<table>
<thead>
<tr>
<th>Title</th>
<th>Evaluation metric for multiple-bug localization with simple and complex predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Zhang, Y; Lo, E; Kao, CM</td>
</tr>
<tr>
<td>Citation</td>
<td>The 19th Asia-Pacific Software Engineering Conference (APSEC 2012), Hong Kong, 4-7 December 2012. In Asia Pacific Software Engineering Conference Proceedings, 2012, v. 1, p. 288-293</td>
</tr>
<tr>
<td>Issued Date</td>
<td>2012</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/10722/164928">http://hdl.handle.net/10722/164928</a></td>
</tr>
<tr>
<td>Rights</td>
<td>Asia Pacific Software Engineering Conference Proceedings. Copyright © IEEE.; ©2012 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.; This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.</td>
</tr>
</tbody>
</table>
Evaluation Metric for Multiple-Bug Localization with Simple and Complex Predicates

Yiwei Zhang, Microsoft Corporation, yiweiz@microsoft.com
Eric Lo, Hong Kong Polytechnic University, ericlo@comp.polyu.edu.hk
Ben Kao, The University of Hong Kong, kao@cs.hku.hk

Abstract—Statistical debugging is a technique that mines data obtained from software executions in order to identify the program statements that are relevant to program bugs. Specifically, program predicates are injected into the program during compilation and statistics about those predicates are collected during the program execution. When bugs are found but the developers have no clue where the bugs are, they may call such a statistical debugger for help. The debugger ranks the injected predicates according to their statistical relevancy to bugs and presents the suspicious ones to the developers. When a bug is found and fixed, but the updated program still contains (some other) bugs, the preceding procedure is iterated until all bugs are fixed. There are two types of predicate-based statistical debugger: one type returns only simple predicates, another type returns only complex predicates. We envision that the next wave of statistic debuggers should be able to return both, depending on the kinds of bugs manifested in the software. In this paper, we take the first step and study the metrics for evaluating the effectiveness of statistical debuggers that can return both types of predicate predictors (simple or complex).

I. INTRODUCTION

Statistical debugging is a bug-localization technique that helps developers pinpoint bugs in programs. Since computer-aided bug-localization can greatly reduce the effort invested in software debugging, it has received growing attention in recent years [1], [14], [12], [13], [9], [10], [2], [6], [7], [5], [3].

A statistical debugging tool locates bugs in programs by analyzing data collected from software executions, such as test case executions and real user sessions. In predicate-based statistical debugging (PSD), a program is first instrumented by injecting extra code that evaluates Boolean expressions (called predicates) at various program points (called instrumentation sites) [8]. Upon termination of an execution of the instrumented program, a statistical report is generated that details how often an instrumentation site has been visited and how often a predicate has been evaluated true. Based on different statistical models, various statistical debugging techniques can be devised to rank the predicates according to their estimated relevancy to bugs. Top-ranked predicates provide insightful hints to developers in their bug-locating exercises. These predicates are considered good predictors of bugs and they significantly reduce the debugging time.

Most PSD techniques [13], [9], [10], [2] are iterative. First, instrumented predicates are ranked according to their statistical relevancy to bugs. These predicates are then studied manually to locate a bug that is presumably indicated by the predicates. If such a bug is found, it is fixed to produce a new program version. The new version of the program is tested again and a new statistical report is collected. In case the program still contains bugs, the preceding procedure is re-applied.

The aforementioned works mainly return simple-predicate predictors as the predictors. A recent work [2] points out that complex-predicate predictors (predictors composed by conjunctions and disjunctions of simple predicates) can capture some kinds of bugs better than simple-predicate predictors. We envision that the next wave of statistic debuggers should be able to return both, depending on the kinds of bugs manifested in the software. When evaluating the effectiveness of such statistical debuggers, it is necessary to devise metrics that can fairly compare the effectiveness of simple-predicate predictors and complex-predicate predictors. In this paper, we propose such an evaluation methodology and metric.

The rest of the paper is organized as follows. Section II gives background information. Section III elaborates the evaluation methodology and metrics. Finally, Section IV concludes and offers directions for future work.

II. PREDICATE-BASED STATISTICAL DEBUGGING TECHNIQUES

In order to enable statistical debugging, a subject program is first instrumented with a set of predicates \( \mathcal{P} = \{P_1, P_2, \ldots, P_n\} \) such that the runtime behavior in each execution of the program is encoded through predicate evaluations. Current instrumentation compliers like CBI [9] inject extra code that evaluates simple Boolean expressions, or simple predicates, at various program points (instrumentation sites). Predicates are designed to capture potentially interesting program behaviors such as return values of function calls, directions of branches, or values of variables. Through the execution of the instrumented program (e.g., running a suite of test runs), a statistical report can be generated to detail how often an instrumentation site has been visited and how often a predicate has been evaluated true.
Given a (simple) predicate \( P \), two conditional probabilities are computed:

\[
Pr_1(P) = Pr(G \text{ fails } | P \text{ is observed}),
\]

\[
Pr_2(P) = Pr(G \text{ fails } | P \text{ is observed true})
\]

The difference of the two probabilities \( \Delta(P) = Pr_2(P) - Pr_1(P) \) is taken as one of two measurements of how much the observation of \( P \) being true increases the likelihood that the subject program \( G \) fails. Intuitively, the larger the probability increment \( (\Delta(P)) \), the stronger is this lift of likelihood, and therefore the stronger is \( P \)'s predictive power.

We observe that the notion of probability increment is distorted by the interactions of bugs in the program. For example, consider the program fragment:

\[
\cdots \\
\text{b = a mod 2;} // Predicate P: (b = 0) is instrumented \\
\text{c = 1/b;} // Bug B1 here \\
\text{\ldots} \\
\text{d = 1/0;} // Another bug B2 here
\]

In this program, a predicate \( (b = 0) \) is instrumented at the program line \( b = a \mod 2 \); So, whenever that program line is executed during runtime, the program will also count how many times the predicate \( (b = 0) \) is evaluated. Let us assume that \( a \) is even in half of the runs. The predicate \( P: (b = 0) \) is therefore observed true half of the time. Since the program always fails, we have \( Pr_1(P) = Pr_2(P) = 1 \) and thus \( \Delta(P) = 0 \). The zero probability increment implies that \( P \) is not bug-relevant. However, if bug \( B_2 \) is not there, we have \( Pr_1(P) = 0.5 \) and \( Pr_2(P) = 1 \) and so \( \Delta(P) = 1 - 0.5 = 0.5 \). The significant increase in the probability allows us to identify \( P \) as a bug-relevant predicate.

Figure 1 shows an example report collected by executing a test suite of eight test runs \( R_1, R_2, \ldots, R_8 \) on an instrumented program injected with seven predicates \( P_1, P_2, \ldots, P_7 \). Each row in the figure shows the execution result of a test run \( R_i \), which is represented by a vector \( \langle |c_{i,1}|, |c_{i,2}|, \ldots, |c_{i,7}| \rangle \). In the vector, \( f_i \) is the failure-bit which is set to ‘1’ if \( R_i \) is a failed run (e.g., the program terminates abnormally or returns an incorrect result) and it is set to ‘0’ if \( R_i \) is a successful run. \( c_{i,j} \) is a counter that records the number of times that predicate \( P_j \) is evaluated true in \( R_i \). For convenience, we say “\( P_j \) is observed true” if predicate \( P_j \) is evaluated to be true at least once during the execution of \( R_i \). Therefore, \( c_{i,j} > 0 \) implies that \( P_j \) is observed true in \( R_i \). We say “\( P_j \) is observed” as long as predicate \( P_j \) is evaluated, no matter whether it is true or false. For example, the vector representing test run \( R_7 \) in Figure 1 is \( \langle |0|, |2|, |0|, |0|, |3|, |0|, |1| \rangle \). This means that \( R_7 \) is a failed run and predicate \( P_5 \) has been observed to be true three times during the execution of \( R_7 \) (because \( c_{7,5} = 3 \)). If \( P_j \) is never observed during the execution of \( R_i \), we set \( c_{i,j} \) as ‘-’.

One can obtain complex predicates from simple ones by applying logical operators such as conjunction and disjunction. Arumugha Nainar et al. [2] study the effectiveness of complex predicates. In that study, a complex predicate is either a conjunction or a disjunction of two simple predicates.

In the following, we give a brief introduction to several predicate-based statistical debugging techniques.

LIBLET05 [9] is the state-of-the-art PSD technique. Given a (simple) predicate \( P \), two conditional probabilities are computed:

\[
Pr_1(P) = Pr(G \text{ fails } | P \text{ is observed}),
\]

\[
Pr_2(P) = Pr(G \text{ fails } | P \text{ is observed true})
\]

The difference of the two probabilities \( \Delta(P) = Pr_2(P) - Pr_1(P) \) is taken as one of two measurements of how much the observation of \( P \) being true increases the likelihood that the subject program \( G \) fails. Intuitively, the larger the probability increment \( (\Delta(P)) \), the stronger is this lift of likelihood, and therefore the stronger is \( P \)'s predictive power.

<table>
<thead>
<tr>
<th>( R_i )</th>
<th>( f_i )</th>
<th>( c_{i,1} )</th>
<th>( c_{i,2} )</th>
<th>( c_{i,3} )</th>
<th>( c_{i,4} )</th>
<th>( c_{i,5} )</th>
<th>( c_{i,6} )</th>
<th>( c_{i,7} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( R_2 )</td>
<td>0.3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( R_3 )</td>
<td>0.1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( R_4 )</td>
<td>0.0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( R_5 )</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( R_6 )</td>
<td>0.1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1. Test Report Example

289
of each predicate is still affected by the interactions of \( m - 1 \) bugs, and so on until the last round where only one bug is left.

In fact, the statistical inference process of most iterative PSD techniques [9], [10], [2] is affected by bug interactions (except in their last iteration, where only bug one is left). As another example, let us consider SOBER [10]. Given a predicate \( P \), SOBER models the evaluations of \( P \) as independent Bernoulli trials. Given an execution run \( R \), SOBER determines the evaluation bias of \( P \) in \( R \) as the fraction of time that \( P \) is evaluated true in \( R \). SOBER compiles a distribution of the evaluation bias values among all the successful runs, and then compares the distribution against the distribution obtained from all the failed runs. If the two distributions are dissimilar, it is a strong evidence that \( P \) is very bug-relevant. Like IBLIT05, when there are \( m \) bugs in the program (say, in the first iteration, \( m > 1 \)), all failed runs actually consists of those that originate from different bugs. The distribution of the evaluation bias obtained from the failed runs is therefore unable to characterize any one particular bug.

ICML06 [14] is a PSD technique that is resistant to bug interactions. It is not iterative but are based on co-clustering and latent topic models, respectively. Empirical results show that they are generally more successful in locating bugs because their statistical inferences are not influenced by bug interactions. However, they require the full set of predicates as input. Since the number of complex predicates is exponentially large, the two techniques are not scalable to handle complex predicates.

In ISSTA07 [2], Arumuga Nairn et al. carried out a pioneering research on complex-predicate bug-localization. It incorporates the concept of complex predicate in statistical debugging and illustrates that complex predicates not only help developers in locating bugs, they also provide additional information for developers to fix the bugs. However, similar to IBLIT05 and SOBER, ISSTA07 is also an iterative approach and thus its statistical inference process may also be influenced by the presence of multiple bugs in each iteration (except for the last iteration). So, we expect that ISSTA07 returns excellent complex predicates when the subject program has only one single bug. However, the accuracy and usefulness of the returned predicates should be degraded when the number of bugs in the subject programs increases. Furthermore, ISSTA07 considers only complex predicates that are disjunction or conjunction of two simple predicates. However, the use of disjunctive complex predicates may lead to the super-bug problem. In addition, limiting complex predicates to those that are composed of two simple ones lowers the technique’s effectiveness in cases where certain bugs are best predicted by larger-sized complex predicates.

III. Evaluation Metrics

In order to objectively quantify the localization accuracy and the usefulness of the predicates returned by the bug-localization techniques, it is necessary to set up an evaluation framework. Most evaluation frameworks [12], [10] to-date are mainly based on a metric of distance between a suggested predicate and the actual location of the bug. We call this metric as distance-based metric, which is what we adopt in this study. To measure the distance between a predicate \( P \) (simple or complex) and the bug that it predicts, obviously, we need a location of the predicate and a location of the associated bug. This step is straightforward if we consider only simple predicates and if there is only one bug in the program: the instrumentation site of the simple predicate and the location of the single known bug are readily determinable. With complex predicates and multiple bugs in a subject program, however we need to address the following two issues: Bug labeling (Section III-A) and Measuring the distance from a complex predicate to a bug (Section III-B).

A. Bug labeling

Let us first consider simple predicates. Suppose there are multiple bugs in the program, then given a simple predicate \( P \) that is returned by a bug-localization technique, which bug in the program does \( P \) predict? In some previous work [12], [10], this question is answered by assuming that the predicate \( P \) is associated with the bug that is the closest to \( P \). More specifically, the subject program is first transformed into a program dependency graph (PDG) [11]. Each statement in the subject program is mapped to a vertex in the PDG. Given a simple predicate \( P \), we locate the vertex \( V_P \) in the PDG that is associated with the instrumentation site of \( P \). We then conduct a breadth-first search (BFS) starting from \( V_P \) until a vertex \( V_B \) associated with a buggy statement \( B \) is encountered. The predicate \( P \) is then assumed to predict \( B \) and is thus labeled as a bug-B predicate.

We note that this technique suffers from a couple of problems. First, if the predicate is a complex one, say, \( P = P_1 \land P_2 \land \ldots \land P_k \) for some \( k \geq 2 \), then there are \( k \) instrumentation sites, each of which could be a starting point of the BFS. Second, the assumption that “the predicate predicts the closest bug” might not be valid. In particular, if there are a large number of bugs in the program, then chances are that the vertex of some random bug \( B' \) is incidentally reached first during the BFS exploration from \( V_P \) to \( V_B \). In this case, \( P \) is mis-labeled as a bug-\( B' \) predicate and the measurement of distance from \( P \) to the bug that it predicts is inaccurate. Considering these complications, we devise a labeling technique that works under the multiple-bug complex-predicate scenario.

First, let us lay down some definitions.
Definition 1 (Simple-predicate signature). Given a simple predicate \( P \), and a set of execution runs \( \mathcal{R} = \{ R_1, R_2, \ldots, R_m \} \), the signature of the simple predicate \( P \) w.r.t. \( \mathcal{R} \) is the set of execution runs in \( \mathcal{R} \) in which \( P \) is observed true. That is, \( \operatorname{Sig}(P)_{\mathcal{R}} = \{ R_i \in \mathcal{R} | P \in \mathcal{R} \} \).

Definition 2 (Complex-predicate signature). Given a complex predicate \( P = (P_1 \land \ldots \land P_c) \), which is the conjunction of \( c \) simple predicates, let the signatures of \( P_1, P_2, \ldots, P_c \) w.r.t. the set of execution runs \( \mathcal{R} \) be \( \operatorname{Sig}(P_1)_{\mathcal{R}}, \ldots, \operatorname{Sig}(P_c)_{\mathcal{R}} \), respectively. The signature of the complex predicate \( P \) w.r.t. \( \mathcal{R} \) is the intersection of all its simple predicates’ signatures, i.e., \( \operatorname{Sig}(P)_{\mathcal{R}} = \bigcap_{i=1}^{c} \operatorname{Sig}(P_i)_{\mathcal{R}} \).

Definition 3 (Bug signature). Given a bug \( B \), and a set of execution runs \( \mathcal{R} = \{ R_1, R_2, \ldots, R_m \} \), the signature of bug \( B \) w.r.t. \( \mathcal{R} \) is the set of failed execution runs in \( \mathcal{R} \) when bug \( B \) is in the program. That is, \( \operatorname{Sig}(B)_{\mathcal{R}} = \{ R_i \in \mathcal{R} | R_i \text{ fails when } B \text{ is in the program} \} \).

So, ideally, a predicate \( P \) as a perfect predictor of a bug \( B \) if the signature of \( P \), \( \operatorname{Sig}(P) \), is identical to the signature of \( B \), \( \operatorname{Sig}(B) \). Our technique of labeling \( P \) with the bug that it predicts is based on this observation. More specifically, we compute the precision and recall of \( \operatorname{Sig}(P) \) matching \( \operatorname{Sig}(B) \).

Given a predicate \( P \), and a candidate bug \( B \) that it might be labeled with, the precision of \( P \) with regard to \( B \) measures how likely a failed run in \( \operatorname{Sig}(P) \) indeed fails due to bug \( B \). More precisely, the precision of \( P \) with regard to \( B \) is defined as follows:

Definition 4 (Precision).

\[
\text{precision}(P, B) = \frac{\operatorname{Sig}(P) \cap \operatorname{Sig}(B)}{\operatorname{Sig}(P)}.
\]

Let us assume there are two candidate bugs \( B_1 \) and \( B_2 \). If \( \text{precision}(P, B_1) \) is greater than \( \text{precision}(P, B_2) \) then more failed runs in \( \operatorname{Sig}(P) \) fails due to \( B_1 \) than those due to \( B_2 \). Thus, it is more reasonable to regard \( P \) as a predictor of \( B_1 \) rather than of \( B_2 \). In other words, \( P \) should be labeled with \( B_1 \).

Recall is used to break ties. The recall of \( P \) with regard to a certain bug \( B \) measures how likely a failed run in \( \operatorname{Sig}(B) \) appears in \( \operatorname{Sig}(P) \) and is calculated as follows:

Definition 5 (Recall).

\[
\text{recall}(P, B) = \frac{\operatorname{Sig}(P) \cap \operatorname{Sig}(B)}{\operatorname{Sig}(B)}.
\]

We combine precision and recall as \( \text{PR-measure} \). More precisely, given a predicate \( P \) and the bug \( B \) it is labeled with, \( P \)'s PR-measure w.r.t. bug \( B \) is defined as

Definition 6 (PR-measure).

\[
\text{PR-measure}(P, B) = (\text{precision}(P, B), \text{recall}(P, B)),
\]

which is an ordered pair. We also define the following “greater-than” operator between two given PR-measures.

Definition 7 (Greater-than). Given a predicate \( P \) and two bugs \( B_1 \) and \( B_2 \), \( P \)'s PR-measures with regard to \( B_1 \) and \( B_2 \) are denoted as \( \text{pr-measure}_1 = (\text{precision}_1, \text{recall}_1) \) and \( \text{pr-measure}_2 = (\text{precision}_2, \text{recall}_2) \), respectively. We say \( \text{pr-measure}_1 \) is greater than \( \text{pr-measure}_2 \), denoted as \( \text{pr-measure}_1 \succ \text{pr-measure}_2 \), if either of the following two conditions is satisfied:

1. \( \text{precision}_1 > \text{precision}_2 \)
2. \( \text{precision}_1 = \text{precision}_2 \) and \( \text{recall}_1 > \text{recall}_2 \)

To sum up, given a predicate \( P \) and a set of bugs \( P \) might predict, we compute \( P \)'s PR-measure with regard to each bug, and label \( P \) with the bug \( B \) that gives the largest PR-measure among all bugs.\(^1\) Note that our labeling technique is reliable because it is independent of the type of predicate (simple or complex) and the number of bugs in the program.

B. Measuring the distance from a complex predicate to a bug

Given a predicate \( P \) that is labeled as a bug-B predicate, if \( P \) is a simple predicate, then the distance between \( P \) and \( B \) can be measured by the \( T \)-score [4] based on the PDG:

Definition 8 (T-score). Let \( V_P \) and \( V_B \) be the vertices in the PDG that are associated with the instrumentation site of the simple predicate \( P \) and bug \( B \), respectively. We perform BFS starting from \( V_P \). Let \( x \) be the number of vertices visited before \( V_B \) is reached, and let \( y \) be the total number of vertices in the PDG, we have

\[
T\text{-score}(P, B) = \frac{x}{y} \times 100\%.
\]

If \( P \) is a complex predicate, say \( P = P_1 \land \ldots \land P_k \) \( (k \geq 2) \), there are \( k \) instrumentation sites. How shall we define distance in this case? We address this issue by proposing four distance metrics. Each metric is designed to mimic a possible behavior of developers. For comprehensiveness, we suggest the effectiveness of the various bug-localization techniques could be evaluated using all four metrics (instead of just one).

1. \( \text{Min-Distance Metric} \). This metric measures the smallest \( T \)-score from any component simple predicate \( P_i \) to \( B \):

\[
\text{MinDist}(P, B) = \min_{1 \leq i \leq k} \{ T\text{-score}(P_i, B) \}.
\]

The MinDist metric models an experienced developer who has some prior knowledge on the subject program, and thus he knows which simple predicate in \( P \) leads to the

\(^1\)Although the signatures of bugs are unknown in practice (unless there is only one bug in the program), in controlled experiments, the bugs are known. Therefore, we can identify the failed cases, and hence the signature of each bug to compute the PR-measure.
bug most directly. Note that the MinDist metric somewhat favors complex predicates that have a large number of component predicates (i.e., a large $k$) because it is the minimum of the component predicates’ T-scores that we take. So, a larger $k$ implies more component predicates, and the minimum T-score tends to be smaller.

(2) **Max-Distance Metric.** This metric measures the largest T-score from any component simple predicate $P_i$ to $B$:

$$MaxDist(P, B) = \max_{1 \leq i \leq k} \{ T-score(P_i, B) \}. \quad (3)$$

The MaxDist metric models a jinxed developer who is so unlucky that he always picks the component predicate that is the farthest away from the bug to start his exploration. Opposite to the MinDist metric, MaxDist is unfavorable to large complex predicates. The MaxDist metric is presented as a worst-case analysis. In practice, these worst cases are unlikely to occur very often.

(3) **Avg-Distance Metric.** This metric measures the average T-score from all component simple predicates to $B$.

$$AvgDist(P, B) = \frac{\sum_{i=1}^{k} T-score(P_i, B)}{k}. \quad (4)$$

The AvgDist metric models the situation in which the developer randomly picks a component predicate in $P$ to start his exploration. The metric thus makes no assumption of the knowledge of the developer. In practice, however, we expect that the developer does have some knowledge about the subject program and therefore he should be able to pick a better component predicate to start with other than a random one.

(4) **$t$-Granularity Metric.** This metric models a developer who explores the component predicates concurrently. He first starts with the component predicate $P_1$ and explores a neighborhood of $t$ vertices from the instrumentation site $V_{P_1}$ of $P_1$ in the PDG. If the bug is not located, the developer switches to $P_2$ and similarly explores a neighborhood of $t$ vertices of $P_2$, and so on. If all $k$ component predicates have been so investigated without the bug located, the developer continues with the first predicate $P_1$ again and further explores a $t$-vertex neighborhood from where he last left off with $P_1$. If the bug is still not yet found, the developer continues with $P_2$ and so on in a round-robin fashion until the bug is finally located. Let $P_{\text{min}}$ be the component predicate that leads the developer to locate the bug first and let $T_{\text{min}} = T-score(P_{\text{min}}, B)$.

The amount of effort spent by the developer with regard to $P_{\text{min}}$ is $T_{\text{min}}$ while the amount of effort spent with regard to the other $k-1$ component predicates are $(k - 1) \cdot t \cdot \left\lceil \frac{T_{\text{min}}}{t} \right\rceil$. Hence, the total amount of effort under the $t$-granularity metric is given by:

$$t\text{-granularity}(P, B) = T_{\text{min}} + (k - 1) \cdot t \cdot \left\lceil \frac{T_{\text{min}}}{t} \right\rceil. \quad (5)$$

**IV. Conclusion**

This paper proposes a metric and methodology for evaluating the effectiveness of statistical debugging techniques that involve both simple-predicate predictors and complex-predicate predictors. In the bug labeling part, we propose to accomplish that by leveraging the concepts of “precision” and “recall” from the informational retrieval community. In the evaluation part, we propose four different metrics: Min-Distance, Max-Distance, Avg-Distance, and $t$-Granularity and encourage evaluations based on all four of them for full comprehensive reasons. As future work, we will devise a statistic debugger that is able to return both simple predicates and complex-predicates for different types of bugs.

**Acknowledgement:** Research supported by grant 525009E from Hong Kong RGC and grant A-PK94 from Hong Kong Polytechnic University.

**REFERENCES**


