Failure Proximity: A Fault Localization-Based Approach *

Chao Liu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
chaoliu@cs.uiuc.edu

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
hanj@cs.uiuc.edu

ABSTRACT

Recent software systems usually feature an automated failure reporting system, with which a huge number of failing traces are collected every day. In order to prioritize fault diagnosis, failing traces due to the same fault are expected to be grouped together. Previous methods, by hypothesizing that similar failing traces imply the same fault, cluster failing traces based on the literal trace similarity, which we call *trace proximity*. However, since a fault can be triggered in many ways, failing traces due to the same fault can be quite different. Therefore, previous methods actually group together traces exhibiting similar behaviors, like similar branch coverage, rather than traces due to the same fault. In this paper, we propose a new type of *failure proximity*, called R-PROXIMITY, which regards *two failing traces as similar if they suggest roughly the same fault location*. The fault location each failing case suggests is automatically obtained with SOBER, an existing statistical debugging tool. We show that with R-PROXIMITY, failing traces due to the same fault can be grouped together. In addition, we find that R-PROXIMITY is helpful for statistical debugging: It can help developers interpret and utilize the statistical debugging result. We illustrate the usage of R-PROXIMITY with a case study on the grep program and some experiments on the Siemens suite, and the result clearly demonstrates the advantage of R-PROXIMITY over trace proximity.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – Debugging aids, Diagnostics, Testing tools, Tracing

*This work was supported in part by the U.S. National Science Foundation NSF ITR-03-25603, IIS-02-09199, and IIS-03-08215. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM 1-59593-468-5/06/0011 ...$5.00.

General Terms

Algorithms, Design, Reliability, Experimentation

Keywords

Failure proximity, Debugging aids, Statistical debugging

1. INTRODUCTION

Recent complex software systems, like Netscape/Mozilla, Microsoft Windows and GNOME, usually feature an automated failure reporting component. When a software crash is detected, related information, like the stack trace, is collected, and with the user's permission, sent back to developers for diagnosis. Besides crash scenes, in the cooperative debugging framework, whole execution traces can be also collected at runtime (with sampling), and sent back to developers with low overhead [11, 12].

In principle, these collected failing traces can assist developers in prioritizing and diagnosing faults. However, it has never been as straightforward as one would expect. In the first place, a developer needs to group failures into clusters such that failing traces due to the same fault are grouped together. Since failing traces mainly contain dynamic behaviors, like branch or statement coverage, identifying traces due to the same fault is in general hard. In the second place, for each grouped cluster, one needs to manually investigate and guess about the fault location, and assign failing traces to developers who are responsible for the (guessed) fault location. Finally, as failure reporting has been fully automated, the sheer number of collected reports has increased dramatically, which has ultimately rendered manual processing infeasible.

Because manual processing is unrealistic, techniques have been developed to automate failure analysis. These techniques typically hypothesize that similar failing cases imply the same fault, and consequently, group together similar failing traces. For example, Podgurski et al. calculate the Euclidean distance between failing traces, and then cluster and visualize these failures based on the pair-wise distance [16]. As one can see, a proper definition of the *failure proximity* is the central question for automated analysis of failing traces. Subsequent clustering and visualization only serve as a vehicle for presentation. We denote this proximity defined in terms of trace similarity as *trace-proximity*, or T-PROXIMITY for short.

Although T-PROXIMITY is widely adopted to measure the proximity between executions [4, 5, 16], it is nevertheless weak in characterizing the semantic proximity between fail-

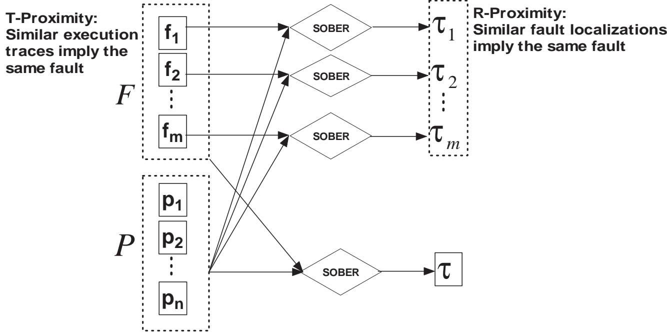


Figure 1: T-Proximity and R-Proximity

ing cases. First, because a fault can be triggered with different inputs, failing traces due to the same fault can be, and usually are, divergent. This suggests that clustering based on T-Proximity tends to group similar executions, rather than the failing cases due to the same fault. Secondly, clustering based on T-Proximity does not provide any information about the possible fault location. In consequence, a developer needs to manually investigate each cluster, and assign failing traces to responsible developers.

In order to circumvent the above weakness of T-Proximity, a new definition of failure proximity is proposed in this paper: Instead of relying on trace similarity, we regard *two failures as similar if they suggest roughly the same fault location.* The fault location each failing trace suggests is obtained with a statistical debugging tool Sober [13] in this paper whereas other automated debugging could be similarly employed. We now illustrate the idea in the following.

Suppose m failing traces, $F = \{f_1, f_2, \cdots, f_m\}$, and n passing traces, $P = \{p_1, p_2, \cdots, p_n\}$, are collected, and each trace is represented by the evaluation history of instrumented predicates. *Conventionally*, Sober takes F and P as the inputs, and generates a ranked list τ of instrumented predicates, *i.e.*, $\tau = \text{Sober}(F, P)$. The predicate ranking τ is the fault localization result, and its top predicates usually point to the fault location. We note that Sober is *not* required to take F and P in their entirety; instead, any subsets of F and P can be fed into Sober for fault localization. Therefore, by contrasting each failing trace f_i against P, the fault location each f_i suggests is *automatically* obtained, *i.e.*, $\tau_i = \text{Sober}(\{f_i\}, P)$, $\forall f_i \in F$. In this way, failure proximity can be measured by examining how τ_i's agree with each other: *Two failing cases are similar, if their induced predicate rankings roughly agree.* Because this failure proximity is defined in terms of predicate rankings, we denote it as "rank-proximity", or R-Proximity in short. A weighted form of the Kendall's tau distance is developed in this paper (Section 3.2) to measure the agreement between rankings. The connection and difference between R-Proximity and T-Proximity are illustrated in Figure 1.

In comparison with T-Proximity, R-Proximity is semantically closer to the underlying faults. Ideally, we want to partition failing cases such that failures due to the same fault are grouped together. However, since the "due-to" relationship between failing cases and underlying faults is unknown without expensive manual investigation, this ideal partition is generally unachievable. Here, we use Sober to substitute for the manual investigation, and consequently, the fault location each failing case suggests is an approximation to the real "due-to" relationship. Therefore, under R-Proximity, failing traces whose induced fault localization agree with each other are regarded near to each other. This is fundamentally different from T-Proximity, which defines failure proximity in terms of the literal similarity between traces.

Moreover, as R-Proximity is defined based on the fault location each failing trace suggests, clustering with R-Proximity automatically provides a guess about the fault location for each cluster. Specifically, because only failures agreeing on the fault location are grouped together, their agreement just represents the fault location they suggest. With such information, failing cases can be assigned to responsible developers without manual fault investigation. This, however, cannot be achieved by clustering with T-Proximity.

Finally, R-Proximity can also help developers interpret statistical debugging results, and this, again, cannot be accomplished with T-Proximity. Statistical debugging was proposed recently [11], and has been shown to be one of the most accurate techniques for fault localization [13]. Although statistical debugging could be accurate, developers usually complain about its interpretability. Statistical debugging produces a predicate ranking τ as its localization result, but without a *concrete* failing case to illustrate why certain predicates are ranked high in τ, a developer may find the result hard to interpret. Under R-Proximity, we can select a failing case t_i whose corresponding τ_i mostly agrees with τ, and this selected case can help the developer understand and utilize the statistical debugging result. In section 4, we report on a case study with the program grep, which exemplifies how R-Proximity helps developers diagnose multiple faults with Sober.

Certainly, the approximation of R-Proximity to the real "due-to" relationship depends on the quality of fault localization. The statistical debugging tool Sober used in this study is known as one of the most accurate [13], and it indeed performs very well in our case study. In the future, when better statistical debugging tools become available, Sober can be safely replaced without changing the definition of R-Proximity. In general, the idea of defining failure proximity based on fault localization should be compatible with other automated debugging tools, like delta debugging [18] and Tarantula [9]. The key point is that we try to find the fault each failing trace suggests in an *automated* way, and then define failure proximity based on the fault, rather than the literal trace similarity.

In summary, this study makes the following contributions:

1. We propose a new approach R-Proximity to assessing the proximity between failing traces. It is defined based on the fault location each failing trace suggests, and is hence more suitable to cluster failing traces than existing trace-similarity based approaches.

2. R-Proximity is defined by fingerprinting each failing trace into a predicate ranking. To the best of our knowledge, this is the first piece of work using fault localization tools for failing trace exploration. This work suggests that automated debugging techniques are *not* restricted to fault localization.

3. R-Proximity helps alleviate the interpretation problem of statistical debugging. We report on a case

study with grep in Section 4, which exemplifies this usage. We also complement the case study with a systematic evaluation of R-Proximity with the Siemens programs, which confirms the advantages of R-Proximity in characterizing the failure proximity.

The rest of the paper is organized as follows. Section 2 provides some background knowledge which is used in following sections. The detail of R-Proximity is discussed in Section 3. We discuss the advantages of R-Proximity in Section 4, and illustrate them with a case study. Experiments complementing the case study are presented in Section 5. With related work and threats to validity discussed in Section 6, Section 7 concludes this study.

2. BACKGROUND

Failure proximity is not a new topic in software engineering research. In fact, the problem of how to define and measure the proximity between executions is one of the central questions in many researches related to dynamic analysis [2,4,8,16].

Before a proximity (or similarity) measure is defined, executions need to be profiled first. An execution is usually profiled according to the runtime behavior, such as control flows and statement coverage. Because most runtime behaviors can be expressed in terms of predicates, we assume that an execution is profiled as a predicate vector in this study, and the predicate vector is referred to as the execution trace.

Suppose a program $\mathcal{P}$ is instrumented with L predicates, then the execution with input t is profiled as an L-dimensional vector v_t, where the ith dimension $v_t(i)$ records how many times the ith predicate P_i evaluates `true` during the execution. Depending on the need and overhead tolerance, many kinds of predicates can be instrumented. In this study, we instrument programs with the following two kinds of predicates, which are shown effective in characterizing executions [12,13].

- **[boolean]**: For each boolean expression B, a predicate, "B=`true`", is instrumented.
- **[return]**: For each function return site R, three predicates, "R>0", "R=0" and "R<0" are instrumented.

With executions profiled as predicate vectors, existing distance metric, like the Euclidian distance or Manhattan block distance, can be used to define a proximity measure between executions. In fact, a one-to-one correspondence exists between distance and proximity: a small distance means proximity. When a distance is calculated from execution traces *directly*, the resultant proximity is called "trace-proximity", or T-Proximity in short. In particular, as the Euclidean distance is used in this study, T-Proximity also refers to the proximity defined with the Euclidean distance. In the next section, we introduce "rank-proximity", which, in contrast, calculates the execution distance *indirectly* from execution traces.

3. R-PROXIMITY: FAILURE PROXIMITY BASED ON FAULT LOCALIZATION

In this section, we examine R-Proximity, which better characterizes the proximity between failing cases. We first discuss how to fingerprint each failing case into a predicate ranking in Section 3.1, and then study how to assess failure proximity based on the ranking presentation in the rest of this section.

3.1 Fingerprint Failing Cases

Given m failing cases $F = \{f_1, f_2, \cdots, f_m\}$ that are due to k faults (or bugs) $B = \{b_1, b_2, \cdots, b_k\}$, the "due-to" relationship between failures and faults is an injective function $\mathcal{F} : F \to B$. Ideally, the optimal proximity assigns a small distance to f_i's due to the same fault, and a large distance to ones due to different faults.

Apparently, the optimal proximity is hard to obtain because the "due-to" function $\mathcal{F}$ is unavailable without expensive manual investigation. But inspired by recent progress on fault localization, we wonder whether it is possible to approximate the "due-to" function with existing automated debugging tools, *i.e.*, substituting the manual investigation with automated fault localization. If it is, a sub-optimal proximity measure can be naturally defined.

A spectrum of fault localization tools have been developed in recent years [3,9,12,13,17–19]. These tools mainly contrast the set of failing cases F against a set of passing cases $P = \{p_1, p_2, \cdots, p_n\}$. Through the contrast, a number of program points are circled out as suspected fault locations. In this study, we use Sober, an existing statistical debugging tool, because (1) it takes the predicate representation of execution traces, and (2) it has been shown effective in fault localization [13].

Sober localizes underlying faults by contrasting the evaluation bias of each predicate in failing cases against that in passing cases. The *evaluation bias* basically measures, for each predicate, what percentage of evaluations are `true`. For example, if one predicate P is evaluated 10 times during one execution, and it evaluates `true` for three times, the evaluation bias of P in this execution is 0.3. Readers interested in more details about Sober are referred to [13].

Conventionally, Sober takes P and F as inputs, and produces a ranked list τ of the L instrumented predicates, *i.e.*, $\tau = \text{Sober}(F, P)$. The ranking τ is called the *composite ranking*, and $\tau(P_i)$ is the *rank* (or position) of the predicate P_i in τ. We say predicate P_i ranks *higher* or *before* predicate P_j in τ if and only if $\tau(P_i) < \tau(P_j)$. In general, higher ranked predicates are more likely to be *fault-relevant*, *i.e.*, pointing to the fault location or its vicinity.

As one may have noticed, Sober is not restricted to contrasting F against P as a whole. Instead, any subsets of F and P can be contrasted. Now let us consider an extreme scenario, where each failing case $f_i \in F$ is contrasted against P with Sober. Specifically,

$$\tau_i = \text{Sober}(\{f_i\}, P) \quad (i = 1, 2, \cdots, m), \tag{1}$$

and τ_i is the fault localization of f_i with Sober. In parallel with the composite ranking τ, τ_i's are called *individual rankings*.

Although τ_i may not necessarily pinpoint the fault location (depending on the quality of Sober, f_i and P), it does suggest which predicates f_i regards as suspicious. In this sense, τ_i embodies f_i's opinion about the fault(s). Therefore, we can represent each failing case f_i by its induced predicate ranking τ_i, and measure the proximity between f_i's in terms of the agreement between τ_i's. Therefore, we need a proper distance definition between rankings, which we will discuss in the next three subsections. Because this proximity is de-

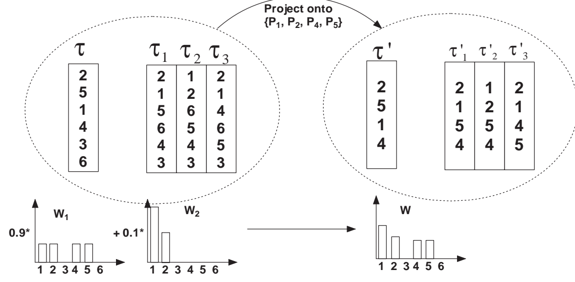


Figure 2: Rank Projection and Predicate Weighting

fined on the ranking presentation of failing cases, it is called "rank-proximity", or R-PROXIMITY for short.

3.2 Kendall's tau Distance

Given the ranking presentation of the failing cases, some standard rank distance, like the Kendall's tau distance [10], can be used. Let π and σ be two rankings of the L predicates, the Kendall's tau distance $D_K(\pi, \sigma)$ is defined as

$$D_K(\pi, \sigma) = \sum_{1 \leq i < j \leq L} K(P_i, P_j) \tag{2}$$

where

$$K(P_i, P_j) = \begin{cases} 1 & \text{if } [\pi(P_i) - \pi(P_j)][\sigma(P_i) - \sigma(P_j)] < 0, \\ 0 & \text{otherwise.} \end{cases}$$

Predicates P_i and P_j constitute a *discordant pair* if their relative orders in π and σ disagree, and the Kendall's tau distance essentially counts the number of discordant pairs between π and σ.

Although the Kendall's tau distance is a valid and reasonable distance measure for rankings in general, it, at least in its classic form (Eq. 2), fails to quantify the distance between predicate rankings. In the first place, we only need to consider fault relevant predicates because predicates are usually instrumented in a blind way and most predicates are superfluous. We need to find the subset of relevant predicates, and project original τ_i's onto it. Specifically, let S be the set of instrumented predicates ($L = |S|$), and $S_r \subseteq S$ is the subset of fault relevant predicates ($l = |S_r|$), a ranking τ is projected onto S_r if all predicates *not* in S_r are excluded from τ. We use τ' to denote the projected ranking of τ. In the second place, even within S_r, not all predicates are equally fault relevant. In general, we expect discordant pairs of more relevant predicates contribute more to the ranking distance. We use the following example to illustrate the two points.

EXAMPLE 1. *Suppose a program $\mathcal{P}$ is instrumented with 6 predicates, i.e., $S = \{P_1, P_2, P_3, P_4, P_5, P_6\}$. Figure 2 shows the composite ranking τ and individual ranking τ_1, τ_2, τ_3. For legibility, predicate indices are used in rankings. As we can see, predicates are not equally fault relevant. For example, P_2 is more relevant than P_5 because it ranks higher in all rankings. Furthermore, it is easy to calculate that $D_K(\tau_1, \tau_2) = D_K(\tau_1, \tau_3) = 2$, but intuitively, we would expect that τ_1 is closer to τ_3 than to τ_2 because τ_1 and τ_3 only disagree about the relative orders between less relevant predicates, like P_4, P_5 and P_6.*

Therefore, predicates need to be first weighted based on their fault relevance, and less relevant ones should be discounted in distance calculation.

3.3 Predicate Weighting

Because no guidance is available to set predicate weights, we choose to automatically derive the predicate weights from the composite and individual rankings. Intuitively, top predicates in these rankings are likely fault relevant.

First, the top-k_1 predicates of τ are taken as relevant, and the weight of predicate P_i is defined as

$$W_1^{k_1}(P_i) = \frac{I(k_1 - \tau(P_i))}{k_1}, \tag{3}$$

where $I(x)$ is an indicator function that equals to 1 if $x \geq 0$ and 0 otherwise. If a predicate ranks lower than k_1 in τ, it gets a zero weight, and is not selected into S_r *at this step*. For those selected predicates, equal weights are assigned although one could also assign decaying weights to lower-ranked predicates.

Second, the individual rankings τ_i's also suggest fault relevant predicates. Intuitively, the m individual rankings are like m votes for relevant predicates, and in consequence, predicates favored by more rankings are more fault relevant. Therefore, if the top-k_2 predicates of each ranking are considered, the weight of predicate P_i is defined

$$W_2^{k_2}(P_i) = \frac{\sum_{j=1}^{m} I(k_2 - \tau_j(P_i))}{mk_2}, \tag{4}$$

where $I(x)$ is the same indicator function as that in Eq. 3. This is called the *frequency weighting* because the weight is proportional to in how many τ_i's P_i ranks within the top-k_2.

Combining these two components, the weight of predicate P_i is

$$W(P_i) = (1 - \alpha) W_1^{k_1}(P_i) + \alpha W_2^{k_2}(P_i), \tag{5}$$

where α is the parameter balancing the two components. In fact, predicates within neither the top-k_1 of τ nor the top-k_2 of τ_i's receive a zero weight, and are essentially excluded from consideration in distance calculation (see Section 3.4). Predicates with nonzero weights then constitute the relevant predicate set S_r. In this study, we set $k_1 = 10$, $k_2 = 1$ and $\alpha = 0.1$ by default. We discuss how to adjust these parameters and their effects through experiments in Section 5.2.

EXAMPLE 2. *Continue Example 1. Taking $k_1 = 4$, $k_2 = 1$ and $\alpha = 0.1$, $W_1 = (0.25, 0.25, 0, 0.25, 0.25, 0)$, $W_2 = (0.67, 0.33, 0, 0, 0, 0)$, $W = (0.292, 0.258, 0, 0.25, 0.25, 0)$. Therefore, predicates P_3 and P_6 are excluded, and $S_r = \{P_1, P_2, P_4, P_5\}$.*

3.4 Weighted Kendall's tau Distance

Based on the predicate weighting, we propose a weighted form of the Kendall's tau distance as below.

DEFINITION 1 (WEIGHTED KENDALL'S TAU DISTANCE). *Given π and σ two rankings of the L predicates, the weighted Kendall's tau distance $D_{W,K}$ is defined as*

$$D_{W,K}(\pi, \sigma) = \sum_{1 \leq i < j \leq L} K(P_i, P_j) W(P_i, P_j), \tag{6}$$

where $K(P_i, P_j)$ is the same as that in Eq. 2, and $W(P_i, P_j) = W(P_i)W(P_j)$.

Because the weighted Kendall's distance is later used to define R-PROXIMITY, its validity as a distance metric is first proved in the following theorem.

THEOREM 1 (METRIC VALIDITY). *The weighted Kendall's tau distance $D_{W,K}$ is a metric on the set of L-predicate rankings if $W(P_i, P_j) > 0$ for $1 \leq i < j \leq L$, i.e., let π, σ, η be three rankings of L predicates, the following four properties hold:*
(1) $D_{W,K}(\pi, \sigma) \geq 0$,
(2) $D_{W,K}(\pi, \sigma) = 0$ *iff* $\pi = \sigma$,
(3) $D_{W,K}(\pi, \sigma) = D_{W,K}(\sigma, \pi)$,
(4) $D_{W,K}(\pi, \sigma) \leq D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma)$.

PROOF. The proofs for (1), (2) and (3) are trivial once $K(P_i, P_j)$ can only be either 0 or 1 and $W(P_i, P_j)$ is positive are recognized. In the following, we prove property (4), the triangle inequality.

To prove the triangle inequality, it suffices to show that for any nonzero term in $D_{W,K}(\pi, \sigma)$, the term also appears in $D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma)$. In fact, if $K(P_i, P_j) = 1$ in $D_{W,K}(\pi, \sigma)$, we know that the relative order between P_i and P_j are different in π and σ. Without loss of generality, suppose $\pi(P_i) < \pi(P_j)$ and $\sigma(P_j) < \sigma(P_i)$. Because there are only two possible orders between P_i and P_j in a ranking, the relative order between P_i and P_j in η must disagree with that in either π or σ. According to the definition of $K(P_i, P_j)$, the term $W(P_i, P_j)$ must also appear in $D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma)$. Therefore,

$$D_{W,K}(\pi, \sigma) \leq D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma) \quad (7)$$

Meanwhile, if the relative order of P_i and P_j agrees between π and σ, but disagrees with that in η, the right side of Eq. 7 is larger. The equal sign holds when no such predicate pairs exist. □

Based on THEOREM 1, the weighted Kendall's tau distance is a valid metric for distances between the projected rankings. And in consequence, R-PROXIMITY is defined: the distance between f_i and f_j is $D_{W,K}(\tau'_i, \tau'_j)$, which equals to $D_{W,K}(\tau_i, \tau_j)$. Continuing EXAMPLE 2. $D_{W,K}(\tau_1, \tau_2) = 0.07536 > 0.0625 = D_{W,K}(\tau_1, \tau_3)$, which conforms to what we expected. Finally, we note that the weighted Kendall's tau distance can be efficiently calculated, because the time complexity of the classic Kendall's tau distance is $O(n \log(n))$ [10], and the $W(P_i, P_j)$'s in Eq. 6 do not change the complexity. In our case, the distance between two rankings is calculated in $O(l \log(l))$ time, where $l = |S_r|$. In the next section, we discuss the advantages of R-PROXIMITY over the T-PROXIMITY.

4. ADVANTAGES OF R-PROXIMITY

R-PROXIMITY features a number of advantages over T-PROXIMITY. First, R-PROXIMITY is semantically closer to the underlying faults. Ideally, we want to partition failing cases such that failures due to the same fault are grouped together. However, since the "due-to" relationship between failing cases and underlying faults is unknown without expensive manual investigation, this ideal partition is generally unachievable. Here, we use SOBER to substitute for the manual investigation, and consequently, the fault location each failing case suggests is an approximation to the real "due-to" relationship. Therefore, under R-PROXIMITY, failing traces whose induced fault localization agree with each other are regarded near to each other. On the other hand, T-PROXIMITY approximates the "due-to" function by hypothesizing that similar traces implies the same fault. As we will see soon in the coming case study, failing cases due to the same fault can actually exhibit quite divergent behaviors.

Secondly, as R-PROXIMITY is defined based on the fault location each failing trace suggests, clustering with R-PROXIMITY automatically provides a guess about the fault location for each cluster. Specifically, because only failures agreeing on the fault location are grouped together, their agreement just represents the fault location they suggest. With such information, failing cases can be assigned to responsible developers accordingly. This, however, cannot be achieved by clustering with T-PROXIMITY.

Finally, R-PROXIMITY can also help developers interpret statistical debugging results, and this, again, cannot be accomplished with T-PROXIMITY. Statistical debugging produces a predicate ranking τ as its localization result, but without a *concrete* failing case to illustrate why certain predicates are ranked high in τ, a developer may find the result hard to interpret. With R-PROXIMITY, we can find a failing case t_i whose corresponding τ_i mostly agrees with τ. This selected case can help the developer understand and utilize the statistical debugging result. In the rest of this section, we illustrate the above three points through a case study with the `grep` program.

4.1 Fault Injection

We obtained a copy of the `grep`-2.2 subject program from the "Subject Infrastructure Repository" [6]. The program has 15,633 lines of C code, and is accompanied with a test suite of 470 test cases. We manually injected two faults into the source code, which are shown in Figure 3.

The first fault (Fault 1) is an "off-by-one" error: We added the "+1" at line 553. This fails 48 of the 470 test cases. The second fault (Fault 2) is a "subclause-missing" error. We commented out the subclause `lcp[i] == rcp [i]` at line 2270, and this incurs another 88 failing cases. We use F_1 and F_2 to denote the sets of failing cases due to Fault 1 and Fault 2, respectively. When both faults are present, all cases in F_1 and F_2 fail, resulting a total of 136 failing cases. Because F_1 and F_2 do not intersect, Fault 1 is the culprit for the failing cases in F_1, and Fault 2 for those in F_2.

We note that although the two faults are manually injected, they do mimic realistic faults. When developers are unclear about the corner condition, logic errors like "off-by-one" or "subclause-missing" usually sneak in. Logic errors, like these two, generally do not incur segmentation faults, so one cannot rely on crash scenes to categorize failing cases and fix faults. Instead, the best that a developer can do is to randomly pick one failing case and manually trace its execution. With the `grep` subject program, for example, the developer needs to hunt for the faults in more than 15k lines of code. In the next subsection, we illustrate how R-PROXIMITY can help developers handle this multiple-fault case.

4.2 Analysis Results

We now illustrate the advantages of R-PROXIMITY in the following three subsections, which correspond to the three aspects discussed at the beginning of this section.

```
    static int grep(int fd)
    {
     ...
541  for( ; ; )
542  {
      ...
548   lastnl = bufbeg;
549   if (lastout)     <------- P1
550     lastout = bufbeg;
551   if (buflim - bufbeg == save)
552     break;
553   beg = bufbeg + save - residue + 1; /* fault 1 */
554   for(lim = buflim; lim > beg && lim[-1] != '\n'; --lim)
555     ;
      ...
574   if (beg != lastout)     <------- P2
575      lastout = 0;
576   save = residue + lim - beg;
      ...
580  }
     ...
587  return nlines;
588 }
```

Fault 1: An off-by-one error in grep.c

```
     static char ** comsubs(char* left, char* right)
     {
       ...
2264   for(lcp = left; *lcp != '\0'; ++lcp)
2265   {
2266     len = 0;
2267     rcp = index(right, *lcp);
2268     while (rcp != NULL)
2269     {
2270       for (i = 1; lcp[i] != '\0'/* && lcp[i] == rcp[i] */; ++i) /* fault 2 */     <------- P3
2271         continue;
2272       if (i > len)
2273         len = i;
2274       rcp = index(rcp + 1, *lcp);
2275     }
2276     if (len == 0)
2277       continue;
2278     if ((cpp = enlist(cpp, lcp, len)) == NULL)
2279       break;
2280   }
2281   return cpp;
2282 }
```

Fault 2: A subclause-missing error in dfa.c

Figure 3: Two Injected Faults

4.2.1 Failure Grouping

In order to show the difference between R-Proximity and T-Proximity, we visualize the pair-wise proximity between the 136 failing cases in proximity graphs, using multidimensional scaling (MDS) techniques [1]. MDS techniques take the pair-wise distances between n points, and try to present them in a much lower dimensional (here 2-D) space while preserving the original pair-wise distances. When two points overlap in the proximity graph, it does *not* mean they originally have a zero distance. Instead, it only suggests that the distance between them is too small for MDS to visualize at a given scale.

Figure 4 shows the proximity graphs for the 136 failing cases with R-Proximity (left) and T-Proximity (right), and the middle figure is the close-up of the rectangular region in the left figure. The red crosses and blue circles represent the failing cases in F_1 and F_2, respectively, and the red cross bounded by a diamond symbol denotes the composite ranking. Ideally, blue circles should cluster together and be away from red crosses.

As we can see from the left figure, the failing cases in F_2 do form a cluster under R-Proximity, and the cluster is also away from most red crosses. In comparison, the grouping with T-Proximity is not as dense as that with R-Proximity: The blue circles stretch in a line. This indicates that failing cases due to the same fault can exhibit quite divergent behaviors, which explicitly undermines the hypothesis that T-Proximity relies on. On the other hand, because Sober is nevertheless an alternative for manual investigation, the "due-to" function cannot be precisely uncovered. In the left figure, a number of red crosses are near to the blue cluster, and the close-up observation even shows that a red cross is inside the blue cluster. This impurity is understandable because one cannot expect an automated tool to perform as accurately as human developers. Moreover, as we can see in the following, this impurity does not impact the usage of R-Proximity.

4.2.2 Guided Assignment of Failing Traces

Besides providing denser groupings, R-Proximity also facilitates the assignment of failing cases to responsible developers. With R-Proximity, a cluster of failing cases naturally suggests the fault location through the top predicates most rankings agree on. For brevity in what follows, let us denote the 21 cases in the ellipse as C_1 (Cluster 1), and the 112 cases in the rectangular region as C_2.

Ranks	Filename	Line Num.	Predicate
P_1	`grep.c`	549	`(lastout) == true`
P_2	`grep.c`	574	`(beg != lastout) == true`
P_3	`dfa.c`	2270	`(lcp[i] != '\0') == true`

Table 1: Three Fault-relevant Predicates

A glance over the top predicates of the 21 rankings in C_1 immediately identifies the fault location. Specifically, the predicates P_1 and P_2 (see Table 1 and Figure 3) appear as the top-2 predicates in 17 of the 21 rankings. Therefore, predicates P_1 and P_2 are the clear indicator of fault locations. As a result, these 21 failing cases should be assigned to developers responsible for the "`grep.c`" file, together with suggestions on possible fault locations.

Similarly, by examining the top predicates of the 112 rankings in C_2, we find that the predicate P_3 is the highest in 44 of the 112 rankings, and the second most frequent predicate appears as the highest in only 8 rankings. This renders the predicate P_3 as a good indicator of the fault location. In consequence, one can assign the 112 failing cases to the developers who are in charge of the "`dfa.c`" file, and tell them that line 2270 likely has some problems. Because 24 of the 112 failing cases are actually due to Fault 1, we see that the impurity does not dilute the significance of P_3 being fault indicator.

The remaining three cases are left unassigned because not all failing cases need to be assigned at one time, especially when only a small number of cases are left. When some faults are fixed based on assigned cases, some unassigned cases may no longer fail. Therefore, with R-Proximity, we identify groups of failing cases with clear indication of the fault location, and assign them to responsible developers accordingly. In contrast, because no fault information is available with clusters under T-Proximity, developers need to manually investigate, and assign failing cases accordingly.

4.2.3 Interpret Statistical Debugging

We now illustrate how R-Proximity helps developers interpret the debugging result from Sober, and locate the two faults in Figure 3. According to the instrumentation schema in Section 2, the `grep` subject is instrumented with 1732

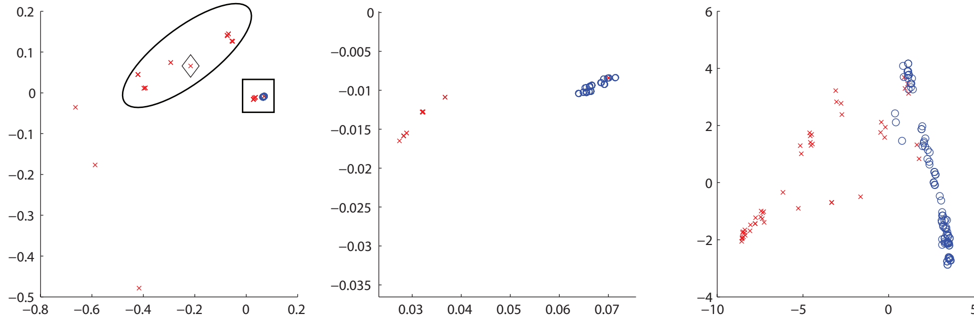


Figure 4: Proximity Graphs with R-Proximity, close-up of the rectangular region and with T-Proximity

boolean and 1404 **return** predicates. A first run of SOBER on the 136 failing and 334 passing cases produces a predicate ranking τ, whose top three predicates are P_1, P_2 and P_3 in order. As we can see, the predicates P_1 and P_2 point to the faulty function, and P_1 is only 4 lines above the real fault location. The predicate P_3, on the other hand, directly points to the *exact* location of the second fault. Although SOBER is pretty accurate in this case, without a *proper* failing case, developers may have difficulty in figuring out why the three predicates are related to the fault(s).

With R-PROXIMITY, a failing case whose induced predicate ranking mostly agrees with τ can be easily found. In this case, the 10th failing case f_{10}, which corresponds to the 122nd test input, is the nearest one to τ under R-PROXIMITY, and in τ_{10}, P_1 and P_2 are the top two predicates. We now can explain why P_1 and P_2 are ranked high based on the execution of f_{10}.

In the execution of f_{10}, the evaluation bias of predicate P_1 is 0.09. In contrast, the evaluation bias is above 0.9286 in all passing cases. So it seems that the evaluation of P_1 is abnormally biased to `false`. Similarly, we find that the evaluation of P_2 is abnormally biased to `true` evaluations. By tracing the execution of f_{10}, we find that normally `beg` is expected to be equal to `lastout` at line 574. However, with the "`+1`" added at line 553, `beg` is no longer equal to `lastout` for most cases. In consequence, the predicate P_2 tends to evaluate `true`, and this makes the variable `lastout` frequently reset to 0 at line 575. This reset of `lastout` finally causes the predicate P_1 at line 549 to evaluate as `false`. Therefore, based on a proper case f_{10}, the localization result in Table 1 is interpreted in a concrete way, which can finally guide developers to fix the fault.

In this case, f_{10} is a *proper* failing case that developers can base debugging on. However, such proper cases cannot be found under T-PROXIMITY, because there is no way one can judge whether a failing trace will suggest the same fault predicates as τ. As an alterative, people may wonder whether a randomly-chosen failing case would similarly work. The answer is less likely. First, a random failing case may be due to a different fault, other than the one suggested by τ. In this example, one has a probability of 0.65 (88/136) to pick up a failing case due to Fault 2, whereas the top predicates in τ are about Fault 1. Furthermore, even if a failing case due to Fault 1 is selected, it may still fail to explain τ. In this example, there are 25 failing cases in F_1 whose evaluation bias of P_1 is 1. In these cases, P_1 exhibits no abnormal symptoms, which renders these cases inappropriate for debugging. In fact, the observation that not all failing cases are equally effective for debugging holds in general, and it conforms well to our debugging experience: In manual debugging, even for the same fault, some failing cases are easy to trace whereas others can be pretty unwieldy.

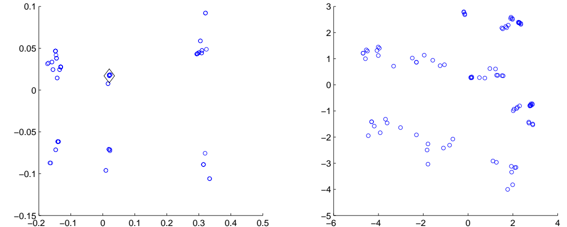


Figure 5: Proximity Graphs with R-Proximity (left) and T-Proximity (right)

After fixing the first fault, a second run of SOBER with the 88 failing and 382 passing cases puts P_3 at the top. Figure 5 plots the proximity graphs for the 88 failing cases with R-PROXIMITY (left) and T-PROXIMITY (right) respectively. Similarly, with R-PROXIMITY, a proper failing case is easily found, and tracing it immediately clears the second fault.

4.3 Time Efficiency

Because SOBER needs to be invoked with every failing case, people may wonder whether computing the R-PROXIMITY distance will take a long time. In fact, since SOBER only carries numerical computation, it is lightweighted even when invoked multiple times. For example, it takes 13.6 seconds to fingerprint the 136 failing cases in the first round, and takes 9.9 seconds for the 88 failing cases in the second round.

Besides fingerprinting, the calculation time of the pairwise distance is also negligible. In the first round, predicate rankings are projected into a subspace of 33 predicates, and the calculation of the pair-wise distance with R-PROXIMITY and T-PROXIMITY takes 0.345 and 0.101 seconds, respectively. In the second round, the projected subspace has 26 predicates, and the time for R-PROXIMITY and T-PROXIMITY distance is 0.245 and 0.042 seconds. All the above time was recorded on a 3.2GHz Intel Pentium-4 PC with 1GB physical memory, running Fedora Core 2.

Subjects	Versions	LOC	Failings	Excluded
schedule	9	397	4–294	4
schedule2	9	299	2–65	2
print_tokens	7	539	6–186	4
print_tokens2	10	489	3–518	0
tot_info	23	398	2–199	6
replace	31	507	3–309	10

Table 2: Characteristics of Subject Programs

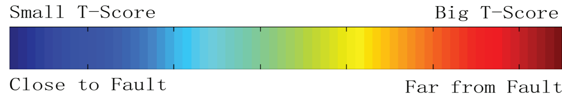


Figure 7: Colorbar for T-score Representation

5. PROPERTY EXPLORATION

In this section, we explore the properties of R-Proximity with a set of subject programs, and verify whether the observations from the previous case study hold in general. The Siemens suite [6] is taken as the subject, which consists of 130 faulty versions of seven subject programs. The program tcas is excluded due to its small code size as the authors did in [7]. Moreover, as we are interested in the grouping property of failing cases under R-Proximity and T-Proximity, versions with less than 30 failing cases are also excluded. Table 2 shows the characteristics of the six subject programs used in this study. The fourth column lists the range of failing cases each subject program has across different versions, and the fifth column shows how many versions are excluded due to the 30-failing-case cutoff. As a result, 63 versions are used in this experimental study. In Section 5.1, we study the correlation between R-Proximity and T-Proximity, which indicates that R-Proximity is a new and different proximity definition from T-Proximity. After that, we examine the design of the weighted Kendall's tau distance in Section 5.2, which demonstrates the necessity of both predicate selection and weighting. Finally, Section 5.3 summarizes this experiment study.

5.1 Proximity Correlation

For R-Proximity to be a new proximity measure, we need to verify that no strong correlation exists between R-Proximity and T-Proximity. For this reason, we calculate the proximity correlation between R-Proximity and T-Proximity for each of the 63 faulty versions under study. Specifically, for each faulty version, the distance between each pair of failing cases is calculated under R-Proximity and T-Proximity respectively, and then the Pearson's correlation coefficient ρ between the two kinds of distances is obtained. Across the 63 versions under study, only one version has a correlation larger than 0.9, and ρ is smaller than 0.7 in 50 of the 63 versions. Therefore, the correlation between these two proximities is weak. This weak correlation is expected because R-Proximity uses Sober to fingerprint failing traces into rankings, and then calculates the weighted Kendall's tau distance between rankings. In comparison, T-Proximity calculates the Euclidean distance based on failing traces directly.

The weak correlation between R-Proximity and T-Proximity is also reflected on the proximity graphs. Figure 6 presents the proximity comparison on six faulty versions, one from each subject program.

Subjects	min-T	med-T	max-T
schedule V4	0.0062	0.0099	0.066
schedule2 V5	0.0085	0.0106	0.018
print_tokens V6	0.0649	0.9337	0.9982
print_tokens2 V4	0.0362	0.2999	0.9328
tot_info V23	0.1242	0.1863	0.8376
replace V13	0.0052	0.0069	0.0373

Table 3: Characteristics of Programs in Figure 6

An overview of Figure 6 reveals that failing cases tend to form clusters under R-Proximity, but are loosely scattered under T-Proximity. Such loose scattering should not be shrugged off by saying that since failing cases are due to the same fault, they are *not* supposed to form separate clusters. In fact, as we have seen in the case study, not all failing cases due to the same fault are equally appropriate for debugging. In more general cases, because a fault can be triggered with different inputs, the fault can incur abnormal behaviors at various program locations. Figure 6, as well as the case study, indicates that failing cases that incur abnormality at different locations *cannot* be distinguished under T-Proximity.

In order to verify that clusters under R-Proximity do suggest different fault locations, we calculate the T-score for each of the failing cases. The T-score was defined in [13] to evaluate the fault localization quality. Intuitively, it measures how far away the suggested fault location is from the real fault location. For clear visual inspection, points are colored based on their T-scores, according to the colorbar shown in Figure 7. Basically, all T-scores in a faulty version are linearly mapped to colors in the colorbar, with the minimum and maximum T-scores corresponding to the blue and the red colors. The minimum, median and maximum T-scores of the six programs are listed in Table 3. In this way, failing cases suggesting similar fault locations should be colored by similar colors.

From the coloring information, we can immediately see that, for all the six subject programs shown in Figure 6, clusters under R-Proximity are in consistent colors. In contrast, points of different colors are mixed together under T-Proximity. This contrast indicates that failing cases that exhibit divergent behaviors can suggest similar fault locations, which reaffirms our observation from the case study.

5.2 Properties of Weighted Kendall's Distance

In Section 3.4, we introduced the weighted Kendall's tau distance as the distance measure for the R-Proximity. The weighted Kendall's tau distance performs predicate selection and predicate weighting simultaneously with the mixed weighting schema (Eq. 5). Here, we study whether predicate selection and weighting are indeed necessary and critical. The effect of those parameters in Eq. 5 is also illustrated in this section.

5.2.1 Is Predicate Selection Critical?

In Eq. 5, setting $\alpha = 0$ and $k_1 = L$ selects all predicates. Figure 8 shows the grouping contrast when $k_1 = 24$ versus when $k_1 = 10$ on schedule V4. Specifically, Figure 8(a) shows the grouping when no predicates are excluded. As we

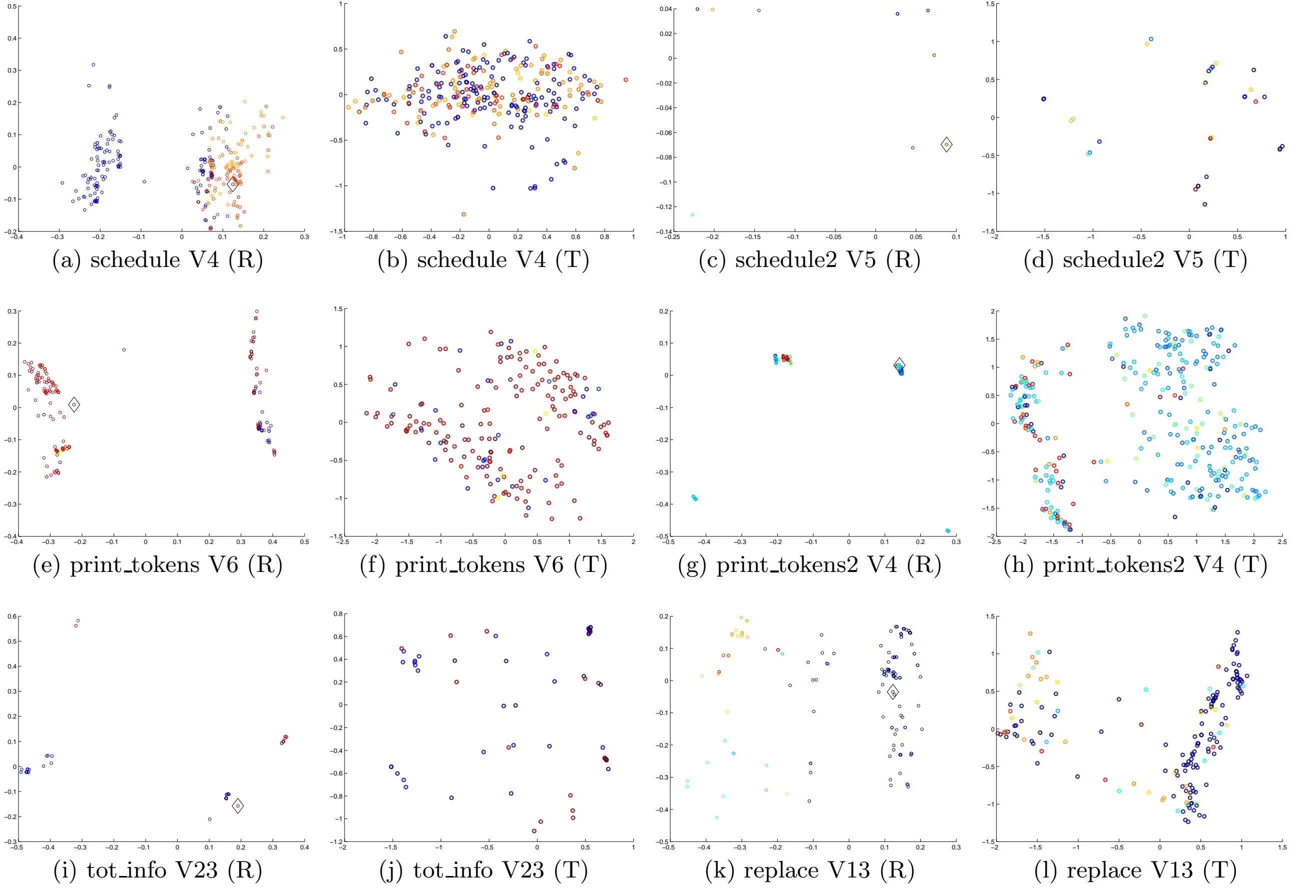


Figure 6: Grouping Contrast between R-Proximity (R) and T-Proximity (T) on Six Subject Programs

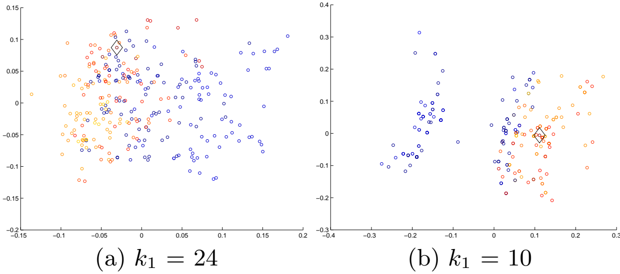


Figure 8: Effect of Predicate Selection

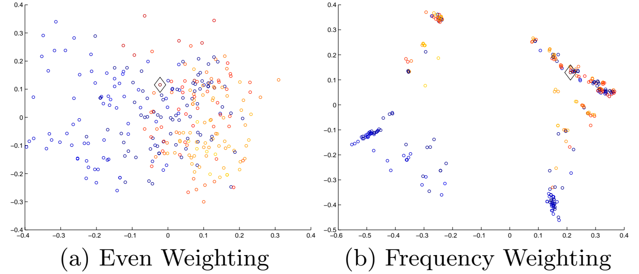


Figure 9: Effect of Predicate Weighting

can see, no clear grouping of the failing cases exists. On the other hand, when the top-10 predicates of τ are selected, cases are well separated with consistent colors. Therefore, predicate selection is critical for R-Proximity to group together failing cases suggesting similar faults.

5.2.2 Is Predicate Weighting Critical?

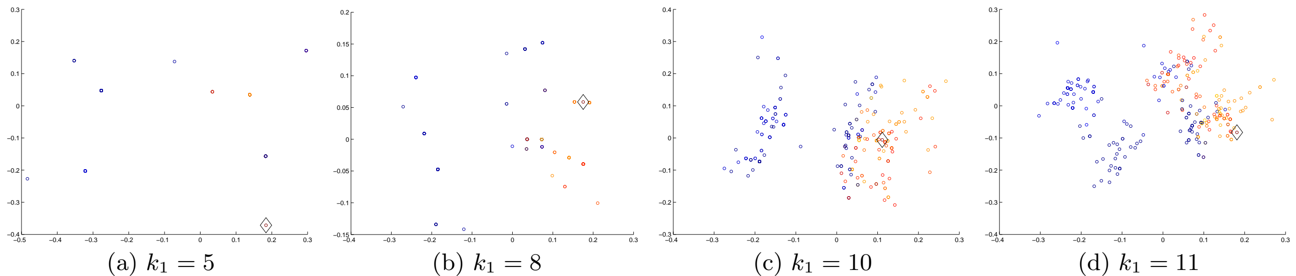
The frequency weighting implemented by Eq. 4 is also critical for proper groupings of failing cases under R-Proximity. The intuition of frequency weighting is that predicates favored by more individual rankings should speak louder in deciding the distances between failing cases.

Specifically, we set $\alpha = 1$ to focus on the effect of frequency weighting in W_2, and Figure 9 shows the contrast. In Figure 9(b), the predicate weights are calculated by Eq. 4, whereas in Figure 9(a), the above calculated weights are wiped even. As we can see, without predicate weighting, failing cases again are mixed together.

5.2.3 Parameter Effects

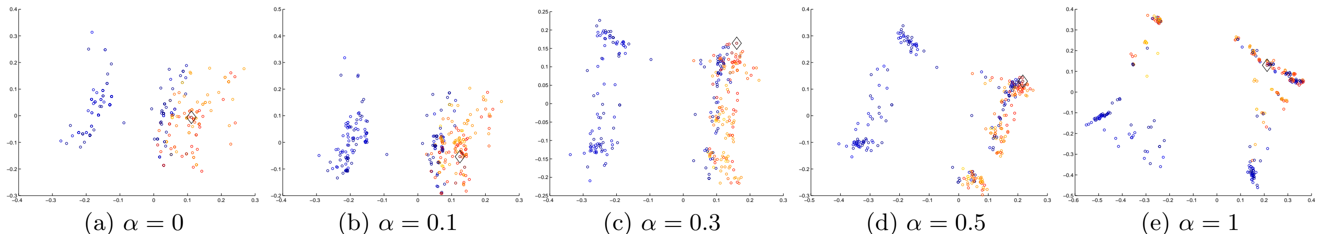
Now that both predicate selection and weighting are shown critical, we now examine how the grouping of failing cases evolves when parameters k_1 and α vary.

With α set as 0, Figure 10 shows how the grouping of



(a) $k_1 = 5$ (b) $k_1 = 8$ (c) $k_1 = 10$ (d) $k_1 = 11$

Figure 10: Grouping of Failing Cases with Different k_1



(a) $\alpha = 0$ (b) $\alpha = 0.1$ (c) $\alpha = 0.3$ (d) $\alpha = 0.5$ (e) $\alpha = 1$

Figure 11: Grouping of Failing Cases with Different α

the failing cases of schedule V4 changes when k_1 varies from 5 to 11. When $k_1 = 5$, these cases are represented by nine points because different rankings become identical when they are projected onto a small subspace. When k_1 increases, more details about the proximity between failing cases become available, and the grouping becomes stable at a certain stage, like $k_1 = 10$ in this case. Certainly, when k_1 reaches the maximum L, the grouping boundary is blurred, as shown in Figure 8(a).

We now examine how the balancing parameter α affects the grouping of failing cases. When α is small, the predicate weight is mainly determined by the composite ranking τ. In consequence, more points tend to cluster around τ. Figure 11 shows how the grouping changes when α varies from 0 to 1. As one can notice, the diamond point, which denotes τ, moves to the group edge from inside when α increases. Because one usage of R-Proximity is to help find a failing case that explains τ for debugging, a small value of α, like 0.1, is generally preferred.

5.3 Experiment Summary

Experiments in subsection 5.1 clearly indicate that R-Proximity is a different proximity measure from T-Proximity, and failing cases tend to form dense and meaningful clusters with R-Proximity. This observation reaffirms our conclusion drawn from the case study in Section 4. The study in subsection 5.2 suggests that both predicate selection and weighting are necessary for the weighted Kendall's tau distance to measure the semantic similarity between failing cases. As a final note, although only proximity graphs for schedule V4 are provided in Section 5.2, the conclusions drawn should hold in general.

6. DISCUSSIONS

Here we discuss the related work and potential threats to validity.

6.1 Related Work

First, this work is closely related to fault localization. Due to the high cost of manual debugging, numerous fault localization techniques have been developed recently, like Delta Debugging [18] and its derivatives [3, 7, 15], NN [17], Liblit05 [12], Sober [13] and Tarantula [9]. This work relates to fault localization in the following ways. First, a statistical debugging tool (Sober) is used in this study to fingerprint failing traces into predicate rankings, on which R-Proximity is then defined. Conventionally, automated debugging tools are used for fault localization purpose *only*. To the best of our knowledge, this is the first piece of work that uses statistical debugging to fingerprint executions. We believe that the idea of representing failing traces by the fault they each suggest is compatible with other fault localization techniques as long as a proper distance is defined between localization results. Second, we demonstrate that R-Proximity can in turn help developers interpret statistical debugging results, which alleviates a plaguing problem of statistical debugging.

Moreover, this study also relates to failing case exploration. Dickinson et al. propose a technique called *cluster filtering* to assist developers in finding failing cases from a set of mostly passing executions [4, 5]. Later on, Podgurski et al. report a study on clustering failure reports [16]. In these studies, T-Proximity is used to assess the execution similarity. In comparison, in this paper, we propose R-Proximity as another type of failure proximity, which is shown to be more suitable for characterizing the semantic proximity between failures. Moreover, similar to previous work [4, 5, 16], multidimensional scaling (MDS) techniques are used in this study to present the failure proximity. However, as the composite ranking τ can be visualized *with other executions* under R-Proximity, our visualization also provides a convenient means for developers to find a proper case for debugging. In previous studies, fault localization results

are not visualized *together with failing traces*. The TARANTULA [9] tool visualizes the fault-relevance of each program statement, but its visualization does not help developers find a proper case to debug.

Finally, our work also relates to the analysis of rank data [14]. In practice, many kinds of data, especially those involving opinions and judgements like merchandize preferences and political elections, are represented as rank data. In this study, we fingerprint failing cases into predicate rankings, and this is the first time failing traces are represented as rank data. In consequence, some interesting questions can be explored. For example, in the future, we will investigate whether more accurate fault localization can be achieved by aggregating individual rankings. Furthermore, this work introduces a weighted form of the Kendall's tau distance, while the classic Kendall's tau distance is a well-studied metric for rankings [10]. We prove its validity as a metric, and demonstrate the critical role weighting plays in R-PROXIMITY.

6.2 Threats to Validity

There are a number of threats to the validity of the case study and experiments. First, although the two faults in the grep subject program mimic realistic "off-by-one" and "subclause-missing" errors, they are nevertheless injected by our authors. For this reason, more case studies on large programs with multiple real faults need to be performed in the future. In general, we expect that similar results will be observed because the effectiveness of R-PROXIMITY depends on the fault localization quality, and statistical debugging has been shown capable of locating real faults in large programs [11–13]. Nonetheless, more experiments are needed to prove or disprove this expectation. Second, hand-crafted test inputs, rather than operational traces collected from the field, are used in this study. In general, operational failing traces are more different from each other. As T-PROXIMITY relies on the literal trace similarity, divergent traces will render T-PROXIMITY less effective in grouping failing traces due to the same fault. In the case study, we have observed that divergent traces can be fingerprinted into similar predicate rankings by SOBER, but it is yet unknown whether similar things will happen with operational traces. Finally, the case study illustrates how R-PROXIMITY helps developers understand and utilize the statistical debugging result, but the ultimate evaluation should be carried out with end-users. However, due to the difficulty (and expense) of controlled user study, most fault localization researches are currently evaluated by the authors [9, 11–13, 17, 19].

7. CONCLUSION

In this paper, we proposed R-PROXIMITY, as an alternative to T-PROXIMITY, to assess the proximity between failing traces. We reason and experimentally validate that with R-PROXIMITY, failing traces due to the same fault tend to be grouped together, but not with T-PROXIMITY. In addition, R-PROXIMITY features some exclusive advantages over T-PROXIMITY. For example, we show that R-PROXIMITY can help developers understand and utilize the statistical debugging result. A number of interesting topics merit further study, for example, it is interesting to explore whether more accurate localization results can be achieved by aggregating the predicate rankings from each failing trace.

Acknowledgements

We would like to thank Gregg Rothermel for making the Siemens suite available, and GrammaTech Inc. for its free copy of the CODESURFER software. We greatly appreciate the anonymous reviewers who offered us insightful suggestions on future work. Finally, we thank Steven Lauterburg and Jesus DeLaTorre for their proofreading of the manuscript.

8. REFERENCES

[1] I. Borg and P. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, first edition, 1996.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA'04*.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE'05*.

[4] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE'01*.

[5] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *ESEC/FSE'01*.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[7] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE'05*.

[8] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely collected program execution data. In *ESEC/FSE'05*.

[9] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE'05*.

[10] M. Kendall and J. D. Gibbons. *Rank Correlation Methods*. Edward Arnold, 1990.

[11] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *PLDI'03*.

[12] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI'05*.

[13] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *ESEC/FSE'05*.

[14] J. I. Marden. *Analyzing and Modeling Rank Data*. Chapman & Hall/CRC, first edition, 1996.

[15] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *ICSE'06*.

[16] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE'03*.

[17] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE'03*.

[18] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE'02*.

[19] X. Zhang, R. Gupta, and N. Gupta. Locating faults through automated predicate switching. In *ICSE'06*.