uncover a buggy revision using the described culprit finder. A developer discovers a certain anomaly, e.g., that the CPU time of a particular test is much greater at revision B than it is at revision A (202). The developer uses the TDI interface to confirm the anomaly (204). The developer uses the TDI to generate a query for the metric just investigated, e.g., CPU time (206).

The developer configures ADCP to detect anomalous CPU time by plugging in the name of the test target and the suspect query (208). To ensure that the anomaly shows up, the developer sets the number of test runs per revision to a sufficiently large number, e.g., 40. The developer uses ADCP to preview the workflow on revisions A and B (210). The developer uses N-sect to configure a 4-ary search (212), which executes the anomaly detection workflow across the revisions range (A, B) in search of the first revision that triggers the anomaly (the culprit).

Upon discovery of the revision that introduced the bug, the culprit finder produces a report that includes the revision number that introduced the bug; whether or not the culprit

revision was validated (e.g., the immediate previous revision doesn't trigger the bug, but the detected revision number does); the size and significance of the anomaly (e.g., the absolute and percentage changes in CPU time, the p-value, or confidence that the anomaly was correctly isolated); etc. The developer adds CPU-time to the saved queries of the ADCP so that it automatically catches CPU time anomalies in the future.

Example use case: Automated use of the culprit finder

In an example, automated execution of the culprit finder can use the following workflow:

- The ADCP asks for a random query from TDI. It gets CPU time as the metric to track.
- The ADCP detects a CPU time anomaly in the last 30 days.
- The ADCP adds CPU time to its list of saved queries.
- The ADCP automatically triggers an N-ary search to find the exact revision where the CPU-time anomaly was first introduced.

Upon automatic detection of the first appearance of the anomaly in a particular revision, the developer that checked in the revision is notified (along with a report of the anomaly). The developer can then take steps to debug the software, rollback the revision, or ignore the anomaly and flag it as "not important." If not flagged, the ADCP continues checking for CPU-time anomalies in future revisions.

In another, and more general, example use case, any software behavior can be represented by a numeric value. Instead of looking for bad data, a developer can use the described culprit finder to find an answer to a question about a new feature that was recently introduced. For example, the developer guesses that the new feature uses a new or updated remote procedure call (RPC) message with a field of a certain type, but searching the logs for such an RPC takes an impractically long time. The developer takes a known snapshot of the RPC

log for a typical test run before the new feature was introduced, and writes a method that examines RPC logs and returns a signal 1.0 if any RPC messages had the expected change, and a signal 0.0 otherwise. They add the method to TDI and use the new signal to find the revision where the signal changes from 0.0 to 1.0 as the revision that likely introduced the new feature. In this manner, the described culprit tool can also be used to enable developers to learn something about the behavior of the software without manually digging through logs.

In this manner, engineering time and costs to detect, to find root cause, and to resolve bugs can be substantially reduced. Production code is kept free of bugs and performance degradations that can impact customer experience. The described culprit finder can be a component or tool in a code-development package, e.g., a revision-control product, a source-control product, an integrated development environment (IDE), etc.

CONCLUSION

This disclosure describes a general-purpose culprit finder that streamlines and automates the association of a behavioral change with a code change in a software system. In particular, the described techniques enable quick determination of the exact code revision that introduced a bug. An ad-hoc signal is defined that numerically represents a behavior of the software. Anomalies in the ad-hoc signal that arise between consecutive software revisions are indicative of the presence of a bug. An iterative N-ary search is performed by dividing the code revisions into N parts and testing the corresponding code revisions in parallel until the search interval includes just the one revision where the bug was first introduced.