Reusing Debugging Knowledge via Trace-based Bug Search

Zhongxian Gu Earl T. Barr Drew Schleck Zhendong Su
Department of Computer Science, University of California, Davis
{zgu,etbarr,dtschleck,su}@ucdavis.edu

Abstract

Some bugs, among the millions that exist, are similar to each other. One bug-fixing tactic is to search for similar bugs that have been reported and resolved in the past. A fix for a similar bug can help a developer understand a bug, or even directly fix it. Studying bugs with similar symptoms, programmers may determine how to detect or resolve them. To speed debugging, we advocate the systematic capture and reuse of debugging knowledge, much of which is currently wasted. The core challenge here is how to search for similar bugs. To tackle this problem, we exploit semantic bug information in the form of execution traces, which precisely capture bug semantics. This paper introduces novel tool and language support for semantically querying and analyzing bugs.

We describe OSCILLOSCOPE, an Eclipse plugin, that uses a bug trace to exhaustively search its database for similar bugs and return their bug reports. OSCILLOSCOPE displays the traces of the bugs it returns against the trace of the target bug, so a developer can visually examine the quality of the matches. OSCILLOSCOPE rests on our bug query language (BQL), a flexible query language over traces. To realize OSCILLOSCOPE, we developed an open infrastructure that consists of a trace collection engine, BQL, a Hadoop-based query engine for BQL, a trace-indexed bug database, as well as a web-based frontend. OSCILLOSCOPE records and uploads bug traces to its infrastructure; it does so automatically when a JUnit test fails. We evaluated OSCILLOSCOPE on bugs collected from popular open-source projects. We show that OSCILLOSCOPE accurately and efficiently finds similar bugs, some of which could have been immediately used to fix open bugs.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Design, Languages, Reliability

Keywords OSCILLOSCOPE, reusing debugging knowledge

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

1. Introduction

Millions of bugs have existed. Many of these bugs are similar to each other. When a programmer encounters a bug, it is likely that a similar bug has been fixed in the past. A fix for a similar bug can help him understand his bug, or even directly fix his bug. Studying bugs with similar causes, programmers may determine how to detect or resolve them. This is why programmers often search for similar, previously resolved, bugs. Indeed, even finding similar bugs that have not been resolved can speed debugging.

We theorize that, in spite of the bewildering array of applications and problems, limitations of the human mind imply that a limited number of sources of error underlie bugs [17]. In the limit, as the number of bugs in a bug database approaches all bugs, an ever larger proportion of the bugs will be similar to another bug in the database. We therefore hypothesize that, against the backdrop of all the bugs programmers have written, unique bugs are rare.

Debugging unites detective work, clear thinking, and trial and error. If captured, the knowledge acquired when debugging one bug can speed the debugging of similar bugs. However, this knowledge is wasted and cannot be reused if we cannot search it. The challenge is to efficiently discover similar bugs. To answer this challenge, this paper employs traces to precisely capture the semantics of a bug. Informally, an execution trace is the sequence of operations a program performs in response to input. Traces capture an abstraction of a program's input/output behavior. A bug can be viewed as behavior that violates a program's intended behavior. Often, these violations leave a footprint, a manifestation of anomalous behavior (Engler *et al.* [5]) in a program's stack or execution trace.

This paper introduces novel tool and language support to help a programmer accurately and efficiently identify similar bugs. To this end, we developed OSCILLOSCOPE, an Eclipse plugin, for finding similar bugs and its supporting infrastructure. At the heart of this infrastructure is our bug query language (BQL), a flexible query language that can express a

wide variety of queries over traces. The OSCILLOSCOPE infrastructure consists of 1) a trace collector, 2) a trace-indexed bug database, 3) BQL, 4) a query engine for BQL, and 5) web and Eclipse user interfaces. OSCILLOSCOPE is open and includes the necessary tool support to facilitate developer involvement and contribution.

The OSCILLOSCOPE database contains and supports queries over both stack and execution traces. Stack traces are less precise but cheaper to harvest than execution traces. We quantify this precision trade-off in Section 4.4. When available, stack traces can be very effective, especially when they capture the program point at which a bug occurred [3, 28]. Indeed, a common practice when bug-fixing is to paste the error into a search engine, like Google. Usually, the error generates an exception stack trace. Anecdotally, this tactic works surprisingly well, especially with the errors that novices make when learning an API. OSCILLOSCOPE generalizes and automates this practice, making systematic use of both execution and stack traces.

To validate our hypothesis that unique bugs are rare, we collected 877 bugs from the Mozilla `Rhino` and Apache `Commons` projects. We gave each of these bugs to OSCILLOSCOPE to search for similar bugs. We manually verified each candidate pair of similar bugs that OSCILLOSCOPE reported. If we were unable to determine, within 10 minutes, that a pair of bugs was similar, *i.e.* knowing one bug a programmer could easily fix the other, we conservatively deemed them dissimilar. Using this procedure, we found similar bugs comprise a substantial portion of these bugs, even against our initial database: $\frac{273}{877} \approx 31\%$. OSCILLOSCOPE finds duplicate bug reports as a special case of its search for similar bugs; while duplicates comprised 74 of the 273 similar bugs, however, the majority are *nontrivially similar*. These bugs are field bugs, not caught during development or testing, and therefore less likely to be similar, a fact that strengthens our hypothesis. When querying unresolved bugs against resolved bugs in the `Rhino` project, OSCILLOSCOPE matches similar bugs effectively, using information retrieval metrics we precisely define in Section 4.1. Of the similar bugs OSCILLOSCOPE returns, 48 of the could have been immediately used to fix open bugs.

Finding bug reports similar to an open, unresolved bug promises tremendous practical impact: it could reuse the knowledge of the community to speed debugging. Linus' Law states "given enough eyeballs, all bugs are shallow." An effective solution to the bug similarity problem will help developers exploit this precept by allowing them to reuse the eyes and minds behind past bug fixes. OSCILLOSCOPE has been designed and developed to this end.

We make the following main contributions:

- We articulate and elaborate the vision that most bugs are similar to bugs that have already been solved and take the first steps toward a practical tool built on traces that validates and shows the promise of this vision;
- We present OSCILLOSCOPE a tool that uses traces to find similar bugs and reuse their debugging knowledge to speed debugging;
- We have developed an open infrastructure for OSCILLOSCOPE, available at `http://bql.cs.ucdavis.edu`, comprising trace collection, a trace-indexed bug database, the bug query language BQL, a Hadoop-based query engine, and web-based and Eclipse plugin user interfaces; and
- We demonstrate the utility and practicality of our approach via the collection and study of bugs from the Apache `Commons` and Mozilla `Rhino` projects.

```
1  DynaBean myBean = new LazyDynaBean();
2  myBean.set("myDynaKey", null);
3  Object o = myBean.get("myDynaKey");
4  if ( o == null )
5    System.out.println(
6      "Expected result."
7    );
8  else
9    System.out.println(
10     "What actually prints."
11   );
```

Figure 1: When a key is explicitly bound to **`null`**, `LazyDynaBean` does not return **`null`**.

2. Illustrating Example

Programmers must often work with unfamiliar APIs, sometimes under the pressure of a deadline. When this happens, a programmer can misuse the API and trigger cryptic errors. The following section describes a real example.

Java programmers use getter and setter methods to interact with Java beans. To handle dynamic beans whose field names may not be statically known, Java provides `Reflection` and `Introspection` APIs. Because these APIs are hard to understand and use, the Apache `BeanUtils` project provides wrappers for them. `BeanUtils` allows a programmer to instantiate a `LazyDynaBean` to set and get value lazily without statically knowing the property name of a Java bean, as on line 2 of Figure 1. When an inexperienced programmer used `LazyDynaBean` in his project, he found, to his surprise, that, even though he had explicitly set a property to **`null`**, when he later retrieved the value from the property, it was not **`null`**. Figure 1 shows sample code that exhibits this bug: executing it prints "What actually prints.", not "Expected result.".

Since this behavior was quite surprising to him, the programmer filed bug `BeanUtils-342` on March 21, 2009. Five months later, a developer replied, stating that the observed behavior is the intended behavior. In Figure 2, `LazyDynaBean`'s `get` method consults the internal map `values` on line 9. If the result is **`null`**, the `get` method first calls the method `createOtherProperty`, which by default calls `createProperty` to instantiate and return an empty object. In the parameter list of `createOtherProperty`, `get` calls `getDynaProperty`, which returns `Object.`**`class`** on a name

```
1  public Object get(String name) {
2    if (name == null) {
3      throw new IllegalArgumentException(
4        "No property name specified"
5      );
6    }
7
8    // Value found
9    Object value = values.get(name);
10   if (value != null) {
11     return value;
12   }
13
14   // Property doesn't exist
15   value = createProperty(
16     name,
17     dynaClass.getDynaProperty(
18       name
19     ).getType()
20   );
21
22   if (value != null) {
23     set(name, value);
24   }
25
26   return value;
27 }
```

Figure 2: `LazyDynaBean.get(String name)` from revision r295107, Wed Oct 5 20:35:31 2005.

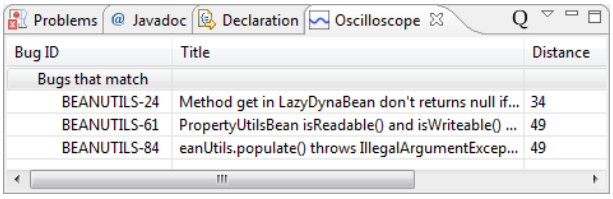

Figure 3: OSCILLOSCOPE returns bug reports similar to `BeanUtils-342`.

Figure 4: Snapshot of the bug report for `BeanUtils-24`.

bound to **`null`**. He did, however, suggest a workaround: subclass `LazyDynaBean` and override its `createOtherProperty` method to return **`null`** when passed `Object.`**`class`** as its type parameter. This in turn would cause `LazyDynaBean.get()` to return **`null`** at line 26, the desired behavior.

How could OSCILLOSCOPE have helped the programmer solve this problem? Assuming the OSCILLOSCOPE database had been populated with traces from the `BeanUtils` project, a programmer would use OSCILLOSCOPE to look for bugs whose traces are similar to the trace for her bug, then return their bug reports. Then she would examine those bug reports to look for clues to help her understand and fix her bug. Ideally, she would find a fix that she could adapt to her bug.

Bug `BeanUtils-342` is the actual bug whose essential behavior Figure 1 depicts. To use OSCILLOSCOPE to search for bugs similar to `BeanUtils-342`, a developer can first issue a predefined query. When a developer does not yet know much about their current bug, a predefined query that we have found to be particulary effective is the "suffix query". This query deems two bugs to be similar when the suffixes of their traces can be rewritten to be the same; its effectiveness is due to the fact that many bugs terminate a program soon after they occur. When a developer specifies the suffix length and edit distance and issues the suffix query to search for bugs similar to `BeanUtils-342`, OSCILLOSCOPE returns the bug reports in Figure 3. The first entry is `BeanUtils-24`, where the `get` method of `LazyDynaBean` did not return **`null`** even when the property was explicitly set to **`null`**.

OSCILLOSCOPE executes the suffix query by computing the edit distance of the suffix of `BeanUtils-342`'s trace against the suffix of each trace in its database. Here is the tail of the method call traces of `BeanUtils-342` and `BeanUtils-24`, the closest bug OSCILLOSCOPE found:

BeanUtils-342	BeanUtils-24
...	...
LazyDynaBean set	BasicDynaClass setProperties
LazyDynaBean isDynaProperty	DynaProperty getName
LazyDynaClass isDynaProperty	DynaProperty getName
LazyDynaClass getDynaProperty	LazyDynaBean isDynaProperty
DynaProperty getType	LazyDynaClass isDynaProperty
LazyDynaBean createProperty	LazyDynaClass getDynaProperty
LazyDynaBean createOtherProperty	LazyDynaBean get
LazyDynaBean set	LazyDynaBean createProperty
DynaProperty getType	LazyDynaBean set
LazyDynaBean class$	DynaProperty getType

Each method call in these two traces is an event; informally, OSCILLOSCOPE looks to match events, in order, across the two traces. Here, it matches the two calls to `isDynaProperty` followed by `getDynaProperty`, then the calls to `get` and `set`. Intuitively, the distance between these two traces is the number of method calls one would have to change to make the traces identical.

Figure 4 is the snapshot of the bug report of `BeanUtils-24`. The same developer who replied to `BeanUtils-342` had also replied to `BeanUtils-24` four years earlier. From his fix to `BeanUtils-24`, the fix for `BeanUtils-342` is immediate. With the help of OSCILLOSCOPE, the programmer could have solved the bug in minutes, instead of possibly waiting five months for the answer. This example shows how OSCILLOSCOPE can help a programmer find and reuse the knowledge embodied in a bug report to fix an open bug.

3. Design and Realization of OSCILLOSCOPE

This section introduces the key components of OSCILLOSCOPE: its user-level support for trace-based search for similar bugs, its bug query language, and core technical issues we overcame to implement it.

3.1 User-Level Support

To support trace-based search for similar bugs, OSCILLOSCOPE must harvest traces, allow users to define bug similarity either by selecting predefined queries or by writing custom queries, process those queries to search a trace-indexed database of bug reports, display the results, and present a user interface that makes this functionality easy to use. Figure 5 depicts the architecture of OSCILLOSCOPE that supports these tasks.

Eclipse Plugin Most developers rely on an integrated development environment (IDE); to integrate OSCILLOSCOPE smoothly into the typical developer's tool chain and workflow, especially to complement the traditional debugging process, we built OSCILLOSCOPE as an Eclipse plugin. OSCILLOSCOPE also supports a web-based user interface, described in a tool demo [9].

OSCILLOSCOPE automates the instrumentation of buggy programs and the uploading of the resulting traces. When a developer who is using OSCILLOSCOPE encounters a bug and wants to find the bug reports similar to her current bug, she tells OSCILLOSCOPE to instrument the buggy code, then re-triggers the bug. OSCILLOSCOPE automatically uploads the resulting trace to its trace-indexed database of bug reports.

Predefined Queries OSCILLOSCOPE is equipped with predefined queries; in practice, users need only select a query and specify that query's parameters, such as a regular expression over method names or an edit distance bound, *i.e.* a measure of the cost of writing one trace into another which Section 3.2.2 describes in detail. Since bugs often cause a program to exit quickly, we have found that the suffix query, (introduced in Section 2) which compares short suffixes of traces with a modest edit distance bound, to be quite effective. Section 4.1 describes how we discovered and validated this suffix query. The bulk of OSCILLOSCOPE's predefined queries, like the suffix query, have simple, natural semantics. Once the buggy trace has been uploaded, the developer can allow OSCILLOSCOPE to automatically use the last selected query to search for similar bugs and return their bug reports, or select a query herself. OSCILLOSCOPE's query engine, which is based on Hadoop, performs the search.

When a JUnit test fails, these steps occur automatically in the background: OSCILLOSCOPE instruments and reruns the test, uploads the resulting test to the database, then, by default, issues a predefined suffix query to search for similar bugs and returns the results. This feature is especially valuable as it targets bugs that occur during development and are more likely to be similar to other bugs than field bugs which have evaded regression testing, inspection and analysis to escape into deployment. To speed response time, OSCILLOSCOPE returns partial results as soon as they are available. Users refresh to see newer results. OSCILLOSCOPE can also visually compare traces to help developers explore and understand the differences and similarities between two traces.

To find the bug report with the fix to the bug in our illustrating example in Section 2, the developer would direct OSCILLOSCOPE to instrument the buggy code, re-triggered the bug, then issued the default suffix query. Behind the scenes, OSCILLOSCOPE would harvest and upload the trace, then execute the query and display the resolved bug report with the relevant fix.

Trace Granularity Using the OSCILLOSCOPE, programmers can trace code at statement or method granularity, as show in Figure 6. Statement-granular traces allow OSCILLOSCOPE to support the search for *local* bugs, bugs whose behavior is contained within a single method, and facilitates matching bugs across different projects that run on the same platform and therefore use the same instruction set architecture. OSCILLOSCOPE supports coarser granularity traces, such as replacing a contiguous sequence of method call on a single class with that class name, by post-processing.

Chop Points Instrumentation is expensive, so we provide OSCILLOSCOPE provides a chop operator that allows programmers to set a *chop point*, a program point at which tracing starts or ends. A programmer can also use chop points to change the granularity of tracing from method calls to statements. By restricting tracing to occur within a pair of chop points, chopping enables the scalability of fine-grained, *i.e.* statement-granular, tracing. When debugging with OSCILLOSCOPE, a developer will resort to setting chop points to collect a partial trace for their current bug when collecting a complete trace is infeasible. Knowing where to place chop points involves guesswork; its payoff is the ability to query the OSCILLOSCOPE database. Thus, the OSCILLOSCOPE database contains partial traces and traces that may contain events at different levels of granularity.

In a world in which OSCILLOSCOPE has taken hold, developers will routinely upload traces when they encounter a bug. To this end, we have setup an open bug database, available at our website, and welcome developers to contribute both the buggy traces and the knowledge they acquired fixing the bugs that caused them. Our tool is not just for students and open source developers. We have made the entire OSCILLOSCOPE framework privately deployable, so that companies can use OSCILLOSCOPE internally without having to share their bug database and worry about leaking confidential information.

We have posited that, when two bugs may share a root cause, this fact manifests itself as a similarity in their traces. Prior to this region of similarity, of course, the two traces might be arbitrarily different. Internally, OSCILLOSCOPE relies on BQL, its bug query language, and the insight and ingenuity of query writers to write BQL queries that isolate the essence of a bug; these queries define bug similarity

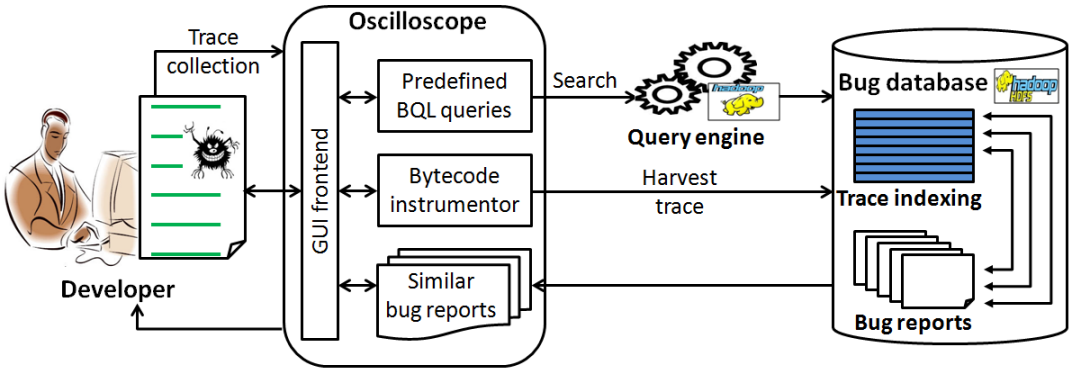

Figure 5: Debugging with the OSCILLOSCOPE Framework.

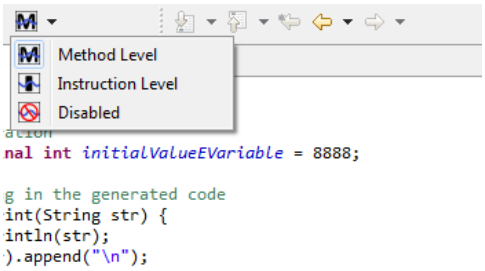

Figure 6: Selecting execution trace granularity in the OSCILLOSCOPE Eclipse plugin.

and drive OSCILLOSCOPE's search for similar bugs. Thus, OSCILLOSCOPE rests on BQL, which we describe next.

3.2 BQL: A Bug Query Language

A bug is behavior that violates a user's requirements. The core difficulty of defining a bug more precisely is that different users, even running the same application, may have different requirements that change over time. In short, one user's bug can be another's feature. To tackle this problem, we have embraced expressivity as the central design principle of BQL; it allows writing of queries that embody different definitions of buggy behavior. To support such queries, our database must contain program behaviors. Three ways to capture behavior are execution traces, stack traces, and vector summarizations of execution traces. An execution trace is a sequence of events that a program emits during execution and precisely captures the behavior of a program on an input. Collecting execution traces is expensive and they tend to be long, mostly containing events irrelevant to a bug. Stack traces encode execution events keeping only the current path from program entry to where a bug occurs. Finally, one can summarize an execution trace into a vector whose components are a fixed permutation of a program's method calls. For example, the summarization of trace *ABA* into a vector whose components are ordered alphabetically is $\langle 2,1\rangle$, which discards the order of events. In Section 4.4 we quantify the loss of precision and recall these approaches entail. Therefore, we define bug similarity and support queries for bug data, such as a fix, in terms of execution traces.

3.2.1 Terminology

T denotes the set of all traces that a set of programs can generate. Each trace in **T** captures an execution of a program (We discuss our data model in more detail in Section 3.2.3). B is the set of all bugs. The bug $b \in B$ includes the afflicted program, those inputs that trigger a bug and the traces they induce, the reporter, dates, severity, developer commentary, and, ideally, the bug's fix. The function $t : B \to 2^{\mathbf{T}}$ returns the set of traces that trigger b. The set of all unresolved bugs U and the set of all resolved bugs R partition B. We formalize the ideal bug similarity in the oracle ϕ. For $b_0, b_1 \in B$,

$$\text{similar}(b_0,b_1) = \begin{cases} \text{T} & \text{if } b_0 \text{ is similar to } b_1 \text{ wrt } \phi \\ \text{F} & \text{otherwise} \end{cases} \quad (1)$$

which we use to define, for $b \in B$,

$$[\![b]\!] = \{x \in B \mid \text{similar}(b,x)\}, \quad (2)$$

the set of all bugs that are, in fact, similar to b.

For $b_0, b_1 \in B$, and the query processing engine Q,

$$\text{match}(b_0,b_1) = \begin{cases} \text{T} & \text{if } b_0 \text{ is similar to } b_1 \text{ wrt } Q \\ \text{F} & \text{otherwise} \end{cases} \quad (3)$$

which we use to define

$$[b] = \{x \in B \mid \text{match}(b,x)\}, \quad (4)$$

the set of all bugs that the query for b returns.

Ideally, we would like $[\![b]\!] = [b]$, but for debugging it suffices that we are 1) *sound*: $\text{match}(b_i,b_j) \Rightarrow \text{similar}(b_i,b_j)$, and 2) *relatively complete*: $\text{similar}(b_i,b_j) \Rightarrow \exists b_k \in [\![b_j]\!] - \{b_i\}\ \text{match}(b_i,b_k)$, for any $b_i \neq b_j$. In Section 4, we demonstrate the extent to which we use traces to achieve these goals.

Both $[\![b]\!]$ and $[b]$ are reflexive: $\forall b \in B,\ b \in [\![b]\!] \wedge b \in [b]$, which means that $\{b\} \subseteq [\![b]\!] \cap [b] \neq \emptyset$. We are often interested in bugs similar to b other than b itself, so we also define

$$U_b = [b] \cap U - \{b\} \quad \text{unresolved bugs that match } b \quad (5)$$

$$R_b = [b] \cap R - \{b\} \quad \text{resolved bugs that match } b. \quad (6)$$

$$
\begin{aligned}
\langle query \rangle ::= \ & \textbf{SELECT } \langle bug \rangle^{+} [\ \textbf{FROM } \langle db \rangle^{+}] \\
& \textbf{WHERE } \langle cond \rangle \ \ [\textbf{DISTANCE } \langle distance \rangle] \\
\langle db \rangle ::= \ & \texttt{X} \mid \textbf{ALL} \\
\langle cond \rangle ::= \ & \langle cond \rangle \ \textbf{\&\&} \ \langle cond \rangle \mid \langle cond \rangle \ \textbf{||} \ \langle cond \rangle \mid \textbf{(} \ \langle cond \rangle \ \textbf{)} \\
& \mid \textbf{INTERSECT?(}\langle bug \rangle, \langle pat \rangle [, d] [, n]\textbf{)} \\
& \mid \textbf{JACCARD?(}\langle bug \rangle, \langle pat \rangle, t [, d]\textbf{)} \\
& \mid \textbf{SUBSET?(}\langle bug \rangle, \langle pat \rangle [, d]\textbf{)} \\
\langle bug \rangle ::= \ & \texttt{Traces} \mid \textbf{[} len \textbf{]} \langle bug \rangle \mid \langle bug \rangle \textbf{[} len \textbf{]} \\
& \mid \textbf{PROJ(}\langle bug \rangle, \texttt{S}\textbf{)} \\
\langle pat \rangle ::= \ & \sigma \mid \langle bug \rangle \mid \langle pat \rangle \ \textbf{|} \ \langle pat \rangle \mid \langle pat \rangle^{*} \mid \textbf{(} \ \langle pat \rangle \ \textbf{)}
\end{aligned}
$$

Figure 7: The syntax of BQL: **X** is a project; for the bug b, $\texttt{Traces} = t(b)$; σ is an event; and S is a set of events.

3.2.2 The syntax of BQL

Figure 7 defines the syntax of BQL, which is modeled after the standard query language SQL. The query "**SELECT** b **FROM ALL WHERE INTERSECT?**(b, "getKeySet")" returns all bugs whose traces have a nonempty intersection with the set of all traces that call the getKeySet method. The clause **FROM** Project1, Project2 restricts a query to bugs in Project1 or Project2. The terminal **ALL** removes this restriction, and is the default when the **FROM** clause is omitted.

Predicates In addition to the standard Boolean operators, BQL provides **SUBSET?**, **INTERSECT?**, and **JACCARD?** predicates to allow a programmer to match a bug with those bugs whose traces match the pattern, when the target bug's traces are a subset of, have a nonempty intersection with, or overlap with those traces. For example, "**SELECT** b **FROM ALL WHERE SUBSET?**(b,b_{527})" returns those bugs whose traces are a subset of b_{527}'s traces.

Traces may differ in numerous ways irrelevant to the semantics of a bug. For example, two traces may have taken different paths to a buggy program point or events with the same semantics may have different names. Concrete execution traces can therefore obscure semantic similarity, both cross-project and even within project. As a first step toward combating the false negatives this can cause, BQL allows two traces to differ in Levenshtein edit distance within a bound. The Levenshtein distance of two strings is the minimum number of substitutions, deletions and insertions of a single character needed to rewrite one string into another. The Levenshtein distance of 011 and 00 is 2 ($011 \rightarrow 01 \rightarrow 00$).

The application of edit distance to a bug's traces generates a larger set of strings. Thus, BQL adds the distance parameter d to its set predicates to bound the edit distance used to produce traces during a similarity search. Edit distance relaxes matching and can introduce false positives. To combat this source of imprecision, the **INTERSECT?** operator also takes n, an optional constraint that specifies the minimum number of a bug's traces that must be rewritten into one of the target bug's traces. For example, assume b_{527} contains multiple traces that trigger an assertion failure and a programmer wants to search for other bugs with multiple traces. The program could use the predicate **INTERSECT?**(b, b_{527}, 50, 3), which forms the set of pairs of traces $t(\texttt{b}) \times t(b_{527})$ and is true if the members of at least 3 of these pairs can be rewritten into one another using 50 edits.

Operators When we know enough about the problem domain or salient features of our bug, we may wish to restrict where traces match. The terminal behavior of a buggy program, embodied in the suffix of its execution trace, often captures a bug's essential features. Or we may wish to consider only those traces in which the application initialized in a certain fashion and restrict attention to prefixes. Thus, BQL provides prefix and suffix operators. These operators use array bracket notation and return the specified length prefix or suffix.

A programmer may wish to project only a subset of events in a trace. For example, when searching for bugs similar to b_{527}, a developer may want to drop methods in the log package to reduce noise. To accomplish this task, he writes

```
SELECT bug FROM ALL WHERE SUBSET?(
    PROJ(bug, "read,write,close"),
    b527, 10
)
```

where "read,write,close" names the only methods in the trace in which we are interested.

Patterns The last line of Figure 7 defines the BQL's pattern matching syntax. Here, the terminals are either (through the $\langle bug \rangle$ the rule), $t(bug)$, the set of traces that trigger a bug, or $\sigma \in \Sigma_x$, a symbol (*i.e.* event) in a trace. Patterns that mix symbols from different event alphabets can succinctly express subsets of traces. For instance, a query writer may wish to find bugs that traverse the class c_a on the way to the method m_1 in c_b and then execute some method in the class c_c. The pattern $c_a m_1 c_c$ achieves this. As a concrete example, consider a developer who wishes to find all traces that invoke methods in the class PropertyConfiguration (abbreviated PC in the query below) before invoking the method escapeJava in StringEscapeUtils; this generates the query

```
SELECT bug FROM Configuration WHERE
    INTERSECT?(bug, "PC escapeJava").
```

3.2.3 Semantics

BQL rests on a hierarchy of disjoint alphabets of execution events, shown in Figure 8. At the lowest level, execution events are instructions. An instruction symbol encodes an opcode and possibly its operands; a method symbol encodes a method's name and possibly its signature and parameters. Sequences of instructions define statements in the source language; sequences of statements define basic blocks whose sequences define a path through a method. Thus, a project

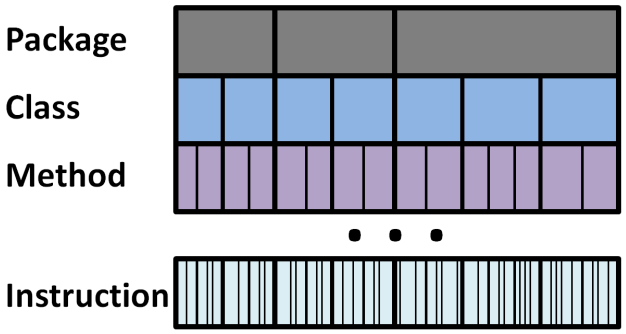

Figure 8: Hierarchy of trace event alphabets.

$$[\![\textbf{SELECT } (b_1,\cdots,b_n) \textbf{ FROM } (\texttt{db}_1,\cdots,\texttt{db}_m) \textbf{ WHERE } \texttt{cond}]\!]$$
$$= \{(b_1,\cdots,b_n) \mid (b_1,\cdots,b_n) \in [\![\texttt{cond}]\!]\, l \cap \bigcup_{i\in[1,m]} [\![\texttt{db}_i]\!]\}$$
$$[\![\textbf{SELECT } (b_1,\cdots,b_n) \textbf{ FROM } (\texttt{db}_1,\cdots,\texttt{db}_m) \textbf{ WHERE } \texttt{cond}$$
$$\textbf{DISTANCE } \texttt{dist}]\!]$$
$$= \{(b_1,\cdots,b_n) \mid (b_1,\cdots,b_n) \in [\![\texttt{cond}]\!][\![\texttt{dist}]\!] \cap \bigcup_{i\in[1,m]} [\![\texttt{db}_i]\!]\}$$
$$[\![\texttt{db}]\!] = \begin{cases} X & \text{if } \texttt{db} = \mathsf{X} \subset \mathbf{T} \\ \mathbf{T} & \text{if } \texttt{db} = \texttt{ALL} \end{cases}$$
$$[\![\texttt{dist}]\!] \in \{h,l,u\}$$
$$[\![\texttt{cond}_1 \texttt{ \&\& } \texttt{cond}_2]\!] = \lambda\delta.[\![\texttt{cond}_1]\!]\delta \wedge [\![\texttt{cond}_2]\!]\delta$$
$$[\![\texttt{cond}_1 \texttt{ || } \texttt{cond}_2]\!] = \lambda\delta.[\![\texttt{cond}_1]\!]\delta \vee [\![\texttt{cond}_2]\!]\delta$$
$$[\![\texttt{(cond)}]\!] = [\![\texttt{cond}]\!] = \lambda\delta.[\![\texttt{cond}]\!]\delta$$

Figure 9: Semantics of queries and Boolean operators.

$$[\![p]\!] = \mathscr{L}(p) \subseteq T \qquad [\![p^*]\!] = [\![p]\!]^*$$
$$[\![p \mid p]\!] = [\![p]\!] \cup [\![p]\!] \qquad [\![(\ p\ )]\!] = [\![p]\!]$$

Figure 10: Semantics of the pattern operators.

defines a sequence of languages of traces defined over each alphabet in its hierarchy.

Formally, a project defines a disjoint sequence of alphabets $\Sigma_i, i \in \mathbb{N}$ where Σ_1 is the instruction alphabet. Let $\mathscr{L}(P@\Sigma)$ denote the language of traces the project P generates over the event alphabet Σ. Then, for the project P and its alphabets, each symbol in a higher level language defines a language in P's lower-level languages: $\forall\sigma \in \Sigma_{i+1}, \mathscr{L}(\sigma) \subseteq \mathscr{L}(P@\Sigma_i)$. Traces can mix symbols from alphabets of different abstraction levels, so long as there exists an instruction-level translation of the trace t such that $t \in \mathscr{L}(P@\Sigma_1)$.

The semantics of BQL is mostly standard. Figure 9 straightforwardly defines queries and the standard Boolean operators. It defines $[\![\texttt{dist}]\!]$ as a distance function and uses lambda notation to propagate its binding to the BQL's set predicates. The semantics of patterns in Figure 10 are standard; the $\mathscr{L}$, used to define pattern semantics, is the classic language operator.

$$[\![\textbf{INTERSECT?}(b,p)]\!] = [\![b]\!] \cap [\![p]\!] \neq \emptyset$$
$$[\![\textbf{INTERSECT?}(b,p,d)]\!] = [\![b]\!] \cap_d [\![p]\!] \neq \emptyset$$
$$[\![\textbf{INTERSECT?}(b,p,n)]\!] = |[\![b]\!] \cap [\![p]\!]| \geq n$$
$$[\![\textbf{INTERSECT?}(b,p,d,n)]\!] = |[\![b]\!] \cap_d [\![p]\!]| \geq n$$
$$[\![\textbf{JACCARD?}(b,p,t)]\!] = \frac{|[\![b]\!] \cap [\![p]\!]|}{|[\![b]\!] \cup [\![p]\!]|} \geq t$$
$$[\![\textbf{JACCARD?}(b,p,t,d)]\!] = \frac{|[\![b]\!] \cap_d [\![p]\!]|}{|[\![b]\!] \cup [\![p]\!]|} \geq t$$
$$[\![\textbf{SUBSET?}(b,p)]\!] = [\![b]\!] \subseteq [\![p]\!]$$
$$[\![\textbf{SUBSET?}(b,p,d)]\!] = \forall x \in [\![b]\!], \exists y \in [\![p]\!]\ \ \delta(x,y) \leq d$$

Figure 11: Semantics of the intersect, Jaccard and subset predicates.

$$[\![\texttt{Traces}]\!] \in 2^{\mathbf{T}}$$
$$[\![\textbf{PROJ}(b,\mathsf{S})]\!] = [\![b]\!]|_{\mathsf{S}}$$
$$[\![[\ len\ ]b]\!] = \{\alpha \mid \exists\beta\ \ \alpha\beta \in [\![b]\!] \wedge |\alpha| = len\}$$
$$[\![b[\ len\ ]]\!] = \{\beta \mid \exists\alpha\ \ \alpha\beta \in [\![b]\!] \wedge |\beta| = len\}$$

Figure 12: Semantics of the trace operators.

Set Predicates and Edit Distance In Figure 9, $[\![b]\!] = [\![\texttt{Traces}]\!] \in 2^{\mathbf{T}}$ and the edit distance function $[\![\texttt{dist}]\!]$ has signature $\Sigma^* \times \Sigma^* \to \mathbb{N}$. The set of allowed edit distance functions is $\{h,l,u\}$. In this set, h denotes Hamming, l denotes Levenshtein (the default) and u denotes a user-specified edit distance function. BQL allows query writers to specify a distance function to give them control over the abstraction and cost of a query. For instance, a query writer may try a query using Hamming distance. If the results are meager, he can retry the same query with Levenshtein distance, which matches more divergent traces.

For $\delta \in \{h,l,u\}$, the function $\cap_d : 2^{\Sigma^*} \times 2^{\Sigma^*} \times \mathbb{N} \to 2^{\Sigma^*}$ is

$$X \cap_d Y = \{z \mid\ z \in X, \exists y \in Y\ \delta(y,z) \leq d \\ \vee z \in Y, \exists x \in X\ \delta(x,z) \leq d\} \tag{7}$$

and constructs the set of all strings in X or Y that are within the specified edit distance of an element of the other set. We use $\cap_d$ to define **JACCARD?** and **INTERSECT?** in Figure 11. For **JACCARD?**, the Jaccard similarity must meet or exceed $t \in [0,1]$; for **INTERSECT?**, the cardinality of the set formed by $\cap_d$ must meet or exceed $n \in \mathbb{N}$.

A user-defined distance function u may be written in any language so long as it matches the required signature. One could define distance metric that reduces a pair of method traces to sets of methods then measure the distance of those sets in terms of the bags of words extracted from the method names, identifiers or comments. Alternatively, one could

define a Jaccard measure over the sets of methods or classes induced by two traces, scaling the result into $\mathbb{N}$.

Trace Operators To specify the length of prefixes and suffixes in Figure 12, we use $len \in \mathbb{N}$. In the definition of **PROJ**, proj_i is the projection map from set theory and $S \subseteq \Sigma_x$, *i.e.* S is a subset of symbols from one of the event abstraction alphabets. BQL's concrete syntax supports regular expressions as syntactic sugar for specifying S.

3.3 Implementation

Four modules comprise OSCILLOSCOPE: a bytecode instrumentation module, a trace-indexed database, a query processing engine, and two user interfaces. The instrumentation module inserts recording statements into bytecode. The instrumentation module is built on the ASM Java bytecode manipulation and analysis framework. For ease of smooth interaction with existing workflows, our database has two forms: a standalone database built on Hadoop's file system and an trace-based index to URLs that point into an existing Bugzilla database. The OSCILLOSCOPE plug-in for Eclipse supports graphically comparing traces. A challenge we faced, and partially overcame, is that of allowing the comparison of traces visually regardless of their length. An interesting challenge that remains is to allow a user to write, and refine, queries visually by clicking on and selecting portions of a displayed trace. The web-based UI is AJAX-based and used Google's GWT.

Internally, traces are strings with the syntax

$$
\begin{array}{rcl}
\langle \textit{Trace} \rangle & ::= & \langle \textit{Event} \rangle \mid \langle \textit{Event} \rangle \langle \textit{Trace} \rangle \\
\langle \textit{Event} \rangle & ::= & \langle \textit{Method} \rangle \mid \langle \textit{Instruction} \rangle \\
\langle \textit{Method} \rangle & ::= & \textbf{M FQClassName MethodName Signature} \\
 & \mid & \textbf{S FQClassName MethodName Signature} \\
\langle \textit{Instruction} \rangle & ::= & \textbf{I OPCODE } \langle \textit{Operands} \rangle \\
\langle \textit{Operands} \rangle & ::= & \textbf{OPERAND} \mid \textbf{OPERAND } \langle \textit{Operands} \rangle
\end{array}
$$

An example method event follows

```
M org/apache/commons/beanutils/LazyDynaClass \
getDynaProperty (Ljava/lang/String;) \
Lorg/apache/commons/beanutils/DynaProperty;.
```

Here, **M** denotes an instance method (while **S** in the syntax denotes a static method). The fully qualified class name follows it, then the method name, and finally the method signature. To capture method events, we inject logging into each method's entry. For statements, we inject logging into basic blocks. To produce coarser-grained traces, we post-process method-level traces to replace contiguous blocks of methods in a single class or package with the name of the class or package.

Query Engine The overhead of query processing lies in two places: retrieving traces and comparing them against the target trace. For trace comparison, we implemented an optimized Levenshtein distance algorithm [10]. To scale to large databases (containing millions of traces), OSCILLOSCOPE's query engine is built on top of Apache Hadoop, a framework that allows for the distributed processing of large data sets across clusters of computers. The essence of Hadoop is MapReduce, inspired by the map and reduce functions commonly used in functional programming. It enables the processing of highly distributable problems across huge datasets (petabytes of data) using a large number of computers. The "map" step divides an application's input into smaller sub-problems and distributes them across clusters and "reduce" step collects the answers to all the sub-problems and combines them to form the output.

OSCILLOSCOPE's query processing is an ideal case for MapReduce, since it compares all traces against the target trace. This comparison is embarrassingly parallelizable: it can be divided into sub-problems of comparing each trace in isolation against the target trace. Each mapper processes a single comparison and the "reduce" step collects those bug identifiers bound to traces within the edit distances bound to form the final result. Section 4 discusses the stress testing we performed for OSCILLOSCOPE against millions of traces.

3.4 Extending OSCILLOSCOPE with New Queries

OSCILLOSCOPE depends on experts to customize its predefined queries for a particular project. These experts will use BQL and its operators to write queries that extract trace subsequences that capture the essence of a bug. Learning BQL itself should not be much of a hindrance to these experts, due to its syntax similarity to SQL and its reliance on familiar regular expressions. To further ease the task of writing queries, OSCILLOSCOPE visualizes the difference of two traces returned by a search and supports iterative query refinement by allowing the query writer to edit the history of queries he issued. To write effective queries, an expert will, of course, need to know her problem domain and relevant bug features; she will have to form hypotheses and, at times, resort to trial and error. The payoff for a query writer, and especially for an organization using OSCILLOSCOPE, is that, once written, queries can be used over and over again to find and fix recurring bugs. Across different versions of a project, even though methods may change names as a project evolves, OSCILLOSCOPE can still find similar bugs if their signature in a trace is sufficiently localized and enough signposts, such as method names, remain unchanged so that edit distance can overcome the distance created by those that have changed.

Single Regex Queries `Configuration-323` occurred when `DefaultConfigurationBuilder` misinterpreted property values as lists while parsing configuration files. The reporter speculated that the invocation of `ConfigurationUtils.copy()` during internal processing was the cause. To search for bugs in the `Configuration` project that invoke the `copy()` method in the `ConfigurationUtils` class, the reporter could have issued

```
SELECT bug FROM Configuration WHERE
    SUBSET?(bug, "ConfigurationUtils.copy()").
```

The result set contains 272 and 283, in addition to 323. Developers acknowledged the problem and provided a workaround. Thus, the bug 323 can be solved identifying and studying 272 and 283. These bugs predate 323. Here, OSCILLOSCOPE found usefully similar bugs using a query based on a simple regular expression that matched a single method call.

Lang-421 is another example of a bug for which a simple query parameterized on a regular expression over method names would have sped its resolution. In this bug, the method escapeJava() in the StringEscapeUtils class incorrectly escaped the '/' character, a valid character in Java. In Apache Commons, the Configuration project depends on Lang. To find out similar bugs in the Configuration project, we set $\alpha =$ StringEscapeUtils.escapeJava and issue the query

```
SELECT b FROM ALL WHERE INTERSECT?(
    PROJ(b, org/apache/commons/lang/*), 'α')
```

to search for all bugs in the database that match pat. This query returns four bugs. It returns Lang-421, the bug that motivated our search. The second bug is Configuration-408, where forward slashes were escaped when a URL was saved as a property name. Studying the description, we confirmed that StringEscapeUtils.escapeJava() caused the problem. The third bug, Configuration-272, concerns incorrectly escaping the ',' character; the class StringEscapeUtils still exhibits this problem. Manually examining the last bug, Lang-473, confirms that it duplicates Lang-421. It is our experience with bugs like these that led us to add the simple "regex query" to our suite of predefined queries.

3.4.1 Limitations

Overcoming instrumentation overhead is an ongoing challenge for OSCILLOSCOPE. For example, stress-testing FindBugs revealed five-fold slowdown. Our first, and most important, countermeasure is our chop operator, described above in Section 3.1. In our experience, statement-granular tracing would be infeasible without it. Longer term, we plan to employ Larus' technique to judiciously place chop points [18]. Another direction for reducing overhead is to use sampling, then trace reconstruction, as in Cooperative Debugging [19]; the effectiveness of this approach depends, of course, on OSCILLOSCOPE garnering enough participation to reliably acquire enough samples to actually reconstruct traces. Currently, OSCILLOSCOPE handles only sequential programs. To handle concurrent programs, we will need to add thread identifiers to traces and explore the use of vector distance on interleaved traces.

4. Evaluation

This evaluation shows that OSCILLOSCOPE does find similar bugs and, in so doing, finds a generally useful class of suffix-based queries. It measures how OSCILLOSCOPE scales and demonstrates the accuracy of basing search on execution traces.

To evaluate OSCILLOSCOPE, we collected method-level traces and studied bugs reported against the Apache Commons (2005–2010): comprising 624153 LOC and 379163 lines of comment, and Rhino (2001–2010): comprising 205775 LOC and 34741 lines of comment projects. We chose these projects because of their popularity. In most cases, reporters failed to provide a test case to reproduce the bug. Even with a test case, recompiling an old version and reproducing the bug was a manual task that consumed 5 minutes on average. This explains why OSCILLOSCOPE's trace database contains 656 of the 2390 bugs reported against Apache Commons and 221 of the 942 bugs reported against Rhino. For each bug, we recorded related information from the bug tracking system such as source code, the fix (if available), and developer comments. Our database currently contains 877 traces (one trace per bug) and its size is 43.1 MB. The minimum, maximum, mean, and variance of the trace lengths in the database are 2, 50012, 5431.1, and 6.68×10^7.

Experimental Procedure In general, we do not know $[\![b]\!]$, those bugs that are *actually* similar to each other (Equation 2). We manually approximated $[\![b]\!]$ from $[b]$, an OSCILLOSCOPE result set, in two ways. First, we checked whether two bugs shared a common *error-triggering* point, the program point at which a bug first causes the program to violate its specification. We studied *every* candidate pair of bugs that OSCILLOSCOPE reported to be similar for at most 10 minutes. For example, we deemed Rhino-217951 and 217965 to be similar because a Number.toFixed() failure triggered each bug. Second, we recorded as similar any bugs identified as such by a project's developers. For example, a Rhino developer commented in Rhino-443590's report that it looks like Rhino-359651. Given our limited time and knowledge of the projects, we often could not determine whether two bugs are, in fact, similar. When we could not determine similarity, we conservatively deemed the bugs dissimilar. This procedure discovers false positives, not false negatives. To account for false negative, we introduce the relative recall measure in Section 4.1 next. We discuss our methodology's construct validity in Section 4.5.

Using this experimental procedure, we found that similar bugs comprise a substantial portion of bugs we have collected to date: $\frac{273}{877} \approx 31\%$. 74 of the 273 similar bugs are identical, caused by duplicate bug reports; the majority, however, are nontrivially similar. Since we conjecture that the number of ways that humans introduce errors is finite and our database contains field bugs, we expect the proportion of similar bugs to increase as our database grows.

4.1 Can OSCILLOSCOPE Find Similar Bugs?

To show that OSCILLOSCOPE accurately finds similar bugs and to validate the utility of a default query distributed with OSCILLOSCOPE, we investigate the precision and recall of suffix queries issued against the 221 bug traces we harvested from the Rhino project. We queried OSCILLOSCOPE with 48

unresolved Rhino bugs. We found similar bugs for 14 of these bugs, under our experimental procedure. When computing the measures below, we used this result as our oracle for $[\![b]\!]$.

When we do not deeply understand a bug, matching trace suffixes is a natural way to search for bugs, since many bugs cause termination. This insight underlies the suffix query we first introduced in Section 2. For suffixes of length len, the suffix query is

```
SELECT bug FROM Rhino WHERE
    SUBSET?(tbug[len], bug[len], distance).
```

Two parameters control suffix comparison: length and edit distance. We conducted two experiments to show how these parameters impact the search for similar bugs. In the first experiment, we fix the suffix length at 50 and increase the allowed Levenshtein distance. In the second experiment, we fix the Levenshtein distance to 10 and vary the suffix length. Table 1 depicts the results of the first experiment, Table 2 those of the second. We report the data in Table 2 in descending suffix length to align the data in both tables in order of increasing match leniency.

In both experiments, we are interested in the number of queries for unresolved bugs that match a resolved bug, as these might help a developer fix the bug. Recall that R is the set of resolved bugs (Section 3) and R_b (Equation 6) is the set of resolved bugs similar to b. In the second column, we report how many unresolved bugs match any resolved bugs. For this purpose, we define $U_R = \{b \in U \mid R_b \neq \emptyset\}$, then, in column three, we report the percentage of unresolved bugs that match at least one resolved bug.

To show that our result sets are accurate, we compute their average size across all unresolved bugs as

$$\frac{\sum |R_b|}{|U|} = \overline{|R_b|}. \tag{8}$$

In the fourth column, we report this average, then because R might grow to be very large, we report the average cardinality of $|R_b|$ as a percentage of $|R|$. This is the average number of resolved bugs a developer would have to examine for clues he might use to solve an unresolved bug. It is a proxy for developer effort. In this experiment, even the two most lenient matches — Levenshtein distance 30 and 25 length suffixes — do not burden a developer with a large number of potentially similar bugs. For Rhino, the maximum number of bugs returned by OSCILLOSCOPE for a bug is six. The effort to analyze each result set is further mitigated by the fact that OSCILLOSCOPE returns ranked result sets, in order of edit distance.

Next, we report the precision and recall of OSCILLOSCOPE over R, the resolved bugs. When TP denotes true positives, FP denotes false positives, and FN denotes false negatives, recall that

$$\text{precision} = \frac{\text{TP}}{\text{TP}+\text{FP}} \qquad \text{recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}. \tag{9}$$

In our context, we have

$$\text{precision} = \frac{|[\![b]\!] \cap [b]|}{|[b]|} \qquad \text{recall} = \frac{|[\![b]\!] \cap [b]|}{|[\![b]\!]|}. \tag{10}$$

To restrict precision to R, we define

$$\text{respr}(X,b) = \begin{cases} 1 & \text{if } [b] \cap X = \emptyset \\ \frac{|([\![b]\!] \cap [b]) \cap X|}{|[b] \cap X|} & \text{otherwise.} \end{cases} \tag{11}$$

then, in column four, we report the average respr

$$\frac{1}{|U|} \sum_{b \in U} \text{respr}(R,b) \tag{12}$$

which we manually compute via our experimental procedure. The majority, $\frac{34}{48}$, of the unresolved bugs in our Rhino data set are unique. These unique bugs dominate the average respr at lower edit distance and longer suffix length. When distance increases from 10 to 20, and suffix length drops from 50 to 25, the average respr drops dramatically. The reason is that Rhino, a JavaScript interpreter, has a standard exit routine. When a program ends without runtime exceptions, Rhino calls functions to free resources. Most the Rhino traces end with this sequence which accounts for OSCILLOSCOPE's FPs and decreases the average respr. In general, both Table 1 and Table 2 show how the greater abstraction comes at the cost of precision.

Our experimental procedure is manual and may overstate recall because we do not accurately account for FNs; accurately accounting for FNs would require examining all traces. Rather than directly report recall (Equation 10), we over-approximate it with *relative recall*:

$$\text{relrecall}(b) = \begin{cases} 1 & \text{if } [\![b]\!] = \{b\} \\ 1 & \text{if } [\![b]\!] \cap [b] - \{b\} \neq \emptyset \\ 0 & \text{otherwise,} \end{cases} \tag{13}$$

which measures how often OSCILLOSCOPE returns useful results. Relative recall scores one whenever a bug 1) is unique or 2) has at least one truly similar bug. Because $\forall [b], b \in [\![b]\!] \cap [b]$, we need to test these two cases separately to distinguish between returning only b when b is, in fact, unique in the database from returning b, but failing to return other, similar bugs when they exist. In contrast to precision, relative recall does not penalize the result set for FPs; in contrast to recall, it does not penalize the result set for FNs. In practice, relative recall can be manually verified, since we only need to find a counter-example in $[\![b]\!]$ to falsify the first condition and checking the second condition is restricted by $|[b]|$, whose maximum across our data set is 11.

As with precision, we restrict relative recall

$$\text{relrecall}(X,b) = \begin{cases} 1 & \text{if } [\![b]\!] = \{b\} \\ 1 & \text{if } ([\![b]\!] \cap [b] - \{b\}) \cap X \neq \emptyset \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

Distance	$\lvert U_R\rvert$	$\frac{\lvert U_R\rvert}{\lvert U\rvert}$	$\overline{\lvert R_b\rvert}$	$\overline{\frac{\lvert R_b\rvert}{\lvert R\rvert}}$	Average Respr(R,b)	Average Relrec(R,b)	Average F-score(R,b)
0	3	6.3%	0.08	0.05%	0.98	0.75	0.85
10	12	25.0%	0.67	0.39%	0.93	0.94	0.93
20	18	37.5%	1.29	0.75%	0.82	0.96	0.88
30	31	64.6%	2.29	1.32%	0.56	0.98	0.71

Table 1: Measures of the utility of our approach as a function of increasing Levenshtein distance and fixed suffix length 50, $\forall b \in U$; U_R is the subset of the unresolved bugs U for which OSCILLOSCOPE finds a similar resolved bug; $\frac{\lvert U_R\rvert}{\lvert U\rvert}$ is the percentage of unresolved bugs that are similar to a resolved bug; $\overline{\lvert R_b\rvert}$ is the average number of resolved bugs returned for each unresolved bug b; since R will vary greatly in size, we report $\overline{\frac{\lvert R_b\rvert}{\lvert R\rvert}}$, the size of each result set as a percentage of the resolved bugs; the meaning of the measures Respr, Relrec and restricted F-score are defined in the text below.

Suffix Length	$\lvert U_R\rvert$	$\frac{\lvert U_R\rvert}{\lvert U\rvert}$	$\overline{\lvert R_b\rvert}$	$\overline{\frac{\lvert R_b\rvert}{\lvert R\rvert}}$	Average Respr(R,b)	Average Relrec(R,b)	Average F-score(R,b)
200	2	4.2%	0.08	0.05%	0.97	0.73	0.83
100	5	10.4%	0.19	0.11%	0.97	0.79	0.87
50	12	25.0%	0.67	0.39%	0.93	0.94	0.93
25	24	50.0%	1.83	1.06%	0.68	0.96	0.80

Table 2: Measures of the utility of our approach as a function of decreasing suffix length and fixed distance threshold 10, $\forall b \in U$; for a detailed discussion of the meaning of the first four columns, please refer to the caption of Table 1; the remaining columns are defined in the text below.

and, in column five, we report its average,

$$\frac{1}{|U|}\sum_{b\in U}\text{relrecall}(R,b). \tag{15}$$

By definition two paths are different. At edit distance zero, a bug can only match itself, possibly returning duplicate bug reports. Many paths differ only in which branch they took in a conditional and thus OSCILLOSCOPE can match them even with a small edit distance budget, which accounts for the large rise in relative recall moving from an edit distance budget of 0 to 10. Minor differences accumulate when longer suffixes of two traces are compared. The loss of irrelevant detail accounts for the large rise in relative recall moving from suffix length 100 to 50.

The F-score is the harmonic mean of precision and recall. Here we define it using restricted precision and relative recall:

$$\frac{2\cdot\text{relrecall}(X,b)\cdot\text{respr}(X,b)}{\text{relrecall}(X,b)+\text{respr}(X,b)}. \tag{16}$$

Column six in both tables reports this measure.

Unique bugs dominate the first Levenshtein measurement. At distance zero, OSCILLOSCOPE returned five bugs in total. Among the 34 unique bugs, OSCILLOSCOPE found a similar bug for only one of them. At suffixes of length 200, OSCILLOSCOPE returned two bugs in total and both are true positives. The distance zero and suffix length 200 queries are more precise, but also more susceptible to FNs, because they return so few bugs. Correctly identified, unique bugs increase relative recall. With the increase of distance or decrease in suffix length, OSCILLOSCOPE finds more potentially similar bugs, at the cost of FPs. At distance 10 and suffixes of length 50, OSCILLOSCOPE found similar bugs for 11 of the non-unique bugs, and returned only one FP for a unique bug.

These tables demonstrate the utility of OSCILLOSCOPE. Its query results often contain resolved bugs that may help solve the open, target bug. They are small and precise and therefore unlikely to waste a developer's time. The second most strict measurement, edit distance 10 in Table 1 and suffix length 50 in Table 2 is the sweet spot where OSCILLOSCOPE is sufficiently permissive to capture interesting bugs while not introducing too many FP.

The six FPs have similar test cases, but their errors are triggered at different program points. For instance, `Rhino-567484` and `352346` have similar test cases that construct and print an `XML` tree. A problem with `XML` construction triggers `567484`, while an error in `XML.toXMLString()` triggers `352346`. Two of FN occurred because the distance threshold was too low. At distance 20, OSCILLOSCOPE matches these two bugs without introducing FPs. Logging calls, using `VMBridge`, separate these two bugs from their peers and required the additional 10 edits to elide. We failed to find similar bugs for `496540` and `496540` because the calls made in `Number.toFixed()` changed extensively enough to disrupt trace similarity. To reduce our FP and FN rates, we plan to investigate object-sensitive dynamic slicing and automatic α-renaming which, given trace alignment, renames symbols, here method names, in order of their appearance. To handle

FNs like those caused by the interleaving of new calls such as logging, we intend to evaluate trace embedding, *i.e.* determining whether one trace a subsequence of another, as an additional distance metric.

4.2 How Useful are the Results?

In this section, we show how OSCILLOSCOPE can help a programmer fix a bug by identifying bugs similar to that bug. To do so, we issue a suffix query for each bug. The query **Suffix-query**

```
SELECT bug FROM ALL WHERE
    SUBSET?(tbug[50], bug[50], 30)
```

returns a distance-ranked result set for `tbug`. We used trace suffixes of length 50 because OSCILLOSCOPE performed best at this length (Section 4.1). We then manually examined the target bug and the bug in the result set with the smallest distance from the target bug, using our experimental procedure.

As a degenerate case of bug similarity, OSCILLOSCOPE effectively finds duplicate bug reports and helps keep a bug database clean. OSCILLOSCOPE reported 33 clusters of duplicate bug reports, 28 of which the project developers had themselves classified as duplicates. We manually examined the remaining five and confirmed that they are in fact duplicates that had been missed. We have reported these newly-discovered duplicates to the project maintainers. So far, one of them, (`JXPATH-128`, `143`), has been confirmed and closed by the project maintainers. We have yet to receive confirmations on the other four: `Configuration-30` and `256`, `Collections-100` and `128`, `BeanUtils-145` and `151`, and `Rhino-559012` and `573410`. Next, we discuss a selection of interesting, similar bugs returned by **Suffix-query**.

OSCILLOSCOPE is particularly effective at finding API misuse bugs. From the `BeanUtils` project, **Suffix-query** identified as similar `BeanUtils-42`, `117`, `332`, `341`, and `372`, which all incorrectly try to use BeanUtils in conjunction with non-public Java beans. Upon seeing a fix for one of these, even a novice could fix the others. `Configuration-94` and `222` describe bugs that happen when a new file is created without checking for an existing file. `Configuration-94` occurs in `AbstractFileConfiguration.save()` and `222` in the `PropertiesConfiguration` constructor. Their fixes share the same idea: check whether a file exists before creating it. `Lang-118` and `131` violate preconditions of `StringEscapeUtils.unescapeHtml()`. `Lang-118` found that `StringEscapeUtils` does not handle hex entities (prefixed with '0x'), while `131` concerns empty entities. Both fixes check the input: seeing one, a developer would know to write the other. `Lang-59` (resolved) and `Lang-654` (unresolved) describe the problem that `DataUtils.truncate` does not work with daylight savings time. Studying `Lang-564`, we found its cause was a partial (*i.e.* bad) fix of `Lang-59`. Developers of `Lang` confirmed this finding. `BeanUtils-115` is the problem that properties from a `DynaBean` are not copied to a standard Java bean. In `BeanUtils-119`, `NoSuchMethodException` is thrown when setting value to a property with name "aRa". The root cause for both bugs is the passing of a parameter that violates Java bean naming conventions: when the second character of a getter/setter method is uppercase, the first must also be. Again, from a fix for either, the fix for the other is immediate.

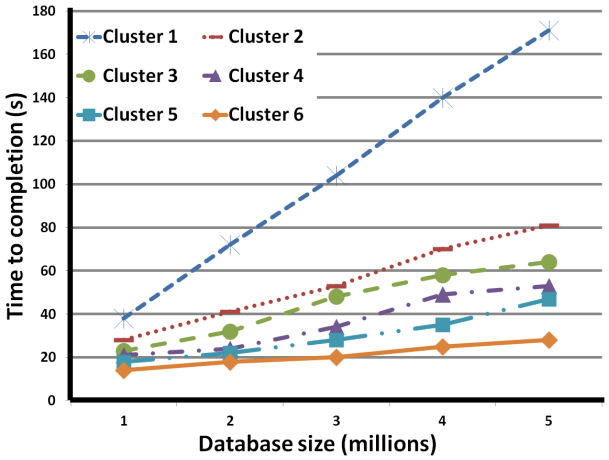

Figure 13: The effect of database size and Hadoop cluster on query time; The six Hadoop cluster settings vary in number of hosts, total memory and number of CPUs, as described in the text.

In addition to identifying bug pairs such as those we just discussed, OSCILLOSCOPE efficiently clusters unresolved bugs. In `Rhino`, OSCILLOSCOPE clustered seven unresolved bugs — `444935`, `443590`, `359651`, `389278`, `543663`, `448443`, and `369860`. These bugs all describe problems with regular expressions. The project developers themselves deemed some of these bugs similar. Problems in processing XML trees causes `Rhino-567484`, `566181` and `566186`. When processing strings, Rhino throws a `StackOverflowException` in both `Rhino-432254` and `567114`.

Although we found interesting, similar bugs comparing only suffix traces, we also learned some of the deficiencies of this strategy. When a bug produces erroneous output and does not throw an exception, the location of the error is usually not at the end of the execution trace. Suffix queries produce FPs on such bugs, especially when they share a prefix because their test cases are similar. The bug `Configuration-6`, improper handling of empty values, and `Configuration-75`, the incorrect deletion of XML subelements, formed one such FP. Suffix comparison can miss similar bugs when a bug occurs in different contexts. The incorrect implementation of `ConfigurationUtils.copy()` caused `Configuration-272`, `283` and `323`. These three bugs differ only in the location of their call to this buggy method. The suffix operator alone failed to detect these bugs as similar, because the context of each call is quite different.

Database Size	Time to completion (s)					
(million)	1, 4GB 4CPU	1, 6GB 8CPU	2, 10GB 12CPU	3, 14GB 16CPU	4, 16GB 20CPU	5, 144GB 38CPU
1	38.2	28.6	23.1	21.1	18.9	14.2
2	72.3	41.3	32.8	24.2	22.5	18.3
3	104.7	53.8	48.4	34.6	28.6	20.2
4	140.7	70.1	58.4	49.4	35.2	25.7
5	171.3	81.2	64.5	53.5	47.9	28.0

Table 3: Measures of OSCILLOSCOPE's query processing time against different sizes of databases with different Hadoop cluster settings; The Hadoop cluster setting "*x*, *y*GB *z*CPU" denotes a cluster with *x* nodes, a total of *y* GB of memory, and *z* CPUs.

4.3 Scalability

Our central hypothesis, that unique bugs are rare against the backdrop of all bugs, means that OSCILLOSCOPE will become ever more useful as its database grows, since it will become ever more likely to find bugs similar to a target bug. This raises performance concern: how much time will it take to process a query against millions of traces. To gain insight into how the size of bug database and Hadoop cluster settings impact the query time, we conducted a two-dimension stress testing. In the first dimension, we fix the database size and increase Hadoop cluster resources. In the second dimension, we fix the Hadoop cluster and vary the database size. We replicated our 877 traces to create a database with millions of traces. We used the **Suffix-query** as the candidate query in the experiment. For each setting, we issued **Suffix-query** five times and computed the average time to completion. Table 3 depicts the results. The first column lists the database size, which ranges from one to five million traces. The rest of the columns show the time to process the query for each each cluster configuration. The Hadoop cluster setting is expressed as the following: "*x*, *y*GB *z*CPU" denotes a cluster with *x* nodes, a total of *y* GB of memory, and *z* CPUs. The settings we used were

Cluster 1	1,	4GB	4CPU
Cluster 2	1,	6GB	8CPU
Cluster 3	2,	10GB	12CPU
Cluster 4	3,	14GB	16CPU
Cluster 5	4,	16GB	20CPU
Cluster 6	5,	144GB	38CPU

Figure 13 shows that the query processing times grow linearly with the increase of database size for all the settings, the expected result which confirms that all the nodes in the cluster were used. Figure 13 also shows that query times drop more dramatically for large database sizes than for small ones. This is because the performance gain gradually overcomes the network latency as the workload grows. Figure 13 clearly demonstrates that OSCILLOSCOPE's query processing will take less time as a function of the nodes in its Hadoop cluster.

4.4 Execution Trace Search Accuracy

Execution traces accurately capture program behavior, but are expensive to harvest and store. To show this cost is worth paying, we compare the execution trace search accuracy against the accuracy of vector and stack trace searches.

Vector Researchers have proposed a vector space model (VSM) to measure the similarity of execution traces [29]. To compare the VSM metric against OSCILLOSCOPE's use of edit distance over execution traces, we repeated the `Rhino` experiment in Section 4.1 using VSM. The VSM approach transforms execution traces to vectors, then computes their distance[1]. In the experiment, we tuned two parameters, trace length and distance threshold of the VSM algorithm to optimize the respr and relrecall results. We varied the trace lengths using suffixes of length 25, 50, 100, 200, and unbounded. For the distance threshold, we tried $[0.5, 1.0]$ in increments of 0.1. Suffix length 50 and distance threshold 0.9 was the sweet spot of the VSM metric where it achieved 0.85 respr and 0.90 relrecall, for an F-score of 0.87. At suffix of length 50 and edit distance 10, OSCILLOSCOPE outperforms VSM, with 0.93 respr and 0.94 relrecall for and 0.93 F-score. We hypothesize the reason is that the VSM model does not consider the temporal order of method invocations. The search for similar bugs requires two operations, search and insertion. VSM requires the recomputation of the vector for every trace in the database whenever an uploaded trace introduces a new method, but it can amortize this insertion cost across searches, which are relatively efficient because they can make use of vector instructions. In contrast, insertion is free for OSCILLOSCOPE, but search operations require edit distance computations.

Are Execution Traces Necessary? The OSCILLOSCOPE database contains both stack and execution traces. Execution traces are more precise than stack traces, but more expensive to harvest. Stack traces are cheaper to collect, but not always available and less precise. For example, a stack trace is usually not available when a program runs to completion but produces incorrect output; they are less precise because they may not capture the program point at which a bug occurred or when they do, they may capture too little of its calling context. Here, we quantify this difference in the precision to shed light on when the effort to harvest execution traces might be justified.

[1] Users of OSCILLOSCOPE, who wish to use VSM's distance function, can implement it as a custom distance function.

70 Rhino bugs in our database have stack traces. 33/70 bugs are similar to at least one other bug, under our experimental procedure. In the experiment, we issued OSCILLOSCOPE queries to search for these similar bug pairs, one restricted to stack traces and the other to execution traces. The stack trace query was

```
SELECT b1, b2 FROM Rhino WHERE SUBSET?(
    PROJ(b1, stack), PROJ(b2, stack)
).
```

In the query, `PROJ(b1, stack)` extracts the stack trace of a bug. We ranked each result set in ascending order by distance. We deemed the closest 50 to "match" under OSCILLOSCOPE, since manually examining the 50 pairs was tractable and each of the 70 bugs appears in at least one of these 50 pairs in both result sets. We examined each of these 50 bug pairs to judge whether it is a TP or an FP. Of the remaining pairs, which we deemed non-matches, we examined all the FN.

The stack trace result set contains 26 TP, 24 FP (the 50 that OSCILLOSCOPE matched), and 6 FN (from among the remaining pairs). The fact that most stack traces are similar accounted for nearly half the FPs. The largest distance over all 50 pairs is only 5, compared with 35 for the execution traces. We examined the stack traces of the 6 FNs. The distance between `Rhino-319614` and `370231` is large because the bugs belong to different versions of `Rhino` between which the buggy method was heavily reformulated. Both `Rhino-432254` and `567114` throw stack overflow exceptions whose traces differ greatly in length. Because `Rhino-238699`'s stack trace does not contain the error-triggering point, `Compile.compile()`, it does not match `299539`. The other three FN have distances 8, 14, and 14, so FPs crowd them out of the top 50 pairs.

The execution trace result set contains 41 TP, 9 FP, and 1 FN. False positives appear when the edit distance exceeds 20. The only FN is the pair `Rhino-319614` and `370231`, which is also a FN in the stack trace result set and for the same reason: the relevant methods were extensively rewritten.

The stack trace result set matched similar bugs with 0.52 respr, 0.81 relrecall (See Section 4.1) and 0.63 F-score; the execution trace result set achieved 0.82 respr, 0.98 relrecall, and 0.89 F-score. These results make clear that the cost of instrumenting and running executable to collect execution traces can pay off, especially as a fallback strategy when queries against stack traces are inconclusive.

4.5 Threats to Validity

We face two threats to the external validity of our results. First, we evaluated OSCILLOSCOPE against bugs collected from the `Rhino` and Apache `Commons` projects, which might not be representative. Second, most of the bug reports we gathered from these projects lack bug-triggering test cases. Without test cases, we were unable to produce traces for almost 70% of the bugs in these projects. Our evaluation therefore rests on the remaining 30%. It may be that those bugs whose reports contain a test case are not representative. These two threats combine to shrink the number of traces over which OSCILLOSCOPE operates.

Nonetheless, our results are promising. We have conjectured that unique bugs are rare, in the limit, as a trace database contains all bug traces. The fact that we have already found useful examples of similar bugs in a small population lends support to our conjecture. Not only is the population small, but it contains only field bugs. In particular, it lacks predeployment bugs, which are resolved during initial development. We contend that these bugs are more likely to manifest common misunderstandings and be more similar than field defects.

Our evaluation is also subject to two threats to its construct validity. First, one cannot know $[\![b]\!]$ in general. We described how we manually approximated $[\![b]\!]$ in our experimental procedure above. Project developers identified 43% of these bugs as similar to another bug. For the remaining 57%, as with any manual process involving non-experts, our assessments may have been incorrect. Second, our database currently contains method, not instruction, granular traces. To the extent to which a method-level trace fails to capture bug semantics, our measurements are inaccurate. Our evaluation data is available at `http://bql.cs.ucdavis.edu`.

5. Related Work

This section opens with a discussion of work that, like OSCILLOSCOPE, leverages programming knowledge. Often, one acquires this knowledge to automate debugging, which we discuss next. We close by discussing work on efficiently analyzing traces.

Reuse of Programmer Knowledge Leveraging past solutions to recurrent problems promises to greatly improve programmer productivity. Most solutions are not recorded. However, the volume of those that are recorded is vast and usually stored as unstructured data on diverse systems, including bug tracking systems, SCM commit messages, and mailing lists. The challenge here is how to support and process queries on this large and unwieldy set of data sources. Each project discussed below principally differs from each other, and OSCILLOSCOPE, in their approach to this problem.

To attack the polluted, semi-structured bug data problem, DebugAdvisor judiciously adds structure, such as constructing bags of terms from documents, and they define a new sort of query, a *fat query*, that unites structured and unstructured query data [1]. In contrast, OSCILLOSCOPE uses execution traces to query bug data. Thus, our approach promises fewer false positives, but may miss relevant bugs, while DebugAdvisor will return larger result sets that may require human processing. Indeed, Ashok *et al.* note "there is much room for improvement in the quality of retrieved results."

When compilation fails, HelpMeOut saves the error message and a snapshot of the source [11]. When a subsequent compilation succeeds, HelpMeOut saves the difference between the failing and succeeding versions. Users can enter compiler errors into HelpMeOut to search for fixes. In con-

trast, OSCILLOSCOPE handles all bug types and compares execution traces, not error messages. Dimmunix, proposed by Jula *et al.*, prevents the recurrence of deadlock [15]. When a deadlock occurs, Dimmunix snapshots the method invocation sequence along with thread scheduling decisions to form a deadlock signature.

Dimmunix monitors program state. If a program's state is similar to a deadlock signature, Dimmunix either rolls back execution or refuses the lock request. Dimmunix targets deadlock bugs, while OSCILLOSCOPE searches for similar bugs across execution traces of all types of bugs.

Code peers are methods or classes that play similar roles, provide similar functionality, or interact in similar ways. A recurring fix is repeatedly applied, with slight modifications, to several code fragments or revisions. Proposed by Ngueyn *et al.*, FixWizard syntactically finds code peers in source code, then identifies recurring fixes to recommend the application of these fixes to overlooked peers [23]. OSCILLOSCOPE uses traces, which precisely capture bug semantics, to search for similar bugs in a database of existing bugs.

Automated Debugging Detecting duplicate bug reports is a subproblem of finding similar bugs. Runeson *et al.* use natural language processing (NLP) to detect duplicates [27]. Comments in bug reports are often a noisy source of bug semantics, but could complement OSCILLOSCOPE's execution trace-based approach. In a recent work, Wang *et al.* augment the NLP analysis of bug reports with execution information [29]. They record whether or not a method is invoked during an execution into a vector. We compare Wang *et al.*'s approach to ours in Section 4.4.

Bug localization and trace explanation aim to find the root cause of a single bug or identify the likely buggy code for manual inspection [2, 4, 7, 8, 12–14, 16, 20, 21, 30–32]. OSCILLOSCOPE helps developers fix a bug by retrieving similar bugs and how they were resolved. OSCILLOSCOPE and bug localization complement each other. A root cause discovered by bug localization may lead to the formulation of precise OSCILLOSCOPE queries for similar, resolved bugs; storing only trace subsequence identified by a root cause can also save space in the database.

Efficient Tracing and Analysis Researchers have proposed query languages over traces for monitoring or verifying program properties [25, 26]. Goldsmith *et al.* propose the Program Trace Query Language (PTQL) for specifying and checking a program's runtime behaviors [6]. A PTQL query instruments and runs a program to check whether or not a specified property is satisfied at runtime. Martin *et al.*'s Program Query Language (PQL) defines queries that operate over event sequences of a program to specify and capture its design rules [22]. PQL supports both static and dynamic analyses to check whether a program behaves as specified. Olender and Osterweil propose a parameterized flow analysis for properties over event sequences expressed in Cecil, a constraint language based on quantified regular expressions (QREs) [24]. These three query languages are designed for analyzing a single, property-specific trace; OSCILLOSCOPE collects raw, unfiltered traces and tackles the bug similarity problem in the form of trace similarity.

6. Conclusion and Future Work

Debugging is hard work. Programmers pose and reject hypotheses while seeking a bug's root cause. Eventually, they write a fix. To reuse this knowledge about a bug, we must accurately find semantically similar bugs. We have proposed comparing execution traces to this end. We have defined and built OSCILLOSCOPE, a tool for searching for similar bugs, and an open infrastructure — trace collection, a flexible query language BQL, a query engine based on Hadoop, database, and both a web-based and plugin user interface — to support it. BQL allows a user to 1) define bug similarity and 2) use that definition to search for similar bugs. We evaluated OSCILLOSCOPE on a collection of bugs from popular open-source projects. Our results show that OSCILLOSCOPE accurately retrieves relevant bugs: When querying unresolved bugs against resolved bugs in the `Rhino` project it achieves 93% respr, 94% relrecall for an F-score of 0.93 (Section 4.1).

OSCILLOSCOPE's database will grow from our activities and the contributions of others. We will use it to study questions such as "Can we quantify the precision-scale trade-off of varying trace granularity?", "What proportion of bugs can only be captured at statement granularity?", and "Which parts of the system do few buggy traces traverse?" We also plan to add the traces of correct executions to OSCILLOSCOPE's database, which currently contains only buggy executions and compare them. We intend to explore adding call context to produce execution trees; because OSCILLOSCOPE is a general and flexible framework, adding this support requires only modifying instrumentation and defining a distance measure over execution trees. In short, we intend to enhance and further experiment with our framework to gain additional insight into "What makes bugs similar?".

OSCILLOSCOPE is an open project. We invite readers to use OSCILLOSCOPE and help us make it more general. Please refer to our website `http://bql.cs.ucdavis.edu` for tutorials and demonstrations.

Acknowledgments

This research was supported in part by NSF (grants 0917392 and 1117603) and the US Air Force (grant FA9550-07-1-0532). The information presented here does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

References

[1] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A recommender system for debugging. In *Proceedings of the 17th Joint Meeting of the European Software Engineering Conference and the ACM SIG-*

SOFT International Symposium on the Foundations of Software Engineering, 2009.

[2] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *Proceedings of the International Conference on Computer Aided Verification*, 2009.

[3] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2008.

[4] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha. Debugging model-transformation failures using dynamic tainting. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, 2010.

[5] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2001.

[6] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[7] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, 2006.

[8] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, 2003.

[9] Z. Gu, E. Barr, and Z. Su. BQL: Capturing and reusing debugging knowledge. In *Proceedings of the 33rd International Conference on Software Engineering (Demo Track)*, 2011.

[10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[11] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems*, 2010.

[12] K. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[13] E. W. Host and B. M. Ostvold. Debugging method names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[15] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[16] S. Kim, K. Pan, and E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006.

[17] A. Ko and B. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.

[18] J. R. Larus. Whole program paths. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[19] B. Liblit. *Cooperative Bug Isolation*. Springer-Verlag, 2007.

[20] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[21] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[22] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[23] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd International Conference on Software Engineering*, 2010.

[24] K. Olender and L. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, 1990.

[25] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, 2011.

[26] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[27] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[28] A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

[29] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[30] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[31] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 1999.

[32] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*, 2006.