AI Software Bugs Segregation System: DSF

Ooi Mei Chen; Liu Keping

mei.chen.ooi@intel.com; keping.liu@intel.com

Abstract

Software bugs fixing always will be expensive. A complex software solution usually consists of multiple sub-components or combinations of multiple software solutions. The more software sub-components integrated; the more Software bugs occur. Most of the time it is exceedingly difficult to identify which sub-component breaks the end-to-end functionality.

Data-Search-Fix (DSF) is an AI software bugs segregation system that search and analyze the historical data in a smart way. Recommend a solution to narrow down debugging scope and reduce debugging cost. DSF consists of 3 major blocks - "Data Collection", "Search Algorithm", and "Fix's recommendation" on Dot net and MSSQL (Microsoft SQL). The data and search algorithm are designed in a form that can be easily swapped with other data or algorithms to fulfill the specific needs for different software communities.

This paper will explain DSF software bugs segregation system detail end-to-end flow with a deep dive example. With the implementation of DSF shows productivity improvement by demonstrating quick turnaround time to identify the issue owner and root cause the issue.

Biography

Ooi Mei Chen has been in software engineering for over 12 years and has held many roles spanning code development, design, and project management. She currently serves as a senior System Software Quality Engineer at Intel Corporation based in Penang, Malaysia. She holds a Degree in Computer Science from University Tunku Abdul Rahman, Malaysia.

Liu Keping is a Technical Leader in Software Quality Assurance at Intel Corporation based in Shanghai, China. She is a certified CMMI assessor, ISO internal assessor, ASPICE internal assessor, CSQE (Certified Software Quality Engineer), and gained 6 Sigma Orange Belt and CPMP certification since 2009. She holds a master's degree in computer science and technology from Central South University in China.

1 What is the Problem?

Software development is a creative work based on a large amount of people's knowledge and skills. If people do not catch their mistakes during code implementation, it will lead to bugs in the products. And software bug fixing will always be expensive. A complex software solution usually consists of multiple sub-components or combinations of multiple software solutions. The more software sub-components integrated; the more Software bugs occur. Most of the time it is exceedingly difficult to identify which sub-component broke the end-to-end functionality.

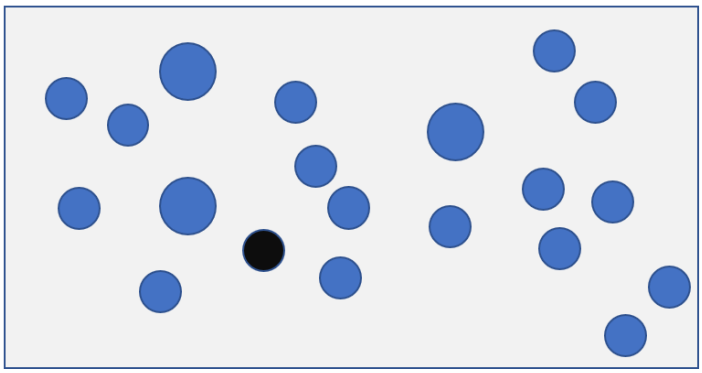

Figure-1 Where is the bug?

The cost spent on bug fixing is increasing as the development phase evolved, straightly burst out after shipping to customer. See Figure-2 the research result from Jones, Capers in "Applied Software Measurement: Global Analysis of Productivity and Quality."

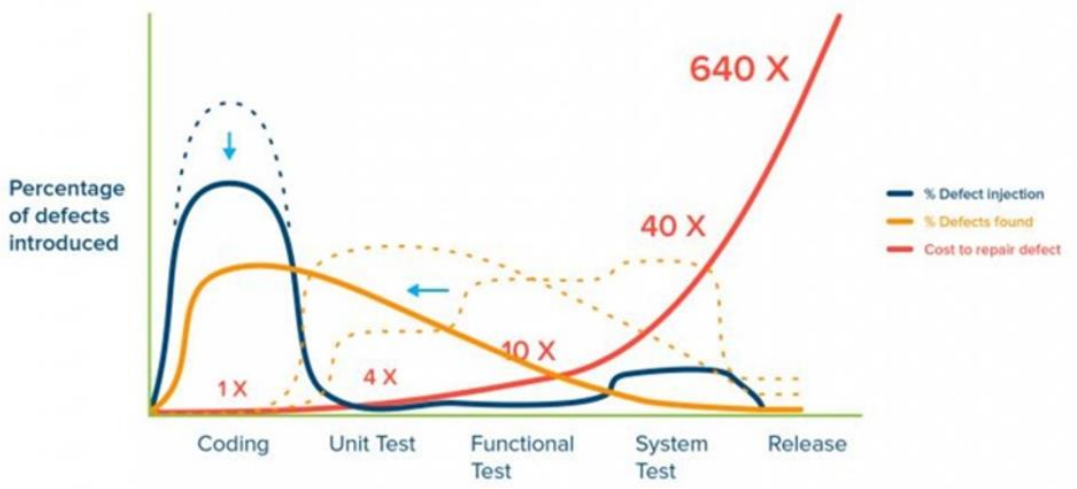

Figure-2 Bug Fixing vs. Cost

Thus, an effective approach to find the root cause of a bug in a quick manner and finds the right injection point which really caused the problem is extremely important. But it is always difficult!

2 Why is this difficult?

Unlike software development which intends to construct a system according to a well-known specifically defined product requirement, bugs fixing is working on an opposite way that aims to find the root cause

and fix potential hidden problems that have occurred in the code database. It is unpredictable and not as planned.

Complex dependency between sub-components is a main factor. A typical example is that: the subcomponents passed the component level unit test but failed in the end-to-end integration test. It is extremely hard to identify which component caused the real problem. This happened frequently, especially in large collaborative product development. Though the team has identified component dependencies in the beginning, with new features/function added in, new dependencies are bought into the system which are impossible to uncover thoroughly. Even if all the dependencies were obvious, it still takes time to figure out the exact “suspect.”

Many software development engineers lack of the overall end-to-end system knowledge is another main factor. When bugs occur, the sub-component development team does not have the end-to-end knowledge to debug the issue.

Some other factors which contribute to the difficulty include

- Critical quality assurance process steps are skipped occasionally when packaging in release phase. Multiple check-in requests submitted at the same time within a short duration. Some files are skipped or overridden which is easy to overlook.
- Responsibilities are not clear which requires skillful communication capability to get the debug work to move ahead. For example, there is a disagreement between the subcomponent teams on where the bug resides and no one willing to dig it out first.

3 Approach

The Data-Search-Fix (DSF) is an AI software bugs segregation system that searches and analyzes the historical data in a smart way by recommend a solution to narrow down the debugging scope and reduce the debugging cost. The DSF consists of 3 major blocks - "Data Collection”, “Search Algorithm”, and “Fix recommendations” on .NET and MSSQL (Microsoft SQL). The data and search algorithm are designed in a form that can be easily swapped with other data or algorithms to fulfill the specific needs for different software communities. The main goal is not to eliminate the debugging work but to identify the potential sub-components and suggest the potential root cause and debug direction with the statistic of the bugs and root cause occurrence.

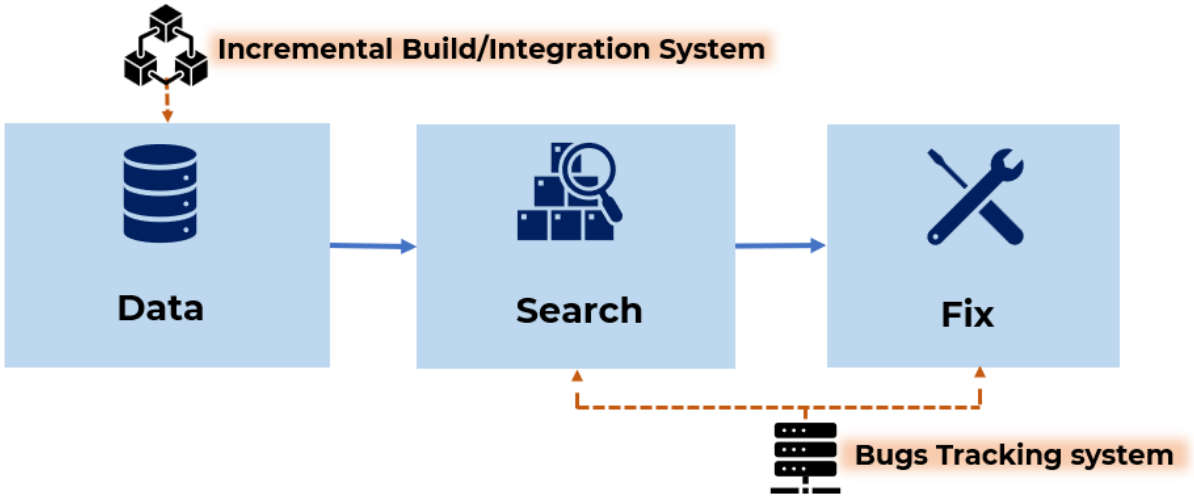

Figure-3 Data-Search-Fix (DSF) AI software bugs segregation system

3.1 Data

Collecting high quality data base on integrated software behaviors is the foundation of building effective data searching and analysis. The DSF has demonstrated an example to store the data in MSSQL and sort the data based on the sub-component functionality complexity, keyword, and injection phase. Data collecting and sorting needs to be considered carefully by the DSF designer with integration, build and software development teams to make sure it corresponds to the search algorithm and is flexible and scalable when the data is used in “Search Algorithm” and “Fix Recommendation”.

3.1.1 Data Storing and Sorting

The “Data Collection” block of the DSF determines what types of data need to be collected. Below are the recommended data that can be used for analysis and segregation.

- How many sub-components or solutions have been integrated into the software
- Which sub-components have more code change or new functionality
- Clear software architecture design, incremental build sequence and dependency of each of the sub-components
- Group the component that have the strong dependency
- Consider the historical data like “root cause,” “injection phase” and other Root Cause Analysis entries from existing root caused bugs captured in bugs tracking system

One more thing to remember is standardize the bug fields like “key word,” “issue log” and “error log” which will benefit the search algorithms. The reason these data are collected and how they will be used will be explained below in section3.2 and section3.3..

3.2 Search

There are many search algorithms that can be used for bug segregation, such as binary search, linear search, interval search, interpolation search etc. Considering the data structure being searched and prior knowledge about the data, DSF decided to use the binary recursive search algorithm to narrow down the sub-components that have the failure test case.

3.2.1 Identify most possibly went-wrong sub-component with Binary Search

To isolate sub-component which breaks the end-to-end functionality, the DSF when supported by an incremental build system also automates the sub-component segregation and end-to-end integration test together with the selected search algorithm. The example below explained how DSF uses the binary recursive search algorithm on a complex software solution that has 20 sub-components about 4 times, and finally narrowed down the issue most possibly went-wrong in component11 to 15.

- **Arrange and find the Mid**

Based on the data collected in 3.1.1, “Which sub-components have more code change or new functionality” this data will be used to sort and define which component is less or more complicated. Secondly, “How many sub-components have been integrated into the software” this data will be used when DSF search divides the total components into two halves and finding the middle index “Mid.” This example we choose to measure the complexity by line of code change in each sub-component. Then the arrangement of a less complicated component on the left and a more complicated component on the right. In the example, in total, 20 components and component10 is the middle index. Note that the component number is not fixed but determined by actual project complexity.

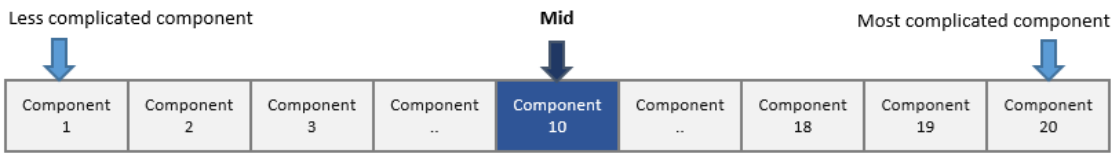

Figure-4 Arrange and find the Mid

- **Pinpoint the bug via integration test**

To isolate sub-component which broke the end-to-end functionality, the DSF when supported by an incremental build system, also automates the sub-component segregation and end-to-end integration test together with the selected search algorithm. The binary recursive search will start with the first half components from 1 to “Mid” (1 to 10 in this case). Continuously integrate and test to see if the bug existed in the first half.

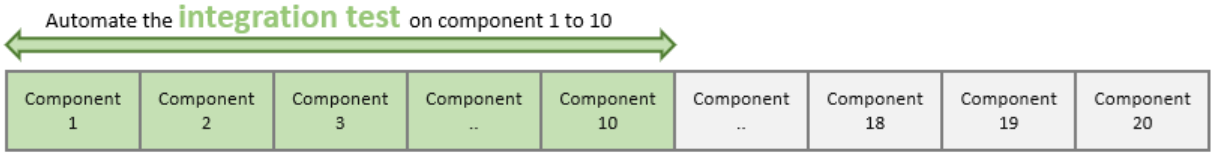

Figure-5 Automate the integration test

If the bug was not observed in component1 to component10, the binary recursive search will proceed to execute the integration and automate testing on the second half (11 to 20 in this case).

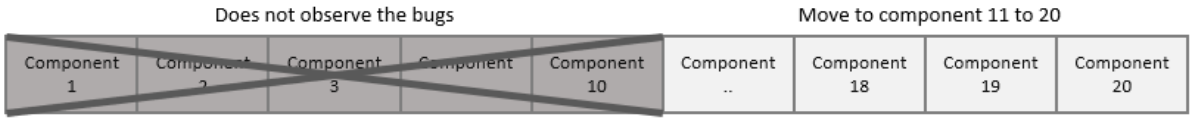

Figure-6 Exclude and move to next block if no bug identified

- **Identify 5 most possibly went-wrong sub-component**

Same as above, the components 11 to 20 are arranged with less complicated components at left and more complicated components at right. By catching the test case failure, finally we observed the bug potential happen in components 11 to 15 - the 5 most possibly went-wrong sub-components. For cross check purposes, the integration and validation can be extended to execute on the second half of components (16 to 20 in this case). This search algorithm can continue until the last sub-component or stop at certain number of sub-components based on the goals that the DSF designer wants to achieve,

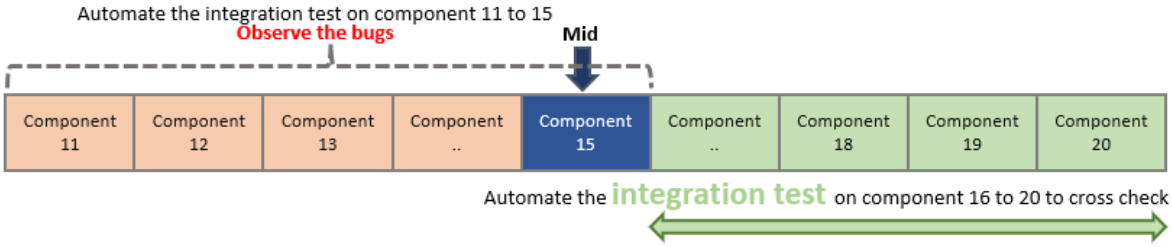

Figure-7 Identify 5 most possibly went-wrong sub-component

```
int binarySearch(int component[], int size) {
  int low = 0;
  int high = size - 1;
  int mid;
  bool found = false;
  int start_component;
  int end_component;

  //Recursive check to look for fail integration test
  while ((low < high) && (found == false))
  {
      //Find the mid component by dividing the component list into half
      mid = (low + high) / 2;

      found = LookForFailIntegrationTest(low, mid);
      if (found == false)
          low = mid;
  }

  if (found == ture)
  {
      start_component = low;
      end_component = mid;
  }

  return -1;
}
```

Figure-8 code snippet for component binary search

According to the code snippet in figure-8, once the found is “true” then it will suggest the most possibly went-wrong sub-components that have the failure test case, the next step is to use the “Fixes Recommendation” function to find the recommended solution based on historical data.

3.3 Fixes Recommendation

To construct the accurate fixing or debugging direction, analysis and filtering needs to be done before the recommendation can be made. Continuing with the example above, for the most possibly went-wrong sub-components, DSF will use the decision tree machine learning algorithm to further process based on historical data like “keyword/issue log/error log,” “root cause,” and “injection phase.”

3.3.1 Analyze the bug’s historical data with Decision Trees in Machine Learning

Decision tree learning is a supervised learning approach usually used in statistics, data mining and machine learning. In this proof of concept, classification trees are used to draw the conclusion with “Yes” or “No” outcome. In the example, DSF will search for the bugs occurrence in the historical data based on the bugs description, similar keyword, injection phase and other Root Cause Analysis entries that been tracked in the existing bugs system.

- **Step 1: Check if the bug happened before**

Based on the most possibly went-wrong sub-components identified, DSF will proceed with the last block” Fixes Recommendation” and reporting. The decision trees start at the (root) node with a question “is the bug happening before?” The answer will lead to branches that hold potential answers “Yes” or “No.”

If there was a similar bug description found in the historical data, the path will proceed to “Yes,” DSF will generate the report and recommend the root cause or debugging direction.

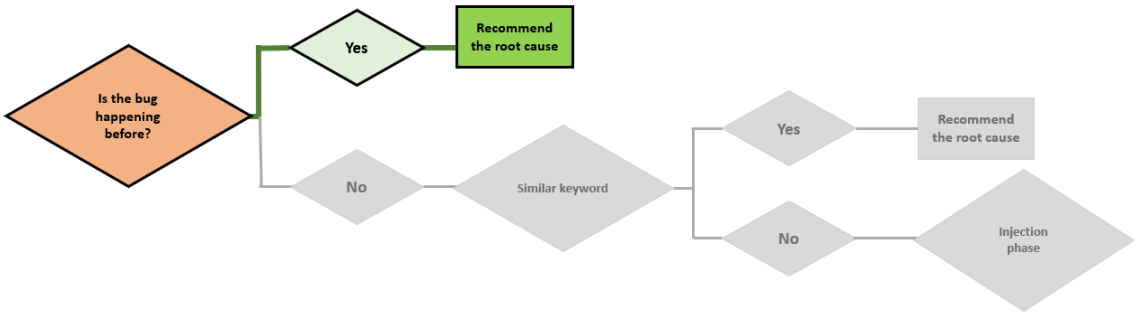

Figure-9 Check if the bug happened before

- **Step 2: Check if there was any similar keyword**

If there was “No” similar bug description, the decision tree will proceed to the next decision criteria (branch) “Similar Keyword.” In this step, the decision tree will use the data collected earlier in section 3.1.1 to filter and find the error log or bug's keyword.

If the answer was “Yes” with similar keyword, DSF will generate the report and recommend the root cause or debugging direction.

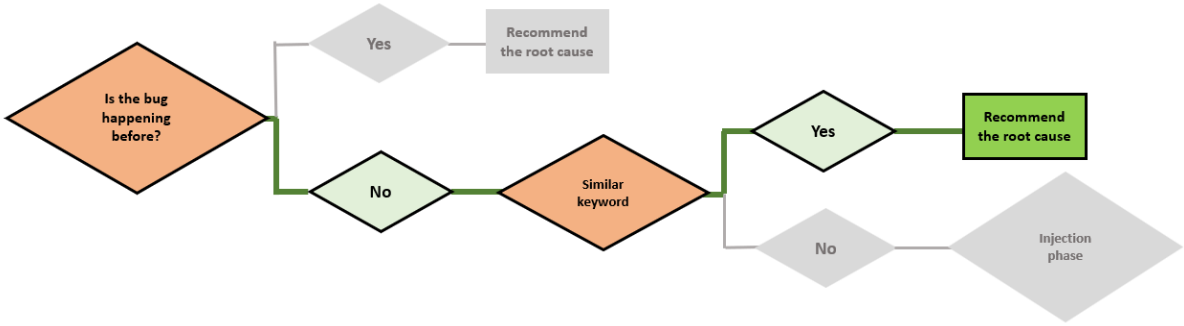

Figure-10 Check if there is any similar keyword

- **Step 3: Check the sub-component injection phase**

If the answer was “No," the decision tree will go on to the next query “injection phase” until the “Yes” reaches or terminal (leaf) node meets checking criteria.

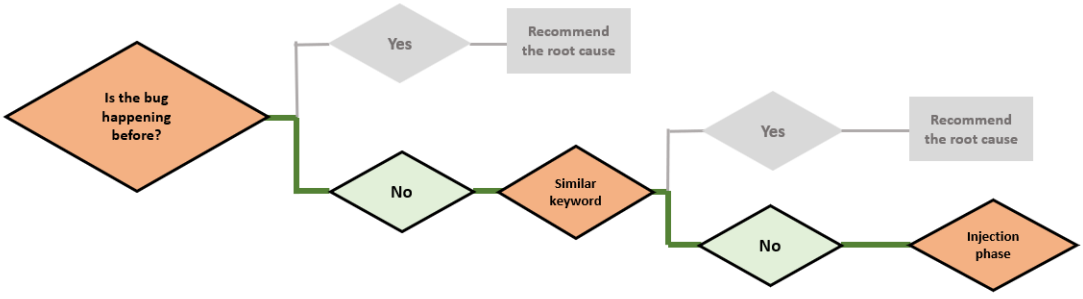

Figure-11 Check the sub-component injection phase

```
DecisionTree* createDecisionTree(vector<string> attributes, vector<vector<string>> issues) {
  if (issues.size() == 0) {
    return nullptr;
  }

  // Find best attribute to split the issues
  int bestAttributeIndex = FindBestAttribute(attributes, issues);

  // Create a new decision tree node with the best attribute
  DecisionTree* node = new DecisionTree();
  node->attribute = attributes[bestAttributeIndex];

  // Recursively create child nodes for each issue with the best root cause
  for (int index = 0; index < issues[0].size(); index++) {
    string value = issues[0][index];
    vector<vector<string>> filteredIssue = filterIssue(issues, bestAttributeIndex, value);
    node->children.push_back(createDecisionTree(attributes, filteredIssue));
  }

  return node;
}
```

Figure-12 code snippet for recommendation decision tree

As stated in figure-12 the decision tree scope can be controlled by the list of the vectors. The decision tree can continue until it recommends the desired number of root causes.

In this example, the final DSF report consists of the most possibly went-wrong sub-components, and the potential root cause and debug direction of each sub-component, with the statistic of the bugs and root cause occurrences.

4 Learning and Challenges

While AI based DSF presents an incredible efficiency in bug segregation, there are some limitations we must face. Learning and solving challenges is a continuous effort to improve the maturity of a system. DSF has gone through the POC (Proof of Concept) and fine tune stage to ensure the end-to-end system is working fine and the result is trustworthy. Below are some of the key elements that need to be considered to make DSF successful:

- Easier coupling and decoupling:
  The software architecture is suggested to be designed with easy module coupling and decoupling. The dependency between sub-components needs to be clearly captured as well.
- Standardized issue description:
  When the testing is done by different validation teams, it is difficult to ensure that issues are described in the same way. It is suggested to create a template to standardize the issue description to make search easier.
- Scalable and flexible criteria:
  The search algorithm and the fixes recommendation decision tree criteria should be designed in a scalable and flexible block so the algorithm and criteria can be easy swapped in and out base on the project need.
- Sufficient training data:
  Make sure the data is meaningful and enough to be analyzed. When training the system, we need lots of data, and long existing software projects work better. Once trained, it can be used for new data analysis. Currently in POC, DSF works better for long existing software projects compared to new software projects - as lack of historical data for training the system. Measures of the successful rate of correct prediction and improvement need to be done from time to time.

5 Conclusions

This paper explained DSF software bugs segregation system detail end-to-end flow with a deep dive example:

- Starting from what data need to be collected
- Then analyze the data by using one of the algorithms, binary search
- And finally report with fixes recommendations

With this implementation, DSF software bugs segregation system shows productivity improvement by demonstrating quick turnaround time to identify the issue owner and issue root cause reports within a complicated software. Lastly, the learning and challenges are also captured for future improvement.

References

- Capers Jones, “Applied Software Measurement: Global Analysis of Productivity and Quality.” Third Edition
- Implementing a Decision Tree from scratch using C++ https://towardsdatascience.com/implementing-a-decision-tree-from-scratch-using-c-57be8377156c
- Decision tree learning https://en.wikipedia.org/wiki/Decision_tree_learning
- Binary search algorithm https://en.wikipedia.org/wiki/Binary_search_algorithm