Fault Localization with a little hand of a Crowd

Christian M. Adriano and Vaibhav P. S. Saini

Donald Bren School of Informatics and Computer Science
University of California, Irvine, CA
{adrianoc, vpsaini}@ics.uci.edu

Abstract — Fault localization is intensively investigated field with a plethora of useful methods and tools. Most of the tools rely on analysis of code and bug repositories by means of automated solutions and metrics. Suspiciousness is one of such metrics demonstrated to localize possible faulty code with acceptable precision. Our aim is to investigate a method in which a crowd could be enlisted to improve fault localization. Such general strategy is not new and has been in place for a number of years in the form of beta tests. Nevertheless both approaches have proven applicability and efficacy, combining them poses interesting questions for feasibility and efficacy. Our approach consists of preparing a system for acceptance test in order to collect information of user satisfaction (ok, not ok) and code execution log traces. Moreover, we defined two metrics for suspiciousness at the level of function calls. In this paper we demonstrate our approach and a proposal for a quantitative evaluation.

Index Terms — Crowd, Suspiciousness, Instrumentation, Exploratory Test, Acceptance Test.

I. INTRODUCTION

Crowdsourcing has been demonstrated viable for many activities such as text authoring [1][2] and for searching and evaluating relevance of documents [3][4]. Crowd development trails the same track and has already shown initiatives for coding [5][6]. Our goal is to investigate the feasibility of crowdsourcing the acceptance test activity in the form of exploratory tests.

Planning for exploratory testing is typically a solitary and trial and error activity, which is very similar to a scientific process. The tester accumulates tacit knowledge by making sense of the actual versus expected behavior of the system. Many activities happen iteratively such as behavior understanding, drawing hypotheses, tracing system outputs and states, etc. Such activities also happen during debugging activities, such as described in studies of types of questions programmers ask while trying to understand a failure [7] or code [8].
Whereas traditional exploratory testing happens in one single mind and comprises a set of iterative and intricate actions, crowd sourcing such exploration requires different people working on the same test cases. Hence, all tacit knowledge must now become explicit, so that the monolithic task of testing can be divided delegated, gathered and evaluated. Exploratory testing now becomes a result of a collective endeavor, from which competing results might emerge.

Testing is one of the few activities in software development that is deeply amenable to parallelization. Besides that, testing can be designed to be modular, which further contributes to parallelization by enabling a decoupling of testing hypothesis.

It is though also true that massive testing has been mostly associated with mechanical brainless activities performed by automated scripts of by unskilled labor force. This state of affairs is perfectly acceptable to certify corrective or adaptive changes made in pre-existing functionalities. One possible explanation might reside on the input space and expected outputs to be well known (for instance, from the historical use of the software in question). Such might not be true for software facing deep changes. For example a legacy system renovation project aimed at migrating Cobol code to Java. Although we still would like to reuse the knowledge accumulated while testing the legacy system, new situations would require a thoughtful and thorough exploration of software. At this point is where exploratory testing comes to play.

Exploratory testing involves the careful analysis of the system behavior during testing and from that draw hypothesis and propose new tests as part of an overarching testing strategy. Translating that to our legacy renovation scenario, the tester might resort on two knowledge baggage - usual side-effects of a migration process and usage history of the system. The former would point to problematic operations or data after changing the programming language of system. A well-known example is keeping the precision in arithmetic calculations when re-implementing the same math in a different language. The latter knowledge baggage aids in highlighting the functionalities and situations in which the system did not behave as expected. For instance, a corner case of calculating moving window averages in which the system reaches days that are holidays in one location but not in another. Exploratory testing must come up with all such subtle and domain specific knowledge, so the user can combine them to draw hypothesis and corresponding test strategies.

Our hypothesis is that the execution trace data can be produced by a crowd of testers and be collected, analyzed and utilized to design new cycles of exploration testing of a system. Moreover, with such approach we can speed up the process of building confidence on software by leveraging on parallelization. By parallelization we mean having different people exploring different functionalities at same time or even the same functionality in different versions of the system.

Another desirable outcome is to leverage on redundancy by having different people trying the same tests and input data while having different opinions in terms of behavior expectation.

We know a lot about bug localization [9], communication through bug reporting [10], bug triage [11][12], and bug delegation [13][14]. Meanwhile, we still know little about how people develop a subjective confidence in a system and how it maps to actual testing in the long run. Nowadays decisions regarding how much testing is enough are mostly tacit and heavily weighted on domain and application experience. For instance, who better knows the code or a feature takes the burden of deciding [13][14]. Relying on code ownership may not be possible for situations in which teams have to deal with legacy systems. This is also the case of development carried out by a crowd. Open source and free software are sometimes suggested as examples of crowd development [15], but such endeavors strong organizational structures which constraint ownership and decision making. Hence, in the absence of ownership information, the team has to rely on qualitative data related to testing and bug fixing tasks. More precisely, the features/requirements and code impacted, as well as their importance to goals defined by user or by the team. The performance of a team to efficiently test a system is thus directly related to the objectivity (accuracy and precision) and relevance of the information about impact and relevance.

In the next sections we describe and define the practical problem faced by software teams in building confidence on software by subjecting it to acceptance test. The problem and research questions are described in section II. In section III we present a brief related work we found relevant and inspiring for the current research. In section IV we describe our approach based on code instrumentation. The system subject to our testing approach is explained in section V. The suspiciousness metrics are presented in section VI. Several technological challenges are discussed in section VII. The proposal for evaluating our approach is detailed in section VIII, which are followed (section IX) by sample solutions obtained by the current implementation. We conclude the paper with an analysis and future work in section X.

II. THE ORACLE PROBLEM FROM A CROWD PERSPECTIVE

A. Problem Framing

We framed the problem by understanding the cycle of choosing test data, delegating test tasks for a crowd, collecting log traces and analyzing results as an exploratory test. By exploratory we signify searching and making decisions on what and how to test. The goal of this process is to incrementally explore different aspects of software while we build confidence on the quality of it. Attributed quality stems from a subjective opinion collected from testers acting like users. In other words, an application with minor harmless defects (e.g. visual misalignment of widgets) may be considered as having acceptable quality. Figure-1 illustrates the crowd testing process with fundamental activities for preparing software to test, preparing the testing tasks for the crowd, sending the tasks, collecting the results, making sense via metrics and running a new cycle of tests.

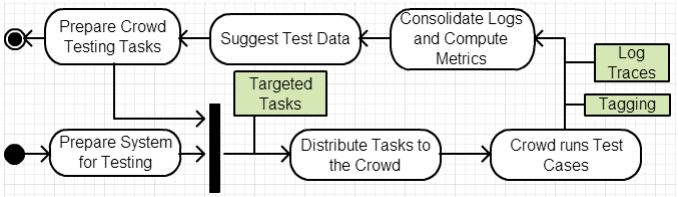

Figure-1. Activities involved in Crowd Testing

1) Prepare System for Testing

Preparing the software before it is made available to the crowd. This involves verifying how the software is structured and deciding where to insert instrumentation.

2) Prepare Crowd Testing Tasks

It involves preparing the content for the testing tasks, i.e., what type of information the crowd must receive to perform tests in a specific system. For instance, it might be necessary to provide specification snippets, example of test inputs and expected outputs.

3) Crowd Runs Test Cases

Running a test comprises two actions. First, selecting and inserting a set of input data. Second, observing the behavior in form of an output and tagging the test execution. Initially we are considering only two types of tags – satisfactory and unsatisfactory. In the future we plan to study the utilization of other types of tags which might help categorize the results in useful ways.

4) Distribute Tasks to the Crowd

Testing tasks would be distributed uniformly, but such decision might be reviewed based on the impact of the skill and knowledge heterogeneity perceived in the crowd.

5) Consolidate Logs and Compute Metrics

It involves collecting all the data produced by each user. After that the system would generate a secondary data necessary to calculate the metrics.

6) Suggest Test Data

Based on the metrics and the consolidated data, we might be able to suggest new data set for testing in order to improve covering of the test input space.

B. Definitions

Below we define the entities that are essential part of our solution

- **Execution Set** = a set of function calls with inputs and outputs. Each execution set is generated by an action of user during acceptance test. For each execution set the user might tag it as satisfactory or unsatisfactory.
- **Execution Trace** = the process of logging the inputs

and outputs happened during an execution set. We log only non-native and non-library calls

- **History of Execution Sets** = the set of all previous execution sets involved in an action of a user during Acceptance Test.
- **Matching process** = Compare the set of function calls from an execution set against the history of calls. The result of this process is a probability attribution for each function call in the execution set to present a failure. Probability attribution would in the form of a set of suspiciousness metrics.
- **Caller** = Method that calls another method
- **Callee** = Method being called by another method

C. *Industry practice*

In order to support our problem we reutilized some results from a case study conducted recently. The case study involved interviews with twenty programmers and testers. The central theme of the interviews concerned the strategy for prioritizing where to invest test effort.

A tester from a military project declared that there is always an intuition where the major bugs should be at each new release or deployment. When inquired whether there were some effort to translate such intuition to a method, the tester answered negatively. This suggests that exploratory testing is an actual though tacit practice, hence constraint by relying on special expertise.

A team leader of an agile team declared that time constraint is the most impacting factor for performing tests during sprints. Deciding what to test and what to fix is really challenging. A partial solution was to invest more heavily in automated testing to speed up the process. The problem is that it leaves the explorative test approach unattended, which is the most effective method to find bugs in new or modified areas of the system. This vision is corroborated by an anecdotal declaration of a quality manager of a major agile software environment vendor:

"We can stop a test session, create a bunch more, and head of in a new direction, which I would say is more fruitful to find bugs in another area [of the software]" – QA Manager at Software Product Summit, 2011.

The interviews also demonstrated that teams from different cultures have distinct approaches to test planning. The first interviewee from the military sector works under a waterfall development process and the team has 10 testers for 7 developers. The amount of testing is justified by the heterogeneity of the deployment environments. For this team, isolating testers and developers is the norm to mitigate bias (testing the features that already work or programming only what will be tested).

Analyzing these two approaches to planning tests and reacting upon new knowledge acquired, we perceive them as complementary in terms of coverage. The former relies on the requirements to decide upon test cases, while the second makes decisions by looking solely at the bug reports. Nevertheless complementary, they aim at completely different needs and do not address the problem raised by our research. The first conceives decision making as data retrieval (concerned with finding things). The second approach conceives decision making as an information space (concerned with understanding things). We need a distinct approach, a search space where a crowd could experiment with different sets of inputs to help us cover the largest possible test input space.

The survey confirmed some of these problems. People prioritize tests based on user satisfaction, as is demonstrated in figure-2. Therefore, the traceability from requirements to failures is a strong input for exploratory testing.

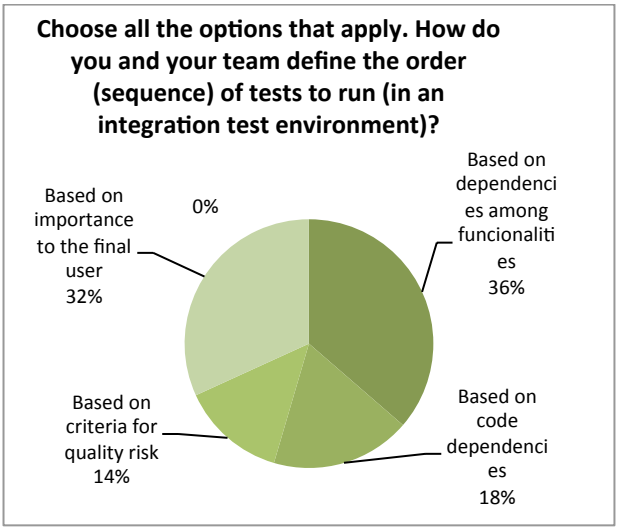

Figure-2 Testing Strategy Reported

The survey also demonstrated that the worst defects were the intermittent (29%, see figure-3), instead of the ones related to Non-Functional Requirements, third party code, or had a large impact on the system. Reproducing a defect implies capturing the same context of the problematic situation. Context is covered by same states and inputs used. Understanding what happens with the application in specific situations involves combining knowledge from multiple sources.

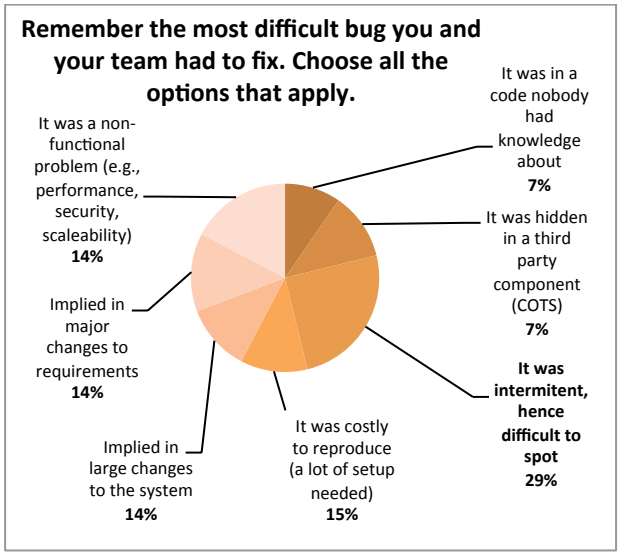

Figure-3 Root-cause of Bugs Hard to Fix

Furthermore, identifying failures involves exploring different options for executing the same functionality and making sense of the expected behavior from documents produced in the requirements and design efforts. Hence, if we were to pursue the hypothesis that exploratory testing can be made by a crowd, questions must be raised first in respect to feasibility of new people performing tests and second, how to collect, analyze and adapt the process with data produced.

D. Research Questions

Based on the problem framing described before, we proposed three research questions to guide our research:

R1. What is the proportion of false negatives and false positives exposed compared against Fault Localization techniques? It is usually argued that programmers are more concerned with false positives, because they don't want to waste their time on false alarms. On the other hand, users are more concerned with hidden bugs, which are related to the false negatives.

R2. In face of non-deterministic and domain specific execution, how much information is needed to successfully localize suspicious code?

R3. What are the current limits imposed by current technology to implement this solution? We chose to experiment initially with Javascript because it is a client side implementation easy to deploy for a crowd to work with.

III. RELATED WORK

A. Invariants

The problem the authors [16] are tried to solve was to analyze logs generated by concurrent systems, which nowadays are very common and not temporal (i.e., it is difficult to say with 100% confidence that each event happened in the exactly same order that it appeared in the log). The authors proposed three different algorithms for that. Our approach to solve such concurrency problem is simpler. It consists of capturing the epoch timestamp for every trace inside the same test execution. Furthermore, we also plan to capture a unique ID of user (e.g. browser ID).

B. Statistical Debugging

The authors [17] categorized test executions as satisfactory and unsatisfactory. For that they applied a statistical approach to isolate different failures and associated them to test executions, which in turn is based on a set of predictor. Predictors consist of factors such as frequency of failure modes and circumstances in which a failure happens. This approach is similar to ours in the categorization aspect, but we are distinct in using the crowd to perform the classification by means of tagging each test execution.

IV. CROWD ORACLE APPROACH

A. Overview

Our approach involves the logging of the trace of each method call. I.e., the trace logs pairs of inputs and outputs of each method. Moreover, if the user deems the execution satisfactory, a tag is attributed to all such pairs. We also aim to keep track of the versions the user was running. Hence, every time a function is created or modified, we store the timestamp. In the future we also aim at storing the dependency among functions.

In order to log the execution traces, we have instrumented the code of the subject system (see section VI). For that we used JQquery AOP. The code for it is publicly available at (https://github.com/saini/coracle).

B. Architecture

1) Data collection

Our solution has a fairly simple architecture. Developers who are interested in using our service will have to include co.js in their html page. The co.js performs three main functions a) instrument the client javascript code b) store execution set in the browser's local storage (window.localstorage) object and c) periodically sends the stored execution sets to server. Server calls made by co.js are RESTful and hence the server maintains no state of the clients. Server stores these execution sets on a NoSQL database. Currently we are using Google cloud storage but later we plan to move to locally hosted Mongodb instances. Analytics module is a standalone application that fetches the execution sets from the database and performs analysis on them. The figure-4 below illustrates the architecture of our solution.

2) The CO.js component

The co.js uses JQuery AOP, a JQuery plugin that adds aspect oriented programing (AOP) features to javascript, to instrument the code during the runtime. Every time an

instrumented javascript function gets called it stores the execution object related to that function. An execution object consists of three attributes a) method name - it's the name of the javascript function that was called - b) input - it's the input that was passed to the javascript function and c) output - the output returned by the javascript function. This execution object then gets stored into the window.localStorage object. The co.js schedules a function that fetches these execution objects from the window.localStorage object and asynchronously sends them to the server. The server components are written in python. The server, when receives the execution object, stores it into the NoSql database. Selecting NoSql database makes sense as there is no set model for the objects in execution set.

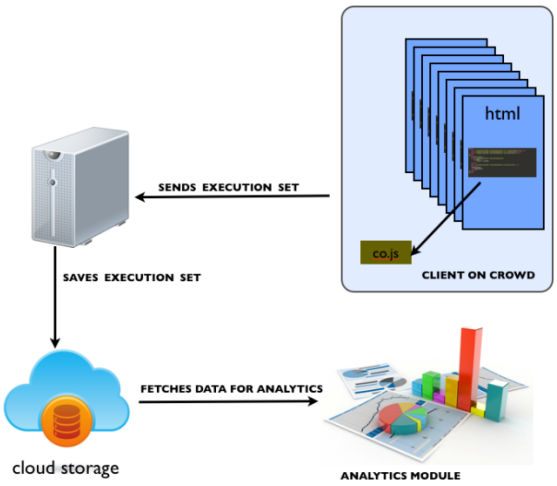

Figure-4 Schematic view of the Instrumentation Solution

V. THE SUBJECT SYSTEM – TEACHER SUBSTITUTE

The software subject of our experiment was the Sububi system, which is a Javascript tool to experiment with different ways for ranking substitute teachers. A substitute teacher is called by a school to cover a hired teacher who could not attend to class. The software is available to the community via a GitHub (https://github.com/christianadriano/sububi). The inputs for testing the system comprise the following data (Figure-5):

- Rating: a number from 1 to 3
- Seniority: years of work experience
- Subjects, Grades and Schools preferred or accepted to teach

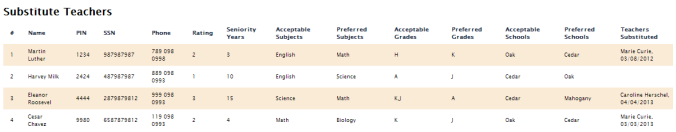

Figure-5 – Substitute Teacher Interface

There are points allocated for matching a vacancy in one school versus preferences of the substitute, the rating and seniority. The points work as a ranking. Figure-6 displays one point circled in red.

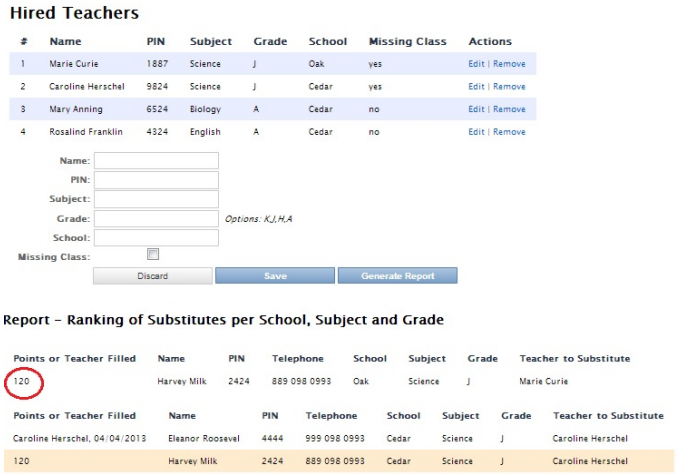

Figure-6 Sububi interface with the Rating calculations

VI. SUSPICIOUSNESS METRICS

A. *Probabilistic Suspiciousness*

Obtain the execution set for each function call in the execution set. Then we get the function calls in the history for which there are the same inputs. With this information we create a distribution of input/output pairs.

For a specific input and output pair, we verify the probability of such pair happening in the distribution. For that we would have two probability distributions - one for satisfactory executions and other for unsatisfactory.

Formula:

Suspiciousness = Probability of input/output pairs not happening in a satisfactory execution * Probability of happening in an unsatisfactory execution

B. *Historical Suspiciousness*

After collecting all input and output pairs, tag them all according to satisfactory and unsatisfactory executions. Furthermore, categorize them in terms of the source code (file) version they pertain to.

For each function call in the execution set compare against function calls in the history set and add points based on the quadrants of figure-7.

Figure-7 Quadrants for Historical Suspiciousness

VII. TECHNOLOGICAL CHALLENGES

A. Javascript Issue

Javascript cannot send cross domain requests because of Access-Control-Allow-Origin policies. We are using jsonp (JSON with padding) as a hack to solve this issue. This means we cannot send POST requests to our server. In order to send the execution sets to the server we are sending smaller but many GET requests to our server, making sure we respect the uri length limits of the GET requests. This will be changed in future, as we will shift to a W3C recommended Cross-Origin Resource Sharing (CORS) protocol. CORS will enable us to send POST requests to server.

B. Instrumentation Issue

JQuery AOP is not aimed directly at instrumentation but instead to be used as a JQuery plugin to provide aspect oriented programing features to Javascript. It allows to add advices (Before, After, After Throw, After Finally, Around and Introduction) to any global or instance object. We want to add advices only to the user defined custom Javascript functions. We want to avoid all the global function or library (JQuery, mootools or any third party library) functions. To achieve this we made it necessary that a context object named 'co_context object' should be present in the DOM. This co_context object contains a key called 'targets' and value as an array of namespaces of the user defined Javascript libraries. An example for co_context is given below:

```
co_context = {
    "targets":[namespace1,namespace2,]
 }
```

Co.js reads the co_context object and adds advices to the functions that are defined in the targets.

C. Asynchronous Javascript Calls

User defined Javascript functions which make asynchronous calls (AJAX) to the application server, owing to the non-blocking nature of asynchronous calls, are difficult to instrument to get the complete execution object. On receipt of the response from server, the response handler, a callback method, gets called and hence there is a nonlinear path of execution. We believe that these ajax calls can be a very rich and important source for the generation of execution sets and can generate some very useful input-output data patterns. These patterns can be further studied to gain insights about the bugs and patterns involved in the integration of client side and server side modules. In our future works, we are planning to handle these cases by adding advices to the response handlers.
After having stated all the limitations we envisioned we pass to the section where we discuss a proposal for evaluation our solution and the corresponding threats to validity.

VIII. SOLUTION EVALUATION

A. Crowd Environment

Below is a sketch of the crowd interface for tagging.

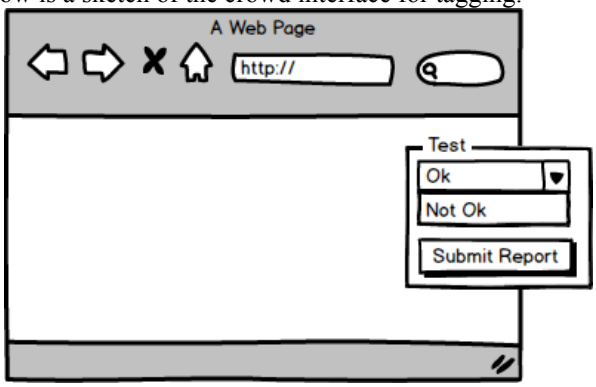

Figure-8 Tagging Interface

B. Data Gleaning

We plan have Sububi available for a group of undergraduate students to performed exploratory test. Students would be paid per number of different failures they identify in the software. Thus users have an incentive to explore a wide range of combinations of inputs.

C. Trace Granularity

The granularity planned is at function call level. We will evaluate the precision we can obtain from that before trying to design finer granularity (e.g., statement level).

D. Comparing against Competing Approaches

The first approach we have in mind is the suspiciousness metric proposed in Tarantula [9]. Some mapping would have to be made to consolidate the statement level metric to function level-based.

E. Possible Threats to Validity

1) Construct Validity.

There is a fundamental assumption in our research consisted in deeming satisfactory executions as expected behavior. This might not hold true for small number of executions or large amount of users without good knowledge of the actual system requirements. Since we would be working with a crowd, we are relying on previous work [15] demonstrating that decision making processes performed by a crowd filters out noises and individual low quality results. I.e., on an average the results are very good.

2) Internal Validity

There is at least one causal relation that might not hold in our research – infrequent input and output pairs to be failure-related. We plan to mitigate such issue by trying to influence how the crowd chooses the test inputs. This is why we have an activity titled "Suggest Test Data" in figure-1.

3) External Validity

Our scenario and instrumentation targeted a web-based client and a Javascript code. This might not generalize for server side modules in which many clients share the same component

at runtime. For such architecture some adaptations to the instrumentation might be needed.

F. Modeling Dependencies among Versions and Log Traces

If we want to take in consideration the context of each method call (as in k-CFA analysis [18]), we would have to store the traces as they happened and discriminate them in respect to the version of the software they pertain. The problem with that is how to make use of the sheer amount of data being collected. An alternative is to store only the dependencies among methods. Such would work as an abstraction of the code which might turn also useful to store metadata about the crowd exploratory testing activity. For instance, weighted dependencies based on criteria such as number of calls or volume of data exchanged between caller and callee. Moreover, the results of tagging could also be cascaded to the dependencies among methods.

As a solution to represent dependencies we would suggest the utilization of the Design Structured Matrix (DSM) [19] method. DSM has been shown useful to specify products with complex dependencies [20] and support designers to analyze change impacts. It is also very useful to detect circular dependencies and to understand how parts are clustered, and therefore support decision making on modularizing the development process itself [21]. Since exploratory testing is also understood as a scientific-like inquiry, keeping an updated and evolving perspective of the testing activity as a design space is fundamental to draw and verify hypotheses.

IX. SAMPLE OF RESULTS

Below we show the results obtained by the instrumentation and data collection as displayed in figure-9.

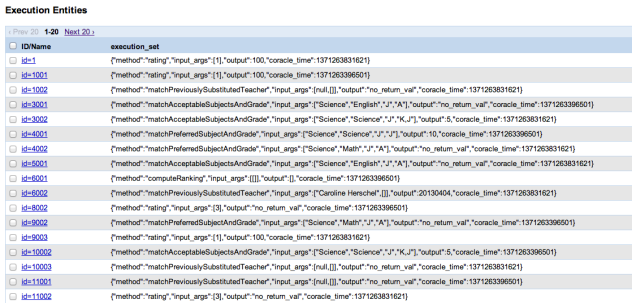

Figure-9 Datastorage view

The field "coracle_time" is the time from epoch in milliseconds used to uniquely identify when the trace was captured at the client side. This is important to confirm which execution happened when. So that all traces form the test execution are clustered within the timestamp.

Moreover, we are planning to utilize the browser ID sent by the user agent (browser) to also uniquely identify users. Hence, time-stamp and the browser ID would mitigate the occurrences of testing collisions. I.e., multiple users executing the same test at same time with the same data input. Such situation might not seem plausible in a traditional testing environment, but with a crowd this might become an issue. Below we exemplify each type of trace captured.

A. Method with Numerical Input and Output


```
{"method":"rating","input_args":1,"output":100,"coracle_time
":1371263396501}
```


In our example we don't have String input and Numerical output. We also don't have String as both input and output.

B. Array inputs and Numerical output


```
{"method":"matchAcceptableSubjectsAndGrade",
"input_args":["Science","Science","J","K,J"],
"output":5,
"coracle_time":1371263396501}
```


C. Array inputs and no output


```
{"method":"matchAcceptableSubjectsAndGrade",
"input_args":["Science","English","J","A"],
"output":"no_return_val",
"coracle_time":1371263396501}
```


After analyzing the traces we found an error (an internal deviation) in the Sububi software. Below is the log trace for that.


```
{"method":"rating",
"input_args":[3],
"output":"no_return_val",
"coracle_time":1371263831621}
```


The error consisted of not actively treating the input with value "3" and instead just returning and empty value.

X. CONCLUSION AND FUTURE WORK

Mobilizing a crowd of testers and supporting their work in a structured and consistent manner is a novel use for the proven strengths of fault localization techniques and exploratory testing methods. The positive outcome is to have exploratory testing producing quantitative data to support concrete and less subjective measures of confidence on the quality of a software product. Hence, it puts teams in a position to effectively discuss how to employ the power of the crowd to speed up the confidence building process.

Meanwhile, our approach opens the space to understand what types of tests are amenable to be delegated to a crowd. In line with that are questions regarding the amount of information and guidance necessary to have new people performing acceptance test. The negative outcome of our research is that many technological issues impose challenges to setup software for crowd testing. As we stated in previous sections we did not have the answers for all of them, although it is crucial to demonstrate the usefulness of the approach in an experiment with crowd realistic scale.

The future work is threefold. First, set and run experiments

to investigate how many faults can be found and discuss the effects of the granularity of traces on precision and accuracy. Second, implement a tagging tool to enable the automatic collection of tester opinion and impressions about each execution. With the tool we will be able to investigate how human factors relate to the quality of the test. I.e., what quality of testing means when working with a crowd. Third, we also plan to experiment with the dependency model and verify if more precise information implies in more precise fault localization.

.

REFERENCES

[1] Bernstein, et al., "Soylent: A Word Processor with a Crowd Inside", in proceedings of UIST'10, pp. 313-322, 2010.

[2] Kittur, A., et al., "CrowdForge: Crowdsourcing Complex Work", in proceedings of UIST'10, pp. 43-52, 2011.

[3] Alonso, O., Rose, D., E., Stewart, B., "Crowdsourcing for Relevance Evaluation", in ACM SIGIR Forum, v.42(2), pp. 9-15, 2008.

[4] Bozzon, et al., "Extending Search to Crowds: A Model-Driven Approach", in S. Ceri and M. Brambilla (Eds.): Search Computing III, LNCS 7538, pp. 207–222, 2012.

[5] Goldman, M., Little, G., Miller, R., C., "Collabode: Collaborative Coding in the Browser", in CHASE'11, pp. 65-68, 2011.

[6] Pastore, F., Mariani, L., Fraser, G., "CrowdOracles: Can the Crowd Solve the Oracle Problem?", in proceedings of ICST, pp. 2-12, 2013.

[7] Breu, et al., "Information Needs in Bug Reports: Improving Cooperation between Developers and Users", in proceedings of CSCW'10, pp. 301-310, 2010.

[8] La Toza, T., D., Myers, B., A., "Developers Ask Reachability Questions", in proceedings of ICSE'10, pp.184-195, 2010.

[9] Jones J. A., Harrold M. J., Stasko J., "Visualization of Test Information to Assist Fault Localization", in proc ICSE 2002.

[10] Breu, S., et al., Information needs in bug reports: improving cooperation between developers and users. in proc of CSCW 2010.

[11] Bortis G., van der Hoek A., "PorchLight: A Tag-based Approach to Bug Triaging", in proc. of CSCW 2011.

[12] Bertram D., et al., "Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams", in proc of CSCW 2010.

[13] Anvik J., Hiew L., Murphy, G.,C., "Who should fix this bug?", in proc ICSE 2006.

[14] Guo, P. J., et al., "Not My Bug! and Other Reasons for Software Bug Report Reassignment". In proc. of CSCW 2011.

[15] Surowiecki, James. The wisdom of crowds. Anchor, 2005.

[16] Beschastnikh, I, Brun, Y, Ernst, M., D., Krishnamurthy, A. and Anderson, T., A., (Dec. 2011) "Mining Temporal Invariants from Partial Ordered Logs", in SIGOPS Operating Systems Review, vol. 45, no. 3, Dec. 2011, pp. 39-46.

[17] Liblit, B., Mayur N., Zheng A., X., Aiken, A., and Jordan, M., I., (2005), "Scalable statistical bug isolation." ACM SIGPLAN Notices 40, no. 6 (2005): 15-26.

[18] Shivers, O.,"Control-flow analysis in Scheme", in Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI), pp. 164–174, 1988

[19] Browning, T.R.; , "Applying the design structure matrix to system decomposition and integration problems: a review and new directions," Engineering Management, IEEE Transactions on , vol.48, no.3, pp.292-306, Aug 2001

[20] Sangal N., et al., "Using dependency models to manage complex software architecture", in proc of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05), 2005

[21] Baldwin, C.Y., Clark K.B., "Design Rules – the Power of Modularity", vol.1 ed. The MIT Press; First Edition, Mar15, 2000

BIOGRAPHIES

Christian M. Adriano holds a bachelor and a master in Computer Engineering from State University of Campinas. He is currently a PhD student in software engineering at the Donald Bren School in the University of California Irvine. Christian has maintained a Project Management Professional credential since 2008 and has more than ten years of experience in developing software for the banking and oil sectors

Vaibhav P. S. Saini holds a bachelor in Information and Communication Technology from Dhirubhai Ambani Institute of Information and Communication Technology. He is currently a PhD student at the University of California Irvine. Vaibhav has worked for four years in the software industry mostly involved in developing web applications.