CoREBench: Studying Complexity of Regression Errors

Marcel Böhme[*]
Saarland University, Germany
marcel.boehme@acm.org

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Intuitively we know, some software errors are more complex than others. If the error can be fixed by changing one faulty statement, it is a simple error. The more substantial the fix must be, the more complex we consider the error.

In this work, we formally define and quantify the complexity of an error w.r.t. the complexity of the error's least complex, correct fix. As a concrete measure of complexity for such fixes, we introduce Cyclomatic Change Complexity which is inspired by existing program complexity metrics.

Moreover, we introduce CoREBench, a collection of 70 regression errors systematically extracted from several open-source C-projects and compare their complexity with that of the *seeded* errors in the two most popular error benchmarks, SIR and the Siemens Suite. We find that seeded errors are significantly less complex, i.e., require significantly less substantial fixes, compared to actual regression errors. For example, among the seeded errors more than 42% are *simple* compared to 8% among the actual ones. This is a concern for the external validity of studies based on seeded errors and we propose CoREBench for the controlled study of regression testing, debugging, and repair techniques.

Categories and Subject Descriptors: D.2.5 [Software Engineering] Testing and Debugging

General Terms: Theory, Measurement, Experimentation

Keywords: Error Complexity, Coupling Effect, Regression

1. INTRODUCTION

Software errors can be arduous. Their fixes can account for half of the code changes even in well-tested software [4]. Before they are fixed, they can remain in the program for many years, causing problems for the software users. When they are fixed, these fixes can introduce even further errors.

Related processes can be automated based on our understanding of the inherent nature of software errors. Testing techniques seek to expose errors; debugging techniques seek to determine the faulty source code for an error; and repair techniques seek to fix the faulty source code.

[*]The first author conducted this work while at NUS.

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
ACM 978-1-4503-2645-2/14/07
http://dx.doi.org/10.1145/2610384.2628058

Two properties of errors are complexity and detectability. While the *complexity* of an error is determined by how substantial the error's fix is required to be, the *detectability* of an error is determined by the amount of input exposing the error. Intuitively, an error that is hard to detect may still require only a simple fix. Offutt [24] relates both properties and conjectures: the detectability of simple faults is similar to the detectability of complex faults – the coupling effect hypothesis. He defines simple faults as ones that *can* be fixed by changing one statement while complex faults cannot.

In this paper, we are the first to *quantify error complexity* and formally define the term and a metric. The complexity of an error is determined by the complexity of the correct, least complex *fix* of the error. The fix must be *correct* because no other errors should be introduced and *least complex* because even Offutt's simple faults can be fixed in multiple ways, including a complete revision of the program.

To measure the complexity of a fix, we formally define *software change complexity* and introduce a concrete change complexity metric – *Cyclomatic Change Complexity* (CyCC), which is inspired by McCabe's cyclomatic program complexity metric [20]. Program complexity is a measure of the interactions among the various elements of the software. Similarly, we define the change complexity as a measure of the interaction among the various *changed* elements in the *changed* software. We give an efficient algorithm to compute CyCC.

Equipped with our novel error complexity metric we set out to learn about the nature of complex regression errors. The two most popular benchmarks for experimentation with regression errors are the Siemens Suite [13] and SIR [9]. In both cases, most errors were introduced through a process called *fault seeding*. Developers were asked to change the given programs slightly such that they contain errors of varying detectability. However, we were not certain about a varying complexity of the *seeded* errors and constructed our own benchmark to compare to *actual* regression errors.

In this paper, we introduce CoREBench as a collection of 70 regression errors and compare the complexity of these to the complexity of the seeded ones in the Siemens Suite and SIR. We harvested the regression errors in CoREBench *systematically* from four widely deployed, well-tested open-source software projects. Indeed, we find that the seeded regression errors are significantly less complex, i.e., require significantly less substantial fixes, compared to the actual regression errors in CoREBench. For example, among the seeded errors more than 42% are simple compared to 8% among the actual ones. *This is a concern for the external validity of studies based on such seeded regression errors.*

We apply our error complexity metric to the regression errors in CoREBench in order to experimentally investigate the nature of complex regression errors. Three of our main findings are enumerated in the following:

• Between the complexity of the change introducing an error and of the change fixing it seems to be no correlation. That is, even simple changes can introduce complex errors. One could say that the *cause* of a regression error is already dormant in the code and the change merely *triggers* it. Or, the regression errors may be evolving when the program is and the *complexity of errors may change during evolution*.

• Between the complexity and life span[1] of an error seems to be no correlation. That is, even complex errors may be fixed on the same day when they are introduced or a few years later. This may be *indirect evidence that simple and complex errors are of similar detectability*, i.e., coupled [24].

• Change Interaction Errors (CIEs)[2] require consistently more substantial fixes than other types of regression errors (Non-CIEs). This suggests that *CIEs are not only of less detectability [4] but also of greater complexity* than Non-CIEs.

We define change complexity as a measure of interaction among the changed elements and introduce the CyCC as a concrete metric. Yet, there are other metrics, such as number of Changed Lines of Code (CLoC), paths, or hunks. We study CLoC versus CyCC and find: While both rarely agree on the *specific* value or rank of a change's complexity, they strongly correlate *in general*. Basically, both indicate high complexity for substantial change. We believe, CyCC is a precise and practical measure of change complexity.

In summary, this paper makes the following *contributions*.

1. **Error Complexity Metric**. We formally define and quantify the complexity of an error w.r.t. the complexity of the error's least complex and correct fix. Investigations into error complexity are relevant for software testing, debugging, and repair: What is the root cause of an error that requires a substantial fix? Is a test suite adequate to expose complex errors? How do we correctly and efficiently repair complex errors?
2. **Change Complexity Metric**. We formally define software change complexity, introduce CyCC as a concrete complexity metric, discuss an algorithm to compute the CyCC efficiently based on a graph containing the control-flow among the changed statements, and make available a tool that computes the CyCC of any C source code commit in under one second on average.
3. **Regression Error Benchmark**. We make available CoREBench, a collection of 70 realistically complex regression errors. For each error, we provide the bug report, the error-introducing source code commit, the error-fixing source code commit, and a validating test case that fails for all versions between these commits, but passes before and after.
4. **Empirical Study**. We study the complexity of actual regression errors and establish that seeded errors in existing benchmarks are significantly less complex.

CoREBench and the implementation of CyCC are available at `http://www.comp.nus.edu.sg/~release/corebench`.

[1]The life span of an error is the time an error is observable from when it is introduced to when it is fixed.

[2]A regression error is a CIE [4] if a *sequence* of changed statements must be executed in order to expose the error while "skipping" one of them does not expose the error.

2. AN ERROR COMPLEXITY METRIC

We define the complexity of an error w.r.t. the complexity of the correct, least complex fix of the error. To measure the complexity of a fix, we formally define software *change* complexity as a measure of the interaction among the *changed* elements in a *changed* program and propose a concrete change complexity metric. Cyclomatic Change Complexity (CyCC) directly measures the number of linearly independent[3] *change sequences* in a changed program and is thus inspired by McCabe's cyclomatic program complexity. Intuitively, CyCC quantifies the amount of *changed* decision logic in the program.

```
351  : intmax_t value = 0;
352  : int sign = (*valuestring == ’-’ ?  -1 :  1);
353  : if (sign < 0)
354  :   valuestring++;
355  : do {
356  :   if (ISDIGIT(*valuestring))
357  :     value = 10*value + sign * (*valuestring-’0’);
358  : } while (*++valuestring)
359--: return value * sign;
359++: return value;
```

Figure 1: Fix of simple error `core.6fc0ccf7`

Figure 1 shows an example of a simple error in `coreutils`. The simplified code fragment parses a `valuestring` into an integer `value`. However, every string containing a negative number is parsed as a positive number. This error is simple because only one statement (in line 359) needs to be changed in order to repair the error.

```
447++: else if (ent->fts_info == FTS_NS) {
448++:   if (ent->fts_level == 0){
449++:     reportSymlinkLoop();
450++:   } else {
451++:     if (symlink_loop(ent->fts_accpath)){
452++:       reportSymlinkLoop();
453++:     }
454++:   }
456++: }
```

Figure 2: Fix of complex error `find.24bf33c0`

Figure 2 shows an example of a complex error in `findutils`. The bug report states that "`find` does not report symlink loop when trying to follow symlinks". Hence, the developer adds the presented code fragment to describe conditions under which symlink loops need to be reported. The error is complex because it requires three additional conditional statements and several statements to fix it correctly.

2.1 Measuring Change Complexity

Traditional program complexity measures the interaction among the elements in a software system. So, we can define:

Definition 1 (*Change Complexity*)

Change complexity is a measure of the interaction among the changed elements in a changed program.

Note that deleted statements are changed elements nevertheless and can be represented by dummy statements in the changed program (see e.g., [25]).

[3]A *linearly independent path* is a complete path through the program that introduces at least one new edge that is not included in any other linearly independent paths.

As a concrete measure of change complexity, we introduce CyCC which is computed based on a graph containing the control-flow among the changed basic blocks – the CSG.

Definition 2 (*Change Sequence Graph (CSG) [4]*)

The change sequence graph of a changed program P' is a directed graph containing as vertices the program entry as source, the program exit as sink, and the changed basic blocks in P', with an edge between any two vertices if control may pass from the first to the second without passing through a third.

The source vertex is connected through an edge to every changed basic block that may be *executed first*, that is, before some other changed basic block is executed. To the sink vertex is connected every changed basic block that may be *executed last*, that is, after any other changed basic block is executed. This simplified definition of CSG accounts for all sequences of changed statements that can be exercised but not for potential interaction locations (see [4]) and can be computed from the changed program's Control Flow Graph.

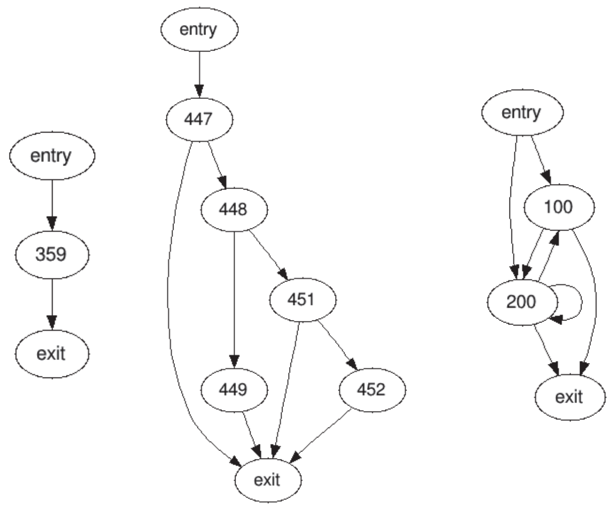

Figure 3: Change sequence graphs with linear independent paths `(359)` **(left);** `(447)`, `(447-448-449)`, `(447-448-451)`, `(447-448-451-452)` **(middle); and** `(100)`, `(200)`, `(100-200)`, `(200-100)`, `(200-200)` **(right).**

For example, Figure 3 depicts three different CSGs. The paths through a CSG from source to sink represent different sequences of changed statements that may be executed. The CSGs on the left and in the middle are computed for the changed code fragments in Figures 1 and 2. It is interesting to note that the size of the CSG depends only on the size of the changed code and not on the size of the complete program.

Definition 3 (*Cyclomatic Change Complexity*)

The complexity of a set of program changes C is defined with reference to the Change Sequence Graph constructed for C as $CyCC = E - N + 2P$, where
E is the number of edges of the CSG,
N is the number of nodes of the CSG, and
P is the number of connected components in the CSG.

Cyclomatic Change Complexity (CyCC) measures the number of linearly independent *sequences of changed statements* from entry to exit in a changed program. We argue that the changed statements in each sequence may "interact" differently. In fact, some sequences are critical in exposing so called Change Interaction Errors [4] while others are not. In Figure 3, based on the number of linearly independent paths in the CSG, we compute a CyCC=1 (left), CyCC=4 (middle), and CyCC=5 (right), respectively.

2.2 Measuring Error Complexity

Before we define and measure the complexity of an error, we quote the IEEE glossary to define what we mean by error.

Definition 4 (*Software Error [15]*)

A software error is the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

An error's *detectability* is determined by the proportion of program inputs that expose the error. Such input is said to fail w.r.t. the error. For example, the code fragment in Figure 1 parses negative numbers incorrectly. E.g, input setting `valuestring` to "`-2`" fails w.r.t. the error as it produces the output `value` of `2` instead of `-2`. If `valuestring` is directly a program input, then the error has a high detectability.

Definition 5 (*Error Complexity*)

The complexity of an error E is the complexity of the least complex change required to pass all input that fails w.r.t. E while the output for all other input remains unchanged.

Intuitively, we define the complexity of an error based on how substantial its fix must be – without introducing new errors. For example, the error in Figure 1 can be correctly repaired with the change of only one statement (line 359). With $CyCC = 1$, it is a *simple error*. The error in Figure 2 can be repaired with a change involving three additional conditional statements. Assuming short-circuit evaluation for these conditions (see [12]), the CSG in Fig. 3 (middle) might be the least complex. The error is of complexity four.

However, we note that other measures, such as number of changed LoC, paths, or hunks, may assign different *specific* values to an error's complexity. While different measures may disagree on its specific value or rank, they should correlate in general (see RQ1 in Sec. 5). For instance, if an error requires a substantial fix involving a *high* number of changed LoC distributed over the code, the values for other measures of complexity should be *high* as well. We believe that CyCC is a precise and practical measure of change complexity as given in Def. 1 and thus of error complexity as in Def. 5.

3. COMPUTING INTER-PROCEDURAL CHANGE SEQUENCE GRAPHS

We present an algorithm to synthesize the *inter-procedural Change Sequence Graph* (CSG) efficiently from the intra-procedural control-flow graphs of the changed methods and the call graph of the changed program. The *intra-procedural Control-Flow Graphs* (CFGs) of the changed methods are traversed to establish the control-flow among the changed basic blocks in the CSG. The *Call Graph* (CG) of the changed program is traversed to establish whether a basic block transitively calls a changed method.

The inter-procedural CSG is computed *more efficiently* than previously in Reference [4] because it does not require the entire *inter*-procedural CFG for the complete program.

Algorithm 1 depicts the CSG construction process. Given two versions of a program, P and P', the algorithm computes the inter-procedural CSG. After determining which methods and basic blocks have changed, the algorithm follows along the control-flow and method calls from every changed basic block onwards. If another changed basic block is found, an edge is added to the CSG between the original changed basic block and the found one. Then, we establish whether the original changed basic block can be executed as first or last changed basic block and a corresponding edge is added to program entry and exit, respectively.

Algorithm 1 Inter-procedural Change Sequence Graph

```
Input: Programs $P$ and $P'$
 1: determine changed methods and basic blocks using diff
 2: let $CG \leftarrow constructCallGraph(P')$
 3: let $CSG \leftarrow \{entry, exit\}$
 4: for each changed method $m \in CG$ do
 5:     let $CFG \leftarrow constructCFG(m)$
 6:     add all changed basic blocks from $CFG$ to $CSG$
 7:     for each changed basic block $c \in CFG$ do
 8:         TRAVERSECHANGE($c$, $CFG$, $c$)
 9:     end for
10: end for
11: CONNECTENTRYEXIT()
12:
13: function TRAVERSECHANGE($curr$, $CFG$, $c$)
14:     if $curr$ marked as traversed then return
15:     else mark $curr$ as traversed
16:     for each $bb$ that directly follows $curr$ in $CFG$ do
17:         if $bb$ is a changed basic block then
18:             add an edge from $c$ to $bb$
19:         else
20:             TRAVERSECHANGE($bb$, $CFG$, $c$)
21:         end if
22:     end for
23:     for each changed $m'$ that $curr$ may call in $CG$ do
24:         let $CFG' \leftarrow constructCFG(m')$
25:         TRAVERSECHANGE($CFG'.first$, $CFG'$, $c$)
26:     end for
27: end function
Output: Inter-procedural $CSG$
```

In more detail, Algorithm 1 works as follows. First, a syntactic differencing-tool, such as the Unix `diff`-tool, determines the syntactic differences between both program versions (line 1). These differences are used subsequently to determine in the changed version those basic blocks and methods that have changed. Then, the call graph is constructed for the changed program and the CSG initialized with *entry* and *exit* vertices (lines 2-3). After this initialization, the algorithm computes the intra-procedural CFG for each changed method m, adds the changed basic blocks from the CFG into the CSG, and starts traversing the control-flow recursively from each changed basic block c onwards (lines 4-10). Since the method TRAVERSECHANGE is a recursive traversal algorithm, we mark the *visited* vertices as such (lines 14-15). If any basic block bb transitively following c is changed, then add an edge from c to bb (lines 16-21). If c or any transitively following basic block, transitively calls a changed method m', continue traversal from the $first$ basic block in the CFG' of m' (lines 23-26). Finally, the method CONNECTENTRYEXIT computes the edges from the *entry*-vertex to any changed basic block that can be executed first, that is before some other basic block is executed, and the edges to the *exit*-vertex from any changed basic block that can be executed last, that is after any other changed basic block is executed (cf. Def. 2). A complete implementation is discussed in Sec. 4.3.2.

4. EMPIRICAL STUDY

4.1 Objects of Empirical Analysis

4.1.1 CoREBench: Complex Regression Errors

CoREBench is a collection of 70 regression errors that we systematically extracted from the code repositories and bug reports of four open-source software projects: `Make`, `Grep`, `Findutils`, and `Coreutils` (see Fig. 4).

We chose these projects because they are well-specified, well-tested, well-maintained, and widely-used open source programs with standardized program interfaces. The version history and all bug reports can be publicly accessed on the GNU homepage.[4] The program interfaces and parameters were specified in POSIX as IEEE standard in 1988 [14].

We built the corpus by (1) identifying a regression-fixing commit in the 1,000 most recent revisions and a test that passes after but fails before the fix, and (2) the regression-introducing commit, such that the same test passes before and fails after the commit. Regression errors which could not be reproduced using a test case are not reported. This was the case for some system- or concurrency-related bugs.

To identify a *regression-fixing commit* ($\overset{Fix}{\dashrightarrow}$), we parsed the commit messages of the 1,000 most recent commits and a file which highlights recent new features and fixes for keywords, such as "regression", "introduced", and "broken". Except for `Make`, the file and commit messages are sufficiently detailed and may even reference the error-introducing commit. For `Make`, we parsed the bug report referenced in the commit messages. Also for `Make`, we removed seven commits in which the regression fix was tangled[5] with other fixes. Computing the error complexity based on tangled fixes will give wrong results. For all regression errors we ensure that the commit is solely devoted to fixing exactly one error. The error-witnessing test case was always provided with the bug-fixing commit or the bug report.

To identify the *error-introducing commit* ($\overset{Reg}{\dashrightarrow}$), we used the error-witnessing test case and a binary search on the complete version history of the subject. The binary search is automated using `git bisect`, which conceptually searches all revisions before the error-fixing commit to determine the exact (error-introducing) commit before which the test case passes ($P_{\checkmark}$) and after which the test case fails ($P_{\text{✗}}$). For `Coreutils`, we add five regression errors that we already identified in Reference [4]. Finally, we determined two commits describing the lifetime and a test case exposing the effects of each regression error:

$$\ldots \dashrightarrow P_{\checkmark} \overset{Reg}{\dashrightarrow} P_{\text{✗}} \dashrightarrow \ldots \dashrightarrow P_{\text{✗}} \overset{Fix}{\dashrightarrow} P_{\checkmark} \dashrightarrow \ldots$$

[4]`http://savannah.gnu.org` and `http://debbugs.gnu.org`
[5]See Reference [11].

Subject	Size	Maturity	#Commits	#Tests	#Bug Reports	Extract. Period	#RErrors
	in kLoC	1st commit	total (last year)		marked fixed	recent 1k commits	extracted
`Coreutils`	83.1	Oct. 1992	27,807 (290)	4772	832	08.05.11 – 06.10.13	**22**
`Findutils`	18.0	Feb. 1996	2,031 (43)	1054	312	01.08.05 – 26.10.13	**15**
`Grep`	9.4	Nov. 1989	1,307 (31)	1582	66	25.09.01 – 26.10.13	**15**
`Make`	35.3	Apr. 1988	2,288 (134)	528	305	01.03.96 – 24.11.13	**18**

Figure 4: Subjects of CoREBench

Using this approach, we have identified and validated 70 regression errors (incl. six segmentation faults) that were introduced by 57 different commits. From the time an error was introduced to the time the error was fixed, it took on average 1.7 years. Eleven errors were fixed incorrectly. In these cases the error was indeed removed in the fixed version. Yet, up to three new errors were introduced that required further fixes. About one third of the errors were introduced by changes not to the program's behavior but to non-functional properties such as performance, memory consumption, or APIs. In some cases one error would *supercede* another error such that the superceded was not observable for the duration that the superceding remained unfixed.[6]

4.1.2 Base Line: SIR and Siemens Suite

The *Subject Infrastructure Repository* (SIR) [9] and the *Siemens Suite* [13] are arguably the most popular error benchmarks. For every correct program version $P_{\checkmark}$, there are several faulty versions $P_{\text{✗}}$. One may evaluate regression testing and debugging techniques by considering:

$$P_{\checkmark} \overset{Reg}{\dashrightarrow} P_{\text{✗}} \overset{Fix}{\dashrightarrow} P_{\checkmark}.$$

The popularity may be due to the provision of test oracles, standardized program interfaces, a large number of test cases, and a uniform format for the materials provided. Program input and output are clearly defined. Each subject consists of a "golden version" as test oracle and several erroneous versions with one fault each. Measuring popularity by the number of citations: In the five years preceding this paper, the publications associated with the SIR [9] and Siemens Suite [13] have been cited almost six hundred times.

Figure 5 shows the characteristics of the subjects in both benchmarks. The number of tests was derived from the file `universe` while the number of regression errors was derived from `Fault_Seeds.h` that accompanies each subject.

	Subject	Size in kLoC	#Tests	#Regression Errors
Siemens Suite	tcas	0.2	1,608	41
	totinfo	0.6	1,052	23
	printtokens	0.7	4,130	7
	printtokens2	0.6	4,115	10
	replace	0.6	5,542	32
	schedule	0.4	2,650	9
	schedule2	0.4	2,710	10
SIR (C Subjects)	space	6.2	13,585	38
	bash	59.8	1,200	32
	flex	10.5	628	81
	grep	10.1	625	57
	gzip	5.7	214	59
	make	35.5	795	35
	sed	14.4	370	32
	vim	122.2	974	22

Figure 5: Subjects of Siemens Suite and SIR

[6]For instance, `find.66c536bb` supercedes `find.dbcb10e9`.

Unfortunately, in both benchmarks almost all errors were created by manual fault seeding[7]. We claim that fault seeding introduces a *bias towards less complex errors*. Our novel measure of error complexity, for the first time, allows us to assess the substance and extent of this bias.

4.2 Variables and Measures

Our experiment manipulated two independent variables (IV):

- **IV1 Genuineness**: There are two categorical factors of genuineness. *Seeded regression errors* result from faults that were manually seeded. *Actual regression errors* appear in typical evolving software projects.
- **IV2 Regression Cause**: We consider two categorical factors of regression cause. *Change Interaction Errors* (CIEs) can be observed only if a certain sequence of changes is exercised (cf. [4]). *All other errors* (Non-CIEs) are regression errors that are not CIEs.

In our experiment, we measured 3 dependent variables (DV):

- **DV1 Error Complexity**: We consider two measures of error complexity which is defined w.r.t. the error-fixing commit. The *Cyclomatic Change Complexity* (CyCC) is described in Section 2.2. The *Changed Lines of Code* (CLoC) corresponds to the number of executable source code lines that were changed. Both are measured for the version just before the error is fixed.
- **DV2 Error Life Span**: We measure the error life span as the number of days between the commit introducing and the commit fixing the error.
- **DV3 Error-Introducing-Commit Complexity**: We measure the error-introducing-commit complexity as CyCC of the commit introducing the error.

4.3 Experimental Design

4.3.1 Measuring Error Complexity for CoREBench

To investigate the complexity of actual regression errors, we analyse their actual fixes. But why should the actual fix be that "least complex, correct" fix describing the error complexity (see Def. 5)? In fact, for each error there can be innumerable fixes and not every fix is *correct* such that not only the observed error is fixed but also no new errors are introduced and *least complex* such that no other correct fix is of less complexity.

In practice, we neither have all possible fixes nor do we have all possible test cases that observe that the error (and only the error) is really fixed. Instead, for the analysis of CoREBench we put forward the following hypothesis:

Competent Repair Hypothesis.

Software developers write fixes with a complexity as low as possible and that are close to being correct.

[7]Except for `space`, all errors are manually generated.

First, the Competent Repair Hypothesis (CRH) states that developers write fixes that are as *simple as possible*. For several errors in CoREBench we found two fixes – the second fixed the error "more efficiently" or repaired "the root cause" of the error even though the first fix was already a *correct* one.[8] Complex fixes are often accompanied by very elaborate explanations why such complex changes were necessary to fix the error.

Then, the CRH states that developers write fixes that are *close to being correct*. Indeed, the fixes for eleven of seventy errors in CoREBench were incorrect such that the repair of one introduced a new error. However, in general we believe that the programmer is likely to fix the error correctly. If this was not the case, we would register an exponential increase of bug reports. This hypothesis is an instance of the Competent Programmer Hypothesis [8] which states that developers "create *programs* that are close to being correct".

4.3.2 Infrastructure and Implementation

We implemented Algorithm 1 based on the C Intermediate Language (`CIL`) program analysis framework [22] and the Unix `diff` tool to compute the Cyclomatic Change Complexity (CyCC) and the executable Changed Lines of Code (CLoC) of a code commit as the two measures of DV1. Both, tool implementation and CoREBench can be downloaded at `http://www.comp.nus.edu.sg/~release/corebench`.

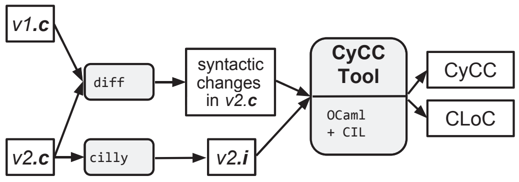

Figure 6: CyCC Tool Implementation

As depicted in Figure 6, the implementation works as follows. First, the changed version ($v2.c$) is compiled into an intermediate file ($v2.i$) using `cilly`. Then, our script uses the `diff` tool to determine the lines of code that have syntactically changed in $v2.c$. Note that *CLoC* is the number of *executable* changed lines of code while the syntactic changes can also comprise comments. If the program version history is maintained remotely and the changed version is available on the local machine, our script uses the previous verison ($v1.c$) from the repository. Otherwise, its location must be provided to compute the difference.

Next, `CIL` can compute the call graph and intra-procedural control-flow graphs (CFG) for the changed program. Using the output of the `diff`-tool, we find the changed methods in the call graph and the changed basic blocks in the CFGs of the changed methods. Note that `diff` detects any line of a multi-line statement that is changed while `CIL` only maintains the first of the potentially multiple lines of a statement. We address this issue for the most common multi-line statement (`if`-conditions) but not for others. Furthermore, top-level variable and method declarations (e.g., `int x;`) are not available in the `CIL` CFGs and macros are readily expanded. Thus, modifications of these program elements, as well as deleted basic blocks, are not reflected in the `CIL`-CFGs and the inter-procedural CSG, respectively.

[8]See commit message of `find.b445af98`

Once the change sequence graph is synthesized for a source code commit, our implementation computes the CLoC and CyCC according to Definition 3. Note that during our experiments, we ignore errors and code commits that yield "empty" CSGs. For CoREBench, we report the results for all 70 regression errors. However, for SIR and the Siemens Suite, several changes were *only* to variable or method declarations (e.g., change of type) or C macros. While these were ignored, we report the results for the remaining 259 regression errors in SIR and 108 regression errors in Siemens.

The experiments were run on a Linux machine with Intel Core2 Quad CPU at 2.83GHz and 4GB of main memory. On average, it took *less than 1 second* to compute the complexity of an error.

4.4 Threats to Validity

Construct validity refers to the degree to which a test measures what it claims, or purports, to be measuring. Three threats to construct validity are the empirical reliability of the competent repair hypothesis, the reliability of CyCC as good measure of error complexity, and the correctness of the implementation of the measure into the `CyCC` tool. **(i)** The Competent Repair Hypothesis (CRH) links that theoretical least complex, correct fix specified in Definition 5 to the actual fix of the errors in CoREBench (see Sec. 4.3.1). Assuming the CRH, we measure the complexity of actual regression errors based on the actual fixes of these errors. If the CRH does generally not hold, the actual error complexity may be different from the measured error complexity. **(ii)** The CyCC metric may not be a good measure of the complexity of a fix and thus of error complexity. However, we note that Definition 3 of CyCC is inspired by an existing measure of *software* complexity [20] which itself inspired Definition 1 of change complexity. We study the relationship to another measure of change complexity (see Sec. 5). **(iii)** The `CyCC` tool may be incorrectly implemented. For instance, some changed elements, like deleted basic blocks, are not represented in the computed CSG from which the CyCC is computed. However, all results are computed using the same tool, subjecting each (compared) measurement to the same potential bias. Furthermore, we make available the source code of the implementation for inspection.

External validity refers to the extent to which the results of a study can be generalized to other objects which are not included in the study. One threat to the external validity is the representativeness of the the chosen objects of empirical analysis. Indeed, our objects are well-maintained, open-source C software projects containing regression errors typical for such projects. However, for instance regression errors in projects written in other languages, like Java, or in commercially developed software may be of different kind and complexity. Hence, the results and conclusion are to be interpreted in this context.

Internal validity refers to the degree to which the independent variable causes the changes seen in the dependent variable being examined within the study. While it is clear that (IV1) the actual regression errors are not seeded and vice versa, it may be that (IV2) regression errors classified as change interaction errors are not actually change interaction errors. However, for each regression error, we attempted to determine the specific sequence of changed statements that need to be exercised to expose the error. In the results we note which errors could thus not be classified.

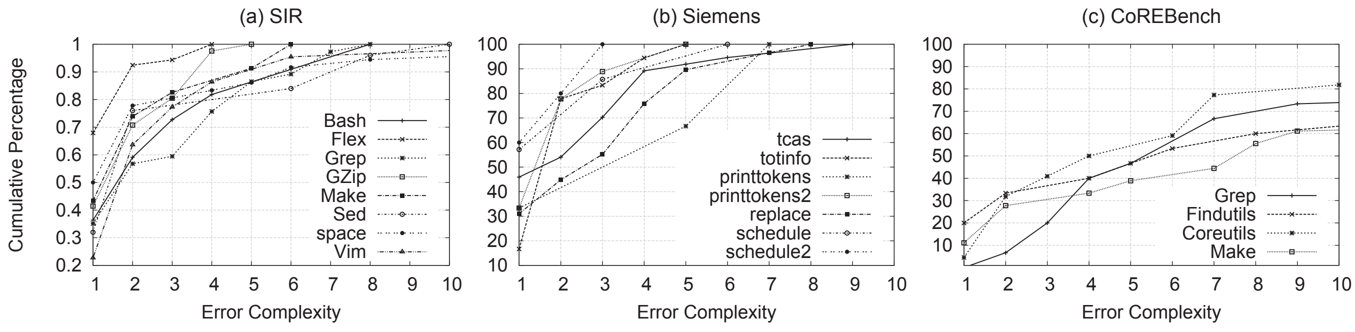

Figure 7: Cumulative distribution of error complexity for all subjects in each benchmark

5. DATA AND ANALYSIS

We investigate the nature of complex regression errors. In our main research hypothesis, we claim that the process of creating errors using manual fault seeding introduces a *bias towards less complex errors*. Formally, we submit a null hypothesis which needs to be rejected in order to empirically prove this claim. We also find out whether actual, more complex regression errors have a longer life span and whether complex errors are introduced by complex commits.

Furthermore, we investigate another measure of change complexity – the number of Changed Lines of Code (CLoC). While we cannot directly compare both measures, we find out whether our Cyclomatic Change Complexity (CyCC) and CLoC agree on the *ranking* of two-hundred commits in terms of their complexity. If so, CLoC and CyCC may be used interchangeably to assess the complexity of a commit.

Lastly, we use the actual regression errors to study the prevalence, complexity, and life span of an interesting class of regression errors – Change Interaction Errors (CIEs; cf.[4]). We classify a regression error E as CIE if a sequence of at least two changed statements must be executed in order to expose E while "skipping" one of the changed statements does not expose E. Conservatively, we also require that each change in the sequence *can* potentially be skipped.

5.1 Research Questions and Null Hypothesis

In statistical inference, the *null hypothesis*, H_0, states there is no relationship between two measured phenomena. The null hypothesis can be *rejected* based on observed data of a scientific experiment with the conclusion that there is very likely a relationship. The null hypothesis can never be accepted as more data may still reveal a relationship.

To test H_0, we measure either a *difference* or the *strength of the relationship*. In the first case, we subtract the mean of one from the mean of the other dataset. In the latter case, we measure Spearman's rank correlation coefficient [28] which is more robust for non-normal distributions than the common Pearson's product moment correlation. If we fail to reject H_0 with a very low correlation coefficient, we can still conclude that if a relationship exists, it is very weak.

- H_0^a : There is no difference between the complexity of *seeded* and *actual regression errors*.
- H_0^b : There is no relationship between the *complexity* and *life span* of a regression error.
- H_0^c : There is no relationship between the complexity of the *error* and the *commit introducing the error*.

Furthermore, we want to answer these research questions:

- **RQ1** Can the *number of Changed Lines of Code* (CLoC) and the *Cyclomatic Change Complexity* (CyCC) be used interchangeably?
- **RQ2** What is the complexity, prevalence, and life span of *Change Interaction Errors*?

H_0^a : Seeded vs. Actual Errors (IV1, DV1)

We compare the error complexity (as CyCC) of the seeded regression errors in the Siemens Suite and SIR with that of the actual regression errors in CoREBench to study the effects of IV1 on DV1 and test H_0^a. For SIR and the Siemens Suite, we measure the complexity of the errors by considering the non-faulty versions as the fix for the error in the faulty versions. For CoREBench, we measure the complexity of the errors by analyzing the complexity of the regression-fixing commits and assume the Competent Repair Hypothesis. Also, for CoREBench we choose the regression errors such that every regression-fixing commit is designated to fixing exactly one error only.

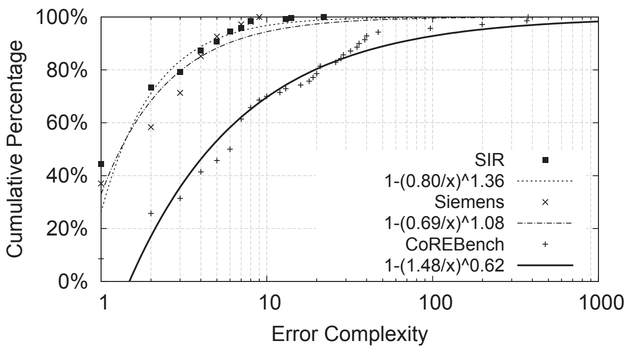

Figure 8: Cumulative distribution of error complexity for seeded errors (SIR and Siemens) vs. actual errors (CoREBench)

We reject H_0^a *and conclude: seeded regression errors are significantly less complex than actual regression errors.* The mean error complexity differs by 21.9 for SIR and 21.7 for the Siemens Suite. Fitting the data to a power-law distribution, we compute the cumulative distribution functions shown in Figure 8. The complexity distributions for each subject and benchmark are shown in Figure 7.

Among the seeded errors, *simple errors* (complexity one) occur five times more often than among the actual errors. Specifically, 42% of the seeded errors are simple while only 8% of the actual errors are. Simple errors are characterized by a localized fault and can often be fixed by changing just one statement. In contrast to actual errors, the complexity of the seeded errors barely exceeds 10. Less than 1% of the seeded errors have a complexity of more than 10 compared to 30% of the actual errors. This means, that actual errors are generally more complex than the errors created through manual fault injection. The most complex error in CoREBench is twenty times more complex than the most complex error in the SIR and the Siemens Suite.

H_0^b : Life Span vs. Complexity (DV1, DV2)

We compare the life span and complexity of actual regression errors to study the correlation between DV1 and DV2 and test H_0^b. Every commit has a timestamp, so we can compute the life span of an error by subtracting the timestamp of the error-introducing from that of the corresponding error-fixing commit. We measure the complexity using CyCC and depict the results in Figure 9.

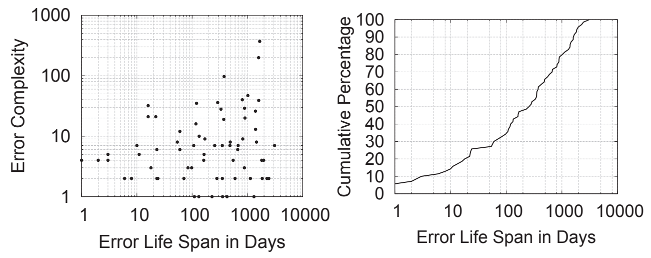

Figure 9: Correlation of error life span vs. complexity (left), cumulative distribution of life span (right)

We cannot reject H_0^b *and conclude that if a relationship between the life span and complexity of an error exists, then it is very weak.* We compute a Spearman's rank correlation coefficient of $\rho = 0.0675$ with a two-sided p-value=0.5790. In other words, even simple errors that are *"easy" to fix* can take a very *long time to fix.* Vice versa, even complex errors that are difficult to fix can be fixed on the same day as the error is introduced.

Independent of error complexity, error life span follows a power-law distribution. Once introduced, 12% of the regression errors are fixed within a week while half of them stay undetected and uncorrected for more than 9 months up to 8.5 years. While there is a large number of errors with a small life span, there is a small number of errors with very large life span.

H_0^c : Introducing vs. Fixing Errors (DV1, DV3)

For each actual regression error, we compare the CyCC of the commit introducing and the commit fixing the error to study the correlation between DV1 and DV3 and test H_0^c. The results are presented in Figure 10. On the left, we show for each regression error the complexity of the commit introducing the error versus the complexity of the commit fixing the error. On the right, we show the cumulative distribution of error-introducing and error-fixing commits independently.

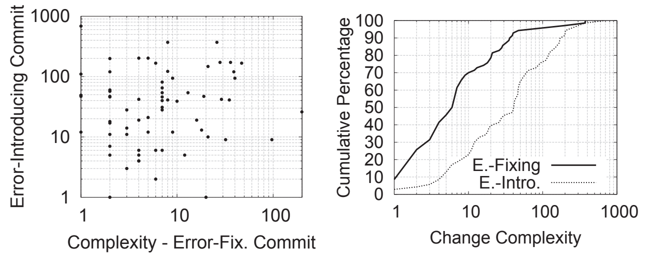

Figure 10: Correlation (left) and cumulative distribution (right) of the complexity of the two commits introducing and fixing an error.

We cannot reject H_0^c *and conclude that if there exists a relationship between the complexity of an error and the complexity of the commit which introduces the error, then it is very weak.* We compute a Spearman's rank correlation coefficient of $\rho = 0.1656$ with a two-sided p-value=0.1705. In other words, even complex errors can be introduced by simple changes and vice versa. One interpretation is that sometimes the root cause of some complex regression errors is already dormant in the program and only "unmasked" in the changed code. Then, we should consider these changes as *the trigger instead of the root cause* of an observed error. Another interpretation is that the error itself evolves during its life span due to many other changes to the program. Then, *the complexity of errors may change during evolution.*

On average, error-introducing commits are more complex when compared to error-fixing commits (see Fig. 10 – right).

RQ.1 Changed Lines of Code as Proxy Measure

For 200 random code commits[9], we measure the CyCC and Changed Lines of Code (CLoC), to study the concordance and correlation of two measures of DV1 (Error Complexity). *Concordance* describes the degree to which both measures agree on the complexity of a set of changes and is measured using Cohen's kappa [6]. Full agreement ($\kappa = 1$) means that CyCC rates a set of changes C_1 more complex than another set of changes C_2 if and only if CLoC rates C_1 more complex than C_2. In contrast, *correlation* describes the strength of the relationship and is measured using Spearman's ρ. Strong correlation ($\rho = 1$) means that if CyCC is large than CLoC is also likely to be large and vice versa.

The results are presented in Figure 11. The Bland-Altman plot [3] on the left allows us to compare the *differences* between the measurements with both measures of complexity for each commit. The mean ($\bar{x}$) of these differences is called *bias* and the reference interval ($\bar{x} \pm 1.96\times$standard deviation) is called *limits of agreement.* If the measures tend to agree, the differences will be plotted near zero. As CLoC and CyCC are not directly comparable and the power-law distribution generates strong outliers, we compare the *ranks* instead of the measurement values. The rank of measurement lies between one and the number of measurements and is greater than the rank of another measurement if and only if the measurement value is greater than that of the other measurement. The plot on the right depicts the (value) correlation of both measures on a logarithmic scale.

[9]We chose the 50 most recent code commits in each of the projects `Coreutils`, `Findutils`, `Grep`, and `Make`.

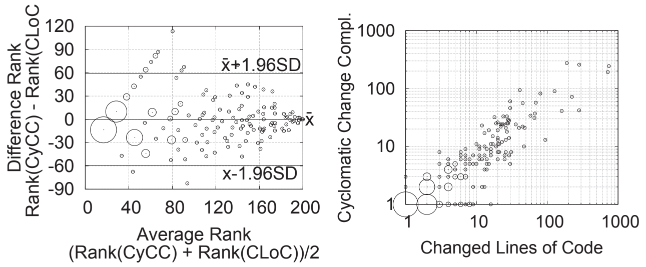

Figure 11: Bland-Altman plot of measurement ranks (left) and correlation (right) of CLoC vs. CyCC.

Moderate Agreement. *The Changed Lines of Code and Cyclomatic Change Complexity cannot be used interchangeably to assess the complexity of a set of changes.* The limits of agreement, shown in the Bland-Altman plot, are far apart (± 59.4 out of 200 ranks). We also compute a Cohen's kappa of $\kappa = 0.014$ for the measurement ranks ($\kappa = 0.151$ for the values) which indicates only moderate agreement between both measures on the complexity of a code commit.

Two measures that are designed to measure the same property (here, change complexity) may not agree but should have a good correlation. Indeed, we compute Spearman's correlation $\rho = 0.86$ with a two-sided p-value < 0.0001. So, as the CLoC increases, the CyCC increases and vice versa.

RQ.2 Complexity, Life Span, and Prevalence of Change Interaction Errors (IV2, DV1, DV2)

We compare the error complexity (as CyCC) and life span of Change Interaction Errors (CIEs) with the error complexity and life span of actual regression errors that are not Change Interaction errors (Non-CIE) to study the effects of IV2 on DV1 and DV2. We also measure the prevalence of CIEs among actual regression errors.

The results are presented in Figure 12. In the table, we show the classification of actual regression errors into CIE, Non-CIE, and Unclassified. For the latter, the regression cause could not be identified. On the left, we show the cumulative distribution of the complexity of CIEs versus Non-CIEs cropped at an error complexity of 50. On the right side, we show the cumulative distribution of the life span of CIE versus Non-CIEs on a logarithmic scale.

	CIE	Non-CIE	Unclassified
`Coreutils`	7	13	2
`Findutils`	5	7	3
`Grep`	5	7	3
`Make`	5	10	3
Total	22	37	11

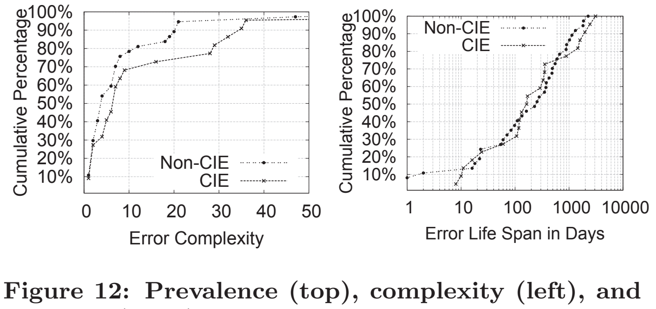

Figure 12: Prevalence (top), complexity (left), and life span (right) of Change Interaction Errors

Error Complexity. *CIEs are consistently more complex than Non-CIEs.* The mean complexity of CIEs (20.1) differs from that of Non-CIEs (9.9) by 10.2. On average 10% more CIEs exceed any given complexity than Non-CIEs. For example, while about 32% of the CIEs exceed a complexity of 10, only 22% of the Non-CIEs exceed the same complexity. This means CIEs are *"more difficult to fix"* than other types of regression errors.

Error Life Span. *CIEs and Non-CIEs have a similar life span.* Indeed, the mean life span of CIEs (623 days) differs from that of Non-CIEs (463 days) by 160 days. However, on average only 1% more CIEs exceed any given life span than Non-CIEs. From the chart (Fig. 12–right) it seems evident that there is no significant difference between the life span of CIEs and that of Non-CIEs. This means CIEs are manually *"as difficult to find"* as other types of regression errors.

Prevalence. *Change interaction errors are prevalent.* In fact, 22 of 59 classified actual regression errors can be classified as CIEs. This means that the existence of change interaction errors as a particular type of regression errors must be considered during the testing and debugging of evolving open source C programs. The prevalence and peculiarity of change interaction errors suggests that *CIEs should not be disregarded* during the empirical evaluation of techniques and methodologies in the scientific research of regression testing, debugging, and program repair.

In summary, compared to any other type of regression errors, CIEs are more difficult to expose automatically [4] while it takes the same time to encounter them manually (cf. error life span). Once discovered, CIEs are "more difficult to fix" (cf. error complexity). Since CIEs are prevalent in open-source C programs, they form an important class of regression errors that can be studied in CoREBench.

6. RELATED WORK

We first discuss investigations into the relationship of error complexity and detectability, continue with work related to quantifying error complexity, and conclude with an overview of related work on the construction and public provisioning of a benchmark suite with actual regression errors.

Offutt [24] asserts a relationship between the detectability and complexity of software errors. He defines a *simple fault* as one "that can be fixed by making a single change to a source statement" while a *complex fault* is one that can thus not be fixed. In his *coupling effect* hypothesis he conjectures that a "test dataset that detects all simple faults in a program will detect a high percentage of the complex faults" which holds if and only if the detectability[10] of simple errors is somewhat similar to the detectability of complex errors. In the present work, we have extended Offutt's definition of error complexity to be ordinal rather than nominal.

Andrews et al. [1, 2] confirm that the detectability of simple errors resulting from auto-generated faults (i.e., mutants) is similar to the detectability of actual (complex) errors and conclude that the mutation-adequacy of a test suite is a good indicator of its fault-detection capability. Namin et al. [21] caution that this insight is highly sensitive to external threats mentioning several influential factors that must be accounted for. In the present paper we have investigated not the detectability but the *complexity* of regression errors and

[10]The detectability of an error is determined by the proportion of input exposing the error (see Sec. 2.2).

found that the complexity of regression errors resulting from seeded faults is different from that of actual regression errors. This raises concerns for the validity of studies based on SIR and Siemens specifically and on seeded errors in general.

While it is intuitively clear that some errors are simple and others certainly more complex, we are not aware of any previous attempt to *quantify* error complexity. However, there has been a great effort to understand how to quantify software complexity [29]. Some established measures of software complexity are McCabe's cyclomatic complexity [20], Henry and Kafura's information flow complexity [10], and Chidamber and Kemerer's object-oriented complexity [5]. To quantify error complexity, we introduce and compare two measures – the cyclomatic change complexity (CyCC) and the number of changed lines of code (CLoC).

A popular technique to extract *actual regression errors* from software repositories is the SZZ-algorithm [26, 16]. First, SZZ identifies the error-fixing commit by parsing the commit messages for relevant keywords. Then, SZZ identifies the error-introducing commit by *blaming* the changed lines in the error-fixing commit. Blaming or annotating is a function of the repository to determine the commits that modified or added any given line of code. Fundamentally, the SZZ-algorithm assumes that the lines changed in the fix contain the fault location and determines which commit changed these lines previously to introduce the error. However, we find that the changed lines in the error-fixing and error-introducing commits in CoREBench do not even overlap for one in every three regression errors.

Three benchmarks that contain *actual program errors* are iBugs [7], BugBench [18], and Marmoset [27]. iBugs consists of a large number of real bug fixes in the version history of two Java projects, AspectJ and Rhino. For some bug fixes, the benchmark also maintains those test cases that were submitted with the fix. BugBench consists of mostly memory-related errors while Marmoset contains errors extracted from student projects and may not contain a representative sample of actual program errors. In contrast to these, our CoREBench allows us to study *regression* testing and regression debugging techniques as well as the evolution of software errors over several program versions for up to eight years from error-introduction to fix.

Two benchmarks that contain *seeded regression errors* introduced by manual fault seeding are the Siemens Suite [13] and SIR [9]. Both are discussed in detail in the empirical section of the present paper.

7. CONCLUSION

The research on and development of automated techniques to expose, locate root-causes of, and repair *regression errors* requires an understanding of the inherent nature of such errors. In order to develop automated regression testing, debugging, and repair techniques, we need to be aware of the underlying, general properties of regression errors.

In this paper, we advertise the study of regression errors with a varying degree of *complexity* and propose the subjects in CoREBench, as a collection of actual regression errors, for such controlled studies. We have analyzed the two most popular benchmarks, the Siemens Suite and SIR, which contain regression errors with a varying degree of *detectability* and found that these errors are often *simple* and generally significantly *less complex* than actual regression errors. In other words, their *fixes* were required to be *less substantial*.

Our novel measure of error complexity enables research and development of regression testing, debugging, and repair techniques *that account for a varying degree of complexity*. We may ask more refined research questions, such as:

- **What is the root-cause of a complex error?** If an error requires a substantial fix, can we assume that there is just one faulty statement causing the error? Are faults of complex errors localizable [19]? The answers may have implications for the performance of (statistical) debugging techniques.

- **Test suite adequacy to expose complex errors?** Some widely used metrics of test suite adequacy, such as statement or branch coverage, are based on the implicit assumption that errors are often simple, i.e., that the fault is localizable within some branch or statement which is covered. Now we may be able to investigate the effectiveness of coverage-adequate test suites w.r.t. a varying degree of error complexity and may develop more sophisticated adequacy-criteria that account for complex errors. Moreover, for the study of the relationship between simple and complex errors (e.g., see coupling effect [24]), we can take error complexity as an *ordinal* rather than a *dichotomous* measure.

- **How do we repair complex errors?** By definition, the fix of complex errors is more substantial than for simple errors. The research community has made significant progress understanding the automated repair of (simple) localizable errors [23, 17]. Now we may be able to evaluate the *efficiency* of such repair techniques w.r.t. a varying complexity of the repaired errors.

The artifact evaluation committee of ISSTA 2014 has found CoREBench and the CyCC tool to exceed expectations. We hope that our novel error complexity metric and the many actual regression errors in CoREBench spur a multitude of studies of regression testing, debugging, and repair techniques and of those assumptions underlying these techniques so as to better understand the nature of complex regression errors.

8. ACKNOWLEDGMENTS

We thank Dr. Konstantin Rubinov and Dr. Jooyong Yi for their insightful comments on earlier drafts of this paper. This work was partially supported by Singapore's Ministry of Education research grant MOE2010-T2-2-073. The first author is funded by an ERC advanced grant 'SPECMATE'.

9. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, 2005.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[3] J. M. Bland and D. G. Altman. Measuring agreement in method comparison studies. *Statistical Methods in Medical Research*, 8(2):135–160, June 1999.

[4] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 334–344, 2013.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[6] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[7] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, 2007.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

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

[10] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.

[11] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 121–130, 2013.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, 1994.

[14] IEEE. *1003.1-1988 INT/1992 Edition, IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988)*. IEEE, New York, NY, USA, 1988.

[15] IEEE. Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[16] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, 2006.

[17] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.

[18] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[19] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 74–77, 2012.

[20] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[21] A. S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 342–352, 2011.

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, 2002.

[23] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, 2013.

[24] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, Jan. 1992.

[25] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 218–227, 2008.

[26] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, 2005.

[27] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with marmoset: An automated programming project snapshot and testing system. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, 2005.

[28] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:72–101, 1904.

[29] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.