Information and Software Technology 86 (2017) 37–53

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Towards an understanding of change types in bug fixing code

Yangyang Zhao[a], Hareton Leung[b], Yibiao Yang[a], Yuming Zhou[a,*], Baowen Xu[a]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[b] Department of Computing, Hong Kong Polytechnic University, Hong Kong

ARTICLE INFO

Article history:
Received 29 June 2016
Revised 11 February 2017
Accepted 13 February 2017
Available online 20 February 2017

Keywords:
Change
Bug fixing code
Empirical study
Understanding
Software quality

ABSTRACT

Context: As developing high quality software becomes increasingly challenging because of the explosive growth of scale and complexity, bugs become inevitable in software systems. The knowledge of bugs will naturally guide software development and hence improve software quality. As changes in bug fixing code provide essential insights into the original bugs, analyzing change types is an intuitive and effective way to understand the characteristics of bugs.

Objective: In this work, we conduct a thorough empirical study to investigate the characteristics of change types in bug fixing code.

Method: We first propose a new change classification scheme with 5 *change types* and 9 *change subtypes*. We then develop an automatic classification tool CTforC to categorize changes. To gain deeper insights into change types, we perform our empirical study based on three questions from three perspectives, i.e. across project, across domain and across version.

Results: Based on 17 versions of 11 systems with thousands of faulty functions, we find that: (1) across project: the frequencies of change subtypes are significantly similar across most studied projects; interface related code changes are the most frequent bug-fixing changes (74.6% on average); most of faulty functions (65.2% on average) in studied projects are finally fixed by only one or two change subtypes; function call statements are likely to be changed together with assignment statements or branch statements; (2) across domain: the frequencies of change subtypes share similar trends across studied domains; changes on function call, assignment, and branch statements are often the three most frequent changes in studied domains; and (3) across version: change subtypes occur with similar frequencies across studied versions, and the most common subtype pairs tend to be same.

Conclusion: Our experimental results improve the understanding of changes in bug fixing code and hence the understanding of the characteristics of bugs.

1. Introduction

1.1. Motivation

With the increasing size and complexity of software systems, bugs become inevitable in software development. Consequently, to enhance software quality, it is essential that bugs be understood and fixed as early as possible. As bugs are usually caused by specific issues and involve fixes in the related source code, the types of bugs highly correlate with the types of source code changes for bug fixing. For example, bugs caused by data issues are naturally fixed by modifying data-related code. Similarly, bugs manifested by interface errors are usually fixed by correcting interface-related code. In other words, changes in bug fixing code provide essential insights into the original bugs. Therefore, analyzing change types in bug fixing code should be an intuitive and effective way to understand the characteristics of bugs.

The study on changes in bug fixes could enable practitioners to better understand the general characteristics like the nature of changes, the most common kind of change types, and the frequencies of change types across versions, projects, or domains. Since bug-fixing change types are associated with the specific kinds of source code that are faulty [7–9], the most common change types indicate which kinds of code are most likely to be faulty. When developing a new software project, this knowledge could guide developers to pay more attention to the statements with the highest chance of being faulty. In particular, if the frequencies of bug-fixing change types are found similar across versions, projects, or domains, developers could learn from the previous bug fixing activities and avoid making similar mistakes again. Besides, when a code bug occurs, this knowledge can also guide developers to prioritize source code to be reviewed, debugged, or tested.

* Corresponding author.
E-mail address: cs.zhou.yuming@gmail.com (Y. Zhou).

http://dx.doi.org/10.1016/j.infsof.2017.02.003
0950-5849/

Despite the above benefits, little effort has been devoted to thoroughly investigate changes in bug fixing code with a broad range of projects, mainly due to the following challenges: (1) lack of adequate bug-fixing data. To draw meaningful conclusion, it is necessary to collect a large number of bug fixes for empirical study. However, as many projects did not provide adequate public history data for analysis, it tends to be difficult to identify bug-fixing changes in these projects; (2) lack of a well-accepted taxonomy for bug fixing changes. There have been a number of studies on categorizing bugs, such as cause-driven or document-driven fault classification [1–3], and fault trigger based classification [4,5]. However, most of them are not associated with source code, thus offering little help on code review; and (3) lack of automatic tool support. Manual change categorization is subjective and unreliable. If taxonomies do not provide concrete guidelines on classification, a change is likely to be categorized differently by different practitioners based on their own understanding and background. Besides, manual change categorization is time-consuming, thus it is infeasible to study a broad range of systems. Consequently, to comprehensively investigate bug fixing changes, it is necessary to develop automatic tools to overcome the problems of manual categorization.

As a result, the primary motivation of this study is to overcome the above challenges and perform a thorough investigation with a relatively large number of projects to improve the understanding on change types in bug fixing code.

1.2. Contributions

As the change types in bug fixing code are essential reflections of the original kinds of bugs, we attempt to study bugs from the view of fine-grained code changes in the bug fixing process. To classify code changes for faulty functions, we develop a taxonomy of *change types*: Data, Computation, Interface, Logic/control, and Others, which are language independent. To support a detailed analysis in the follow-up experiments, we further subdivide the five change categories into 9 *change subtypes,* i.e. changes on data declaration or initialization statements (CDDI), changes on assignment statements (CAS), changes on function declaration/definition statements (CFDD), changes on function call statements (CFC), changes on loop statements (CLS), changes on branch statements (CBS), changes on return/goto statements (CRGS), changes on preprocessor directives (CPD), and other changes (CO).

In this study, we study the code change types based on projects developed in C language. The primary reasons are that: (1) there is no previous work devoted to studying the bug fixing code changes in these systems; and (2) there is currently no automatic change type classification tools for C language (to the best of our knowledge). We develop an automatic tool CTforC based on Coccinelle [12,13]. CTforC has three main components, i.e. change location, change pattern detection, and change type classification. It can automatically identify faulty functions, match changed code, and finally output change types. The evaluation results show that CTforC can achieve accuracies consistently above 98% when compared to manual change classification.

With the defined change types, firstly, we study 11 well-known open-source systems to explore the general characteristics of change types across projects; Secondly, we investigate 3 domains, i.e GNU, Apache, and Tool, to examine whether the frequencies of change subtypes are domain specific; Finally, we examine two systems with multiple versions (i.e. Apr versions and Libav versions), to study whether change subtypes occur with similar frequencies in different versions of the same system.

Based on 17 versions of 11 C systems with thousands of faulty functions, we have the following key findings.

- Across projects. There are general characteristics of change types across studied projects. More specifically, interface related code changes are usually the most frequent bug-fixing changes, accounting for 74.6% on average (Finding1); The frequencies of change types are significantly similar across most studied systems (Finding 2); A significant number of faulty functions (65.2% on average) are fixed by only one or two change types (Finding 3); Besides, function call statements are very likely to be changed together with assignment statements or branch statements in most studied systems (Finding 4).
- Across domains. The frequencies of change subtypes do not vary substantially, but share significantly similar trends across studied domains (Finding 6), and changes on function-call statements are the most commonly observed changes regardless of the domain of the studied projects (Finding 5).
- Across versions. The frequencies of change subtypes are similar across different versions of the studied system (Finding 7), and the most common subtype pairs (i.e. a pair of change subtypes which occur together in the same function) are always the same across studied versions (Finding 8).

The above findings provide a comprehensive view on the characteristics of change types, improve the visualization of how bugs were repaired, and gain deeper insights into the nature of bugs from source code perspective. We believe these results are valuable to guide both software development and future researches in this direction.

Paper outline: The remainder of the paper is organized as follows. Section 2 describes the fundamental concepts, our proposed change classification scheme, and the classification methods for special cases. Section 3 presents the experimental methods of our study, including the research questions, studied systems, and our automatic classification tool. Section 4 reports the evaluation results of our automatic tool and presents experimental results in detail for each research question. Section 5 analyzes the threats to validity of our study, followed by the introduction of related work in Section 6. Finally, in Section 7, we give the conclusions and outline the directions for future work.

2. Change types

In this section, we first introduce the concepts used in our study. Then, we propose a new classification scheme and introduce each change type in detail. Finally, we illustrate the classification methods for special cases.

2.1. Concepts

For better understanding, it is necessary to first define the special concepts used in our study. It is noteworthy that, in this study, we use “bug”, “defect”, and “fault” interchangeably, since “faults” and “defects” are synonyms of bugs [6].

- Bug: errors in a software system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.
- Bug version: a released version with bugs.
- Fix version: a subsequent version of the bug version, which is released purely for fixing bugs.
- Faulty function: a function in the bug version which is changed in its corresponding fix version.
- Fixing Code Change: code change between the bug version and fix version, like adding, deleting, modifying, or moving source code. (Note that, the fixing code changes are occasionally abbreviated to changes in this study)
- Change type: the type of a fixing code change.

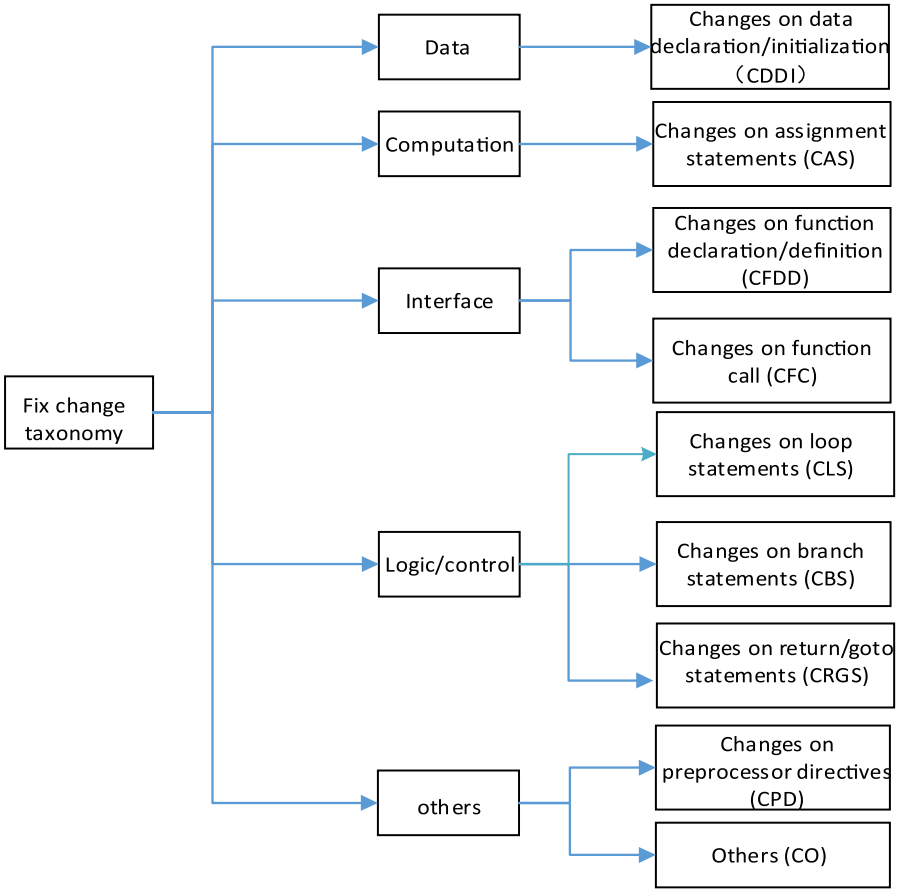

Fig. 1. Fixing code change taxonomy.

- Bug-fixing behavior: a set of fixing code changes to fix the bugs existing in the bug version.

2.2. Change type definition

It has been pointed out that there are strong associations between bugs and the corresponding incorrect source code [7–9]. In this study, we attempt to study bugs from the viewpoint of fixing code changes between the bug version and fix version. As categorization is the basic cognitive process for recognizing, differentiating and understanding [44], to better study fixing code changes, we attempt to arrange the changes into groups based on the similar characteristics. We began this phase by individually reviewing a set of faulty modules randomly, including checking the faulty modules and the corresponding fixed versions, locating the fine-granted code changes, recognizing the common features, and deciding the essential types. To mitigate the risk of bias arising from missing and incorrect classification, we augmented the reviewing rates to improve our classification rules and remove spurious classification as much as possible. The classification scheme evolved over the process of manual review, and was gradually refined to cover more kinds of changes. Finally, we come up with 5 categories and 9 subtypes based on our understanding of the code changes, which tend to be stabilized after discussing and double checking 500 faulty modules. The choice of categories is mainly based on the syntactic structure of programming languages and inspired by the fault taxonomies in the literature [7,10].

Fig. 1 illustrates the fixing code change taxonomy. It serves to divide fixing code changes into different categories based on the static pattern matching of changed code, while ignoring the dynamic information. In Fig. 1, there are totally five main *change types*, including Data, Computation, Interface, Logic/control, and others. To support a more detailed study in the follow-up experiments, we further subdivide them into nine subtypes, i.e. CDDI, CAS, CFDD, CFC, CLS, CBS, CRGS, CPD, and CO (which are called "*change subtypes*" in the remainder of this paper). These change subtypes, which are simple, follow the nature of source code and indicate the fine-granted features of bug fixing activities. Each type is detailed below. To make the definitions easier to understand, we give 11 examples of code fractions. All of them are generated by the GNU diff tool, and represent the code differences between the bug version and the fix version. "−" marks the faulty statements deleted from the bug version and correspondingly "+" marks the new statements added in the fix version. There are 3 kinds of change operations, including modification, addition, and deletion. The modification involves both "−" and "+", while addition only involves "+" and deletion only involves "−".

2.2.1. Data

As data should be declared and initialized before being used, the data category includes changes on data declaration or initialization statements (CDDI). Example-1 shows an example of CDDI. As can be seen, the declared type of data `a` is "`int`" in the bug version, which is modified as "`float`" in its fix version. As data-related faults usually occur in data statements, this change type also indicates the occurrence of data-related defects.

Example-1:

```
− int a;
+ float a;
```

2.2.2. Computation

This category includes the changes on assignment statements (CAS), i.e. adding/deleting assignment statements, or modifying the

equation on the right hand side with the left hand side the same in both the bug and fix versions. As Example-2 shows, data a is assigned a wrong value "b+1″ in the bug version, which is modified as "b−1″ in the fix version. Computation-related faults refer to bugs that lead to a wrong value being assigned to a variable. Such kind of faults can often be fixed by CAS.

Example-2:

```
− a=b+1;
+ a=b−1;
```

2.2.3. Interface

If a function has an interface-related fault, it may be caused by wrong definition or faulty function dependency on other functions. As Example-3 shows, a call with incorrect parameter number "m(a, b)" leads to an unexpected function loading, which is finally fixed by modifying its parameter number. The faults related to interface issues can be fixed by adding/deleting/modifying the function declaration/definition or function call statements. As a result, this category is further subdivided into changes on function declaration/definition (CFDD) and changes on function call (CFC).

Example-3:

```
− m(a, b);
+ m(a);
```

2.2.4. Logic/control

Logic and control statements are essential parts of any software. Faults occurring in these statements may cause the incorrect execution sequence or an abnormal state. More specifically, based on the statements types, logic/control related code changes are further subdivided into the following three types:

2.2.4.1. Changes on loop statements (CLS). Loops are important to perform tasks repeatedly in programming. A loop statement executes its body multiple times until its loop condition become false. There are three loop types in C language, including `for`, `while`, and `do-while`. Besides, loop control statements, like `continue` and `break` statements, are used to change execution from its normal sequence. Changes on these loop statements are categorized into CLS. Faults in these statements are mainly caused by incorrect loop condition, infinite loop or wrong loop termination. Example-4 illustrates an array boundary exception, i.e. accessing an array `arraylist` outside of its bound, which is fixed by changing the maximal array index in the `for` loop condition in the fix version.

Example-4:

```
- for (int i=0; i <= length(arraylist);
  i++)
+for (int i=0; i < length(arraylist); i++)
  sum=sum+arraylist[i];
```

2.2.4.2. Changes on branch statements (CBS). Branch statements are decision making statements, such as `if` structure and `switch` structure, deciding what follow-up actions to take. Faults related to branch issues may be fixed by adding an `if` predicate to ensure a precondition is met first, removing an `if` predicate from the code it encloses, adding an `else` branch to cover another possible condition, modifying the `switch` condition, or adding `case` branch, etc. Example-5 illustrates a Null Pointer Exception, which is manifested by an invalid `foo` and finally fixed by adding precondition checking to ensure `foo` is valid before it is accessed.

Example-5:

```
+ if (foo) { foo->ok=1; +}
```

2.2.4.3. Changes on return/goto statements (CRGS). `Return` statements are used to terminate the current function and return a specified value. `Goto` statements can transfer the control from the current location to anywhere the label is specified in the program. Changes on them are grouped into CRGS. Example-6 illustrates an example of changing return statement.

Example-6:

```
− return 0;
+ return 1;
```

2.2.5. Others

The above four kinds of code changes are commonly observed, and could cover most fixing code changes. The "Others" category contains bug fixing changes, which are relatively less common and cannot be classified into the categories above.

2.2.5.1. Changes on preprocessor directives (CPD, for C language). In C programs, there exist a lot of preprocessor directives. Preprocessor directives are lines included in the code preceded by a hash sign (`#`). These lines are not program statements but directives for the preprocessor, such as `#define`, `#if`, `#else`, `#elif`, `#ifdef` and `#endif` expressions. In practice, some bugs are fixed by changing a macro value, changing the condition expression of `#if`, or adding a `#else` branch, etc. This category covers all the fixing code changes on preprocessor directives (CPD). The change in Example-7 fixes the macro redefinition bug. `#if` precondition is added to check whether the macro `EXAMPLE_H` has been defined.

Example-7:

```
+ #if !defined(EXAMPLE_H)
  #define EXAMPLE_H 0
+ #endif
```

2.2.5.2. Others (CO). Besides, changes about code movement are under this category.

2.3. Classification methods for special cases

When categorizing changes, there exist two special cases involving nesting, which may cause perplexity to classification decision. To leave little room for confusion, the following examples illustrate the classification methods for each case.

- Case 1: When function call is nested within other kinds of statements.

  Function calls are usually written inside other kinds of statements, such as data initializations, assignments, `if/for/while/` conditional predicates, and so on. When a statement involving function-calls is changed, it will be categorized according to the kind of the statement and whether the function-calls are also changed or not.

  Example-8 illustrates one case when a function call occurs in an assignment statement. In this example, a fault is fixed by changing an assignment statement, which involves a function call *m(0)*. In our study, this change is only classified into the computation category (CAS) without Function call (CFC), since the function call is not changed in the fix version.

  Example-8:

  ```
  − a=m(0)+1;
  + a=m(0)+2;
  ```

  However, if the fault of an assignment statement is just caused by the function call itself, as shown in Example-9, *m(0)* is incorrect and modified to *m(1)*, such change is classified under both computation (CAS) and Function call (CFC) categories. Besides, if the fault is fixed by modifying both function call statement and assignment statement as shown in Excample-10, it will be classified into CAS and CFC simultaneously.

Example-9:
```
− a=m(0)+1;
+a =m(1)+1;
```
Example-10:
```
− a=m(0)+1;
+ a=m(1)+2;
```

- Case 2: Nesting between different code structure

There exist many nesting cases in source code such as nested loops and nested `if...else` statements. The classification of these cases only considers the changed statements regardless the outer structure. In Example-11, an `if` statement is nested in the `for` structure, and the fault only occurs on the `if` statements, while the `for` structure is correct. In this case, the involved change type is only changes on branch statements (CBS).

Example-11:
```
  for(i=0; i < 3; i++){
- if (a < 9)
+ if (a <= 9)
    a=a+i;
  }
```

3. Experimental methods

In this section, we first highlight the three research questions of our study in Section 3.1, and then introduce the studied systems in Section 3.2. In Section 3.3, we describe our automatic change code classification tool.

3.1. Research questions

To gain deeper insights into the characteristics of change types and hence bugs, we consider the following research questions, which have not been studied or not thoroughly studied before.

- RQ1 (Across projects): Are there general characteristics of change types across projects?

The general characteristics of change types are tightly correlated with bug characteristics. If the frequencies of change types tend to be similar across systems, we can rank the change types from most to least common, and then pay more attention to the most common changes to reduce the most frequently encountered bugs during software development.

- RQ2 (Across domains): Do the frequencies of change subtypes vary substantially across different domains?

Software from different domains may involve different applications, with different performance requirements. For example, a data management system sets strict demands on the high accuracy of data and computation, hence it may possibly involve low percentage of data and computation related bugs. It makes intuitive sense that their encountered bugs may have different distributions from other systems. However, there is no general consensus on this issue. Consequently, it is necessary to examine whether the frequencies of change types are domain specific.

- RQ3 (Across versions): Do change subtypes occur with similar frequencies in different versions of the same system?

If the frequencies of change subtypes share similar trends across different versions, we can refer to the bug-fixing behaviors in previous versions when fixing the bugs in the current version. The results from this study will provide insights on the similarity of bug-fixing behaviors between versions of the same system.

3.2. Studied systems

To investigate the characteristics of bug fixing change types (RQ1), we study eleven well-known open-source systems written in C, including Bash, Vim, Subversion, Gstreamer, Gst-plugins-base, Libgcrypt, Make, Gimp, Wine, Libav, and Apr. For RQ2, these projects are divided into three domains based on implementation environment and application, i.e. GNU, Apache, and Tool. More specifically, Bash, Make, Libgcrypt, and Gimp are GNU projects. Among them, Bash is a command language interpreter as a free software replacement for the Bourne shell; Make controls the generation of executables and other non-source files of a program from the program's source files; Libgcrypt is a general purpose cryptographic library based on the code from GnuPG; and Gimp is a GNU image manipulation program used for image retouching and editing, resizing, photo-montages, and more specialized tasks. In addition to these four GNU projects, Subversion and Apr are Apache projects. Subversion is a well-known software versioning and revision control system, and Apr (short for Apache Portable Runtime) is a supporting library for the Apache web server. To cover more software application domains and make our investigation more comprehensive, we additionally employ a domain called "Tool", which includes 5 widely used tools, i.e. Vim, Gstreamer, Gst-plugins-base, Wine, and Libav. Among them, Vim is a highly configurable text editor for UNIX systems to enable efficient text editing. Gstreamer is a library for constructing graphs of media-handling components, and Gst-plugins-base is a well-groomed and well-maintained collection of GStreamer plug-ins and elements. Wine is a free software which enables Linux, Mac, FreeBSD, and Solaris users to run Windows applications without a copy of Microsoft Windows. The last one is Libav, a software project that produces libraries and programs for handling multimedia data.

Table 1 summarizes the descriptive information for these systems. The first two columns report the system name and its corresponding domain. The following four columns respectively list

Table 1
Descriptive information of the eleven systems under study.

System	Domain	Subject release				No. of patches or the latest bug-fixing release version
		Version	Size(KSLOC)	#File	#Function	
Bash	GNU	3.2	57	338	1946	51
Vim	Tool	6.0	116	93	3017	270
Subversion	Apache	1.2.0	181	905	6864	Subversion1.2.3
Gstreamer	Tool	1.0.0	109	395	4718	Gstreamer1.0.10
Gst-plugins-base	Tool	1.0.0	174	520	5979	Gst-plugins-base1.0.10
Libgcrypt	GNU	1.2.0	23	91	692	Libgcrypt1.2.4
Make	GNU	3.79	14	57	310	Make3.79.1
Gimp	GNU	2.0.0	434	1774	12792	Gimp 2.0.6
Wine	Tool	1.6.rc1	2191	3685	87263	wine 1.6
Libav	Tool	0.8	356	1425	10574	Libav0.8.17
Apr	Apache	1.4.2	31	362	1765	Apr 1.4.8

Table 2
Descriptive information of multiple versions of Libav and Apr.

System	Subject release					Latest bug-fixing release	
	Version	Release date	Size(KSLOC)	#File	#Function	Version	Release date
Libav	0.7	2011-06-20	337	1353	9971	0.7.7	2013-02-02
	0.8	2012-01-21	356	1425	10574	0.8.17	2015-03-11
	9	2013-01-05	396	1618	10412	9.18	2015-03-11
	10	2014-03-23	446	1780	10899	10.6	2015-03-11
	11	2014-09-13	464	1906	11189	11.3	2015-03-11
Apr	1.2.1	2005-08-18	28	346	1636	1.2.12	2007-11-15
	1.3.0	2008-05-30	30	358	1725	1.3.12	2010-02-11
	1.4.2	2010-01-26	31	362	1765	1.4.8	2013-06-20

the studied version, total size, the number of files, and the number of functions. As can be seen, most of these projects are moderate-to-large-sized systems. To perform our experiments, we need to collect the patch data or identify the fix version. One practical approach is mining the bug fixing information from software repository [11]. We search history database, identify the purpose of each release, and then find bug fixing releases. For example, from the change logs of Libav, we obtain the information that "between major releases, point releases will appear that add only bug fixes but no new features", which suggests that the latest point release can be regarded as the fix version of its corresponding major release. As a result, the fix version of Libav0.8 is its latest point release Libav0.8.17. With this method, we are able to find the corresponding fixing patch data or the latest bug-fixing version. The results are shown in the last column of Table 1. Specifically, Bash3.2 and Vim6.0 respectively have 51 and 270 fixing patch files. Except them, all the other systems have their own latest fix versions.

To study change types across versions (RQ3), we use two systems: (1) a set of Libav versions, including Libav0.7, Libav0.8, Libav9, Libav10, and Libav11; (2) a set of Apr versions, including Apr1.2.1, Apr 1.3.0, and Apr1.4.2. Similarly, the descriptive information of these projects is shown in Table 2.

We choose these systems as our research subjects for the following reasons: (1) they are open-source systems, and widely used in both practical application and research areas. As such, it is relatively easy to externally validate our empirical results by other researchers; (2) they are non-trivial systems, and belong to different problem domains with different development and maintenance styles. Therefore, our investigation on the characteristics of change types based on them is more comprehensive and meaningful; and (3) their fixing patches or bug fixing versions are publicly available, which enable us to locate the faulty functions and study code fixing changes in these systems.

3.3. Change type classification

To reduce the effort for change classification and overcome the subjectivity of manual categorization, we develop an automatic tool called CTforC based on Coccinelle to categorize code changes.

3.3.1. Coccinelle

Coccinelle[1] is a program matching and transformation engine that can analyze and transform C code according to specified rules or semantic patches [12,13]. It provides the language SmPL (Semantic Patch Language) for specifying desired matches and transformations. Semantic patches are much more powerful than patches or regular expressions. Each rule of SmPL is composed of two parts: (1) a set of metavariable declarations surrounded by a pair of @@. Metavariable types can be expression, statement, type, constant, and so on. Particularly, a position metavariable is used by attaching it using @ to any token, including another metavariable. Its value is the position (file, line number, etc.) of the code matched by the token; and (2) a code matching and transformation specification. The transformation specification essentially has the form of C code, except the usage of special annotations like +, −, *, ?, and |. For example, lines to remove are annotated with − in the first column, and lines to add are annotated with +. Specifically, * is used for semantic matching, i.e., highlighting the fragments annotated with *, without performing any modification of the matched code. As our aim of employing Coccinelle is to find the code patterns in the faulty code without modifying the code, hence * is widely used in our experiment.

```
@dataDeclare@
type T;
identifier X;
position P;
@@
*T@P X;



@script:python depends on dataDeclare@
P << dataDeclare.P;
@@
    print("Data declare at "+P[0].line);
```

Fig. 2. An example for finding data declarations.

Another important feature of Coccinelle is that it can embed Python/OCaml code. Python/OCaml rules inherit metavariables, such as identifier or token positions, from SmPL rules. The inherited metavariables can then be manipulated by Python/Ocaml code. In other words, SmPL allows matching code, but not performing computation, while Python/OCaml allows performing computation, but not matching code.

Fig. 2 shows an example of Coccinelle scripts, which attempts to look for all data declarations with the form of "`T X;`" and prints out its position. It consists of SmPL and Python code. In SmPL part, T is an arbitrary type, X is an arbitrary identifier, and P is a position matavariable. * annotates the data declaration pattern. When the pattern is matched, P will remember exactly what fragment of code was matched. In Python part, Python inherits P, and could do arbitrary computation on P, especially printing Observations.

A precondition of Coccinelle execution is that the C code and semantic patches are all syntactically correct. An essential command is *spatch*. When executing *spatch* command, firstly both the semantic patch file and the source code are parsed. If it succeeds, the code which matches the pattern formalized in the semantic

[1] http://coccinelle.lip6.fr/

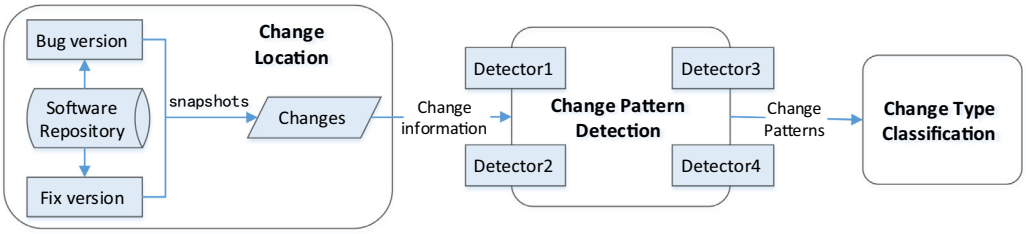

Fig. 3. The framework of CTforC.

Algorithm 1:
Change-Pattern-Detector.

Input: a sequence of change locations *CL*, file *F* for storage	
Step	
1:	Initialize global state, and preprocess the incoming data
2:	Declare metavariables, including pattern metavariables and position metavariables.
3:	Define code matching specifications.
4:	Use Python to inherited metavariables.
5:	Manipulate metavariables and global values.
6:	Check conditions based on our taxonomy and the processed *CL*.
7:	Access global state and store the information of matched change patterns into *F*.

patch file will be recognized and then highlighted or modified according to the rule.

3.3.2. Classification tool CTforC

In many previous works, Coccinelle has been widely used for automatically finding code containing bugs or requiring collateral evolutions, and then systematically fixing bugs, or performing collateral evolutions [14–17]. Instead of fixing bugs or code refactoring, we adapt Coccinelle for static analysis of known faulty code to find existing code patterns without any modification. Based on Coccinelle, we build a tool CTforC to classify bug fixing code changes automatically, which is only for C language. CTforC has three main components, i.e. change location, change pattern detection, and change type classification. Fig. 3 illustrates how these components interact with each other.

Change location. For each system, we mine its software repository to extract its bug fix information. Among the studied systems, two have patch files and the others have corresponding bug-fixing release versions available. As a result, we can get two code snapshots for each system, representing the bug version and the fix version respectively. With the code snapshots, based on GNU diff tool, we can locate the changed functions and output the change information for each faulty function.

Change pattern detection. This component takes the code snapshots of bug version and fix version and change location information as inputs, invokes detectors for static pattern matching, and then outputs the patterns existing in changed code. Attempting to cover all possible code patterns, we use the following Algorithm 1 *Change-Pattern-Detector* and build a set of detectors. As can be seen from the pseudocode of Algorithm 1, our *Change-Pattern-Detector* algorithm takes into account: (1) Initialization: initialize global state, and preprocess the incoming data; (2) SmPL: declare metavariables and define code matching specifications; (3) Python: manipulate metavariables and global states, and check conditions; and (4) Finalization: access the global state and store the information. More specifically, all detectors implement the interfaces of Coccinelle and adhere to the SmPL grammar. Each detector is a piece of Coccinelle scripts, devoted to matching a specific code pattern, and outputting a binary result (i.e. 1 or 0) which indicates whether the pattern exists or not. For example, if a faulty function was fixed by changing two code lines, i.e. "*int a;*" and "*for(i=0; i <=j; i ++)*" respectively, both the *data declaration* pattern detector and the *for structure* pattern detector will return 1. And hence the output of this component includes both data declaration pattern and for structure pattern. Actually, the script in Fig. 2, which attempts to find all data declarations with the form of "`T X;`", works in the same way as our built detectors and can somewhat be regarded as a simplified detector. The main difference is that the real detectors not only detect all matched patterns and obtain the needed information, but also need to check specific conditions, like whether the found patterns actually occur in the bug fixing code. If conditions are satisfied, the detector can decide that the faulty function involves the specific change pattern. Besides, it is important to note that, due to the constraint of Coccinelle, all the incoming C source code and the detectors built should be syntactically correct, which is the precondition of this component. A well-built detector can be applied to new projects developed in C languages, regardless the coding styles, if the source code is grammatically correct.

Change type classification. The main role of this component is to group the identified change patterns into categories. As introduced in Section 2, we have defined five change types with nine subtypes. However, the change patterns detected by the previous component are more fine-grained than our defined change types. In other word, there is a one-to-many relationship between change types and change patterns. For example, all the patterns of *for* structure, *while* structure and *do/while* structure should be classified into one subtype, i.e. changes on loop statements (CLS). As a result, with the detected change patterns from the previous component, we still need to map each pattern to its corresponding subtype. In this component, we build classifiers to group the detected change patterns into the 9 subtypes according to our taxonomy. For example, if a faulty function was fixed by multiple kinds of change patterns (e.g. *"for(i=0; i <= j; i ++)"*, *"while(condition)"* and *"int var1;"*), all of them will be considered. And after matching each pattern to its corresponding change type, the changes for fixing this function are classified into both CDDI (changes on data declaration or initialization statements) and CLS (changes on loop statements).

4. Experimental results

In this section, we firstly examine our automatic tool CTforC, and then present in detail the experimental results respectively for each research question. More specifically, in Section 4.1, we report the effectiveness of CTforC. In Section 4.2, we show the results from studying the general characteristics of change types across studied systems. In Section 4.3, we report the results from investigating the change types across different domains. Next, we list the observations from studying the change types across versions in Section 4.4. For each research question, we decompose the results through a series of findings and implications for particular aspects of bug-fixing behaviors. We focus on key findings which indicate more general or practically valuable characteristics of change types. Last, we summarize the whole observations in Section 4.5.

Table 3
Evaluation results of CTforC.

System	#FaultyFunc	#ParsedFunc	ParsedRate	#realTypeSum	Accuracy
Bash3.2	62	60	97%	134	100%
Libgcrypt1.2.0	84	84	100%	182	98.4%
Make3.79	45	43	96%	112	99.1%
Gimp2.0.0	488	487	99.80%	1075	99.9%
Vim 6.0	252	246	98%	668	99.7%
Subversion1.2.0	44	44	100%	138	100%
Gstreamer1.0.0	140	140	100%	285	100%
Gst-plugins-base1.0.0	207	205	99%	477	99.8%
Wine1.6.rc1	409	408	99.80%	802	99.5%
Libav0.8	700	693	99%	1941	99.7%
Apr1.4.2	105	105	100%	210	98.6%

4.1. Effectiveness of CTforC

We manually evaluate the effectiveness of our change type classification tool CTforC introduced in Section 3.3. This evaluation involved one professor, two PhD candidates, and another graduate student. All of them have at least 3 years' experience using C programming language, and didn't participate in the proposition of the taxonomy and the development of our tool. Before performing the manual classification, we introduced the taxonomy to these subjects to ensure they fully understand the meaning of each type. And then we generated the diff hunk for each faulty function. The PhD candidates and graduate student individually review the diff hunks, and categorize the code changes according to our taxonomy manually. Those functions with different opinions were identified and further checked by the professor. With discussion and double checking, consensus is reached and all changes are finally classified. These manual classification results are used as the ground truths as well as the benchmark for comparison.

As our detectors are built without prior knowledge of the changes in our studied projects, we can use our studied projects in Table 3 for evaluation. The subjects manually reviewed total 2536 faulty functions. And about 10% functions with divergences were re-checked by the professor. Table 3 shows the overall evaluation results of CTforC. The first column lists the system, and the following seven columns are divided into two parts: the first part reveals how many faulty functions are identified, and how many of them are used in our study; the second part shows how effective CTforC is. More specifically, in the first part, the first column reports the number of the located faulty functions. The second shows, among these faulty functions, how many functions can be parsed successfully by Coccinelle. The following column is the corresponding percentage. As mentioned in Section 3.3.1, one essential condition of Coccinelle that must be satisfied is that the C files are syntactically correct and can be parsed successively. After preprocessing, we find that there are still a few ungrammatical functions. Taking Bash3.2 for example, two faulty functions are unavailable, since they involve some fragmentary and complex preprocessor directives, which Coccinelle fails to parse. However, as can be seen, such functions only account for a small percentage (less than four percent) in several systems, which we believe would not make a great influence on our study. As a consequence, we ignore them and use the functions in the column of "*#parsedFunc*" for the further experiments. In the second part, the first column reveals the number of real change types existing in the successively parsed functions, which are classified by ourselves. The last column shows the evaluation results in terms of the common evaluation indicator *Accuracy*. *Accuracy* shows the proportion of true results (both true positives and true negatives) among the total number of cases examined. The accuracies are obtained by comparing the change classification results of CTforC with the manual change classification results. From the results, we can see that CTforC can achieve accuracies consistently above 98% across all the studied systems, with quite considerable low false positive rate and false negative rate. The small rate of missing or mistakes does not limit CTforC's applicability. What's more, in some systems, such as Bash3.2, Subversion1.2.0 and Gstreamer1.0.0, the classification results of CTforC are even totally the same as the real change types, with accuracies reaching 100%.

Overall, from the results of Table 3, we believe that CTforC is indeed effective, and the change types identified by CTforC are believable, which forms the foundation of the next experiments and ensures the reliability of our final conclusions. To save time and effort, for the other versions of Apr and Libav, we directly use the change types classified by CTforC, without further manual checking.

4.2. RQ 1: across projects

To gain deeper insights into the general characteristics of change types across projects, we study 11 systems from four perspectives: (1) the most common change type; (2) the frequency similarity of change subtypes; (3) the complexity of bug-fixing behaviors; and (4) the most common change subtype pairs. The main observations are presented as follows.

4.2.1. The most common change type

Table 4 summarizes the frequency of each change type. The first and second columns respectively list the system name and the total faulty function number. As mentioned in Section 4.1, these faulty functions used in our study are parsed successfully by Coccinelle. The 3rd to 7th Columns list the number (percentage) of functions that involve the corresponding change type. The largest percentage in each row is marked in bold. Particularly, as each function may involve more than one change type, the sum of frequencies of all change types can be more than 100%. For illustration, we take Bash3.2 as an example. There are 6 faulty functions involving changes on data statements, hence the frequency of *Data* category is 10% (i.e. 6/60*100%=10%). The frequencies of other types are also calculated in the same way. The sum of frequencies is 193.3% (10% + 48.3% + 45.0% + 68.3% + 21.7% = 193.3%), greater than 100%. From Table 4, it is interesting to observe that, except Bash3.2 and Make3.79, the most frequent change type is consistently *Interface* across all other studied systems, with a percentage ranging from 63.2% to 87.1%. Although the most common change type in Make3.79 is *Logic/control*, the frequency of *Interface* also reaches 81.4%, which means a considerable number of functions involve interface-related changes. Moreover, in Bash3.2, nearly half (45%) of faulty functions are related to interface issues. Besides, Bash3.2 and Vim6.0 have relatively higher percentages of functions involving "other" category (21.7% and 26.8% respectively). We further analyzed these functions, and found that 1) in Bash3.2, all these faulty functions were fixed by changing preprocessor di-

Table 4
The frequency of five defined change types.

System	#Faulty	The number(percentage) of functions which involve the corresponding change type				
	func	Data	Computation	Interface	logic/control	others
Bash3.2	60	6(10.0%)	29(48.3%)	27(45.0%)	**41(68.3%)**	13(21.7%)
Libgcrypt1.2.0	84	35(41.7%)	33(39.3%)	**60(71.4%)**	29(34.5%)	3(3.6%)
Make3.79	43	8(18.6%)	11(25.6%)	35(81.4%)	**39(90.7%)**	4(9.3%)
Gimp2.0.0	487	122(25.1%)	201(41.3%)	**394(80.9%)**	274(56.3%)	24(4.9%)
Vim 6.0	246	46(18.7%)	149(60.6%)	**172(69.9%)**	158(64.2%)	66(26.8%)
Subversion1.2.0	44	18(40.9%)	31(70.5%)	**38(86.4%)**	30(68.2%)	2(4.5%)
Gstreamer1.0.0	140	26(18.6%)	60(42.9%)	**122(87.1%)**	53(37.9%)	3(2.1%)
Gst-plugins-base1.0.0	205	55(26.8%)	99(48.3%)	**170(82.9%)**	91(44.4%)	15(7.3%)
Wine1.6.rc1	408	98(24.0%)	168(41.2%)	**258(63.2%)**	198(48.5%)	15(3.7%)
Libav0.8	693	209(30.2%)	296(42.7%)	**511(73.7%)**	501(72.3%)	8(1.2%)
Apr1.4.2	105	23(21.9%)	57(54.3%)	**83(79.0%)**	31(29.5%)	6(5.7%)

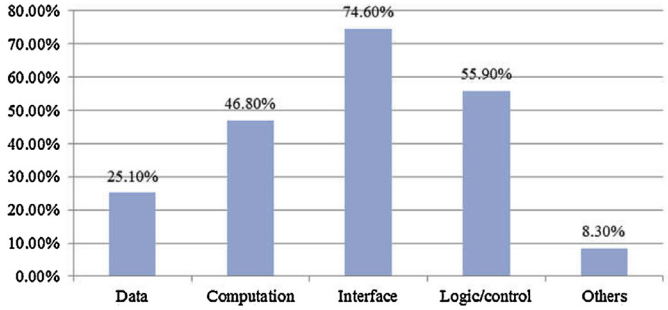

Fig. 4. The average frequency of change types across the 11 studied systems.

rectives; 2) in Vim6.0, most of them (92.5%) involved changes on preprocessor directives.

Fig. 4 further shows the average frequency of each type across all the studied systems. As presented in Fig. 4, on average, interface related changes have the largest percentage (74.6% overall), indicating interface issues should be highly concerned. This observation is consistent with the results of Basili and Perricone in 1984, which presented that interface faults were the most common faults [18]. The second notable type is Logic/control, accounting for 55.9%. This somewhat agrees with one finding of Marick's survey in [19] that logic faults were a most common source of faults. Furthermore, we find that, to some extent, the above observations are consistent with Pan's results in [10]. Their study found that the two most common individual patterns were MC-DAP (method call with different actual parameter values) and IF-CC (change in if conditional). Interestingly, these patterns are respectively categorized into Interface and Logic/control in our study, which are exactly the two most frequent changes in our study as well.

Finding 1*: For most studied systems, interface related code changes are the most frequent bug-fixing changes, accounting for 74.6% on average, and specifically contributes percentages ranging from 45% to 87% across all the studied systems.*

Implication 1*: Interface-related bugs are commonly observed. Practitioners should pay more attention to interface issues when developing software or fixing bugs.*

4.2.2. The frequency similarity of change subtypes

To examine the frequency similarity across all studied projects, we use Pearson's correlation, which is a most common measure of correlation to measure the degree to which the variables are related [20]. We first calculate the frequency of each change subtype for each system, and then compute the Pearson's correlations between the frequencies of change subtypes for different projects. The results are shown in Table 5. The significant correlations at the 0.05 level are marked by *, and the significant correlations at the 0.01 level are marked by **. As can be seen from Table 5, the frequencies of change types across most studied systems are significantly similar, with the correlation values ranging from 0.678 to 0.993, and the *p*-values less than 0.05. One exception is that the frequencies of change types in Libgcrypt1.2.0 differ from Bash3.2 and Make3.79, with the *p*-values larger than 0.05. In addition, we also employ Spearman's correlation to further study the frequency similarity. The results consistently show that the frequencies of change types in most studied systems are strongly associated with each other.

Finding 2*: The frequencies of change types across most studied systems are significantly similar.*

Implication 2*: The distribution of different kinds of bugs tends to be similar across projects. The existing bug fixing information hence can be used for reference when reviewing a newly developed system.*

4.2.3. The complexity of bug-fixing behaviors

In addition, we investigate the complexity of bug-fixing behaviors based on the number of change subtypes involved in each function, i.e. how many kinds of changed are needed for fixing a faulty function. If a faulty function is fixed by one or two kinds of changes, we think its fixing behavior is uncomplicated, otherwise complicated. Table 6 illustrates the distribution of faulty functions involving one, two, three, and more subtypes respectively. The column "#Func" lists the corresponding function number, and the "Percentage" column gives the percentage in all faulty functions. The largest percentage for each system is marked in bold. For example, 38.2% of faulty functions in Bash3.2 are finally fixed by two kinds of code changes, and 28.3% are fixed involving only one kind of code changes. From the results of Table 6, it is easy to observe that, in four systems, about 41.5% $\sim$ 47.6% faulty functions are fixed by just one subtype of changes. In another five systems, 31.7% $\sim$ 48.1% faulty functions are fixed by applying two subtypes of changes. In contrast, for Subversion1.2.0 and Libav0.8, a number of functions are fixed by more than three kinds of code changes. While on average, as Fig. 5 shows, faulty functions are more likely to be fixed by only one or two change subtypes, with a sum of percentage over 65%. Moreover, bug-fixing behaviors with three change subtypes are relatively less common across all the evaluated systems.

Finding 3*: A significant number of faulty functions (65.2% on average) in studied projects can be fixed by only one or two kinds of changes.*

Implication 3: *It is reasonable to assume that most bug fixing behaviors are relatively uncomplicated. When dealing with a new bug, it may be more efficient to first examine whether one or two kinds of changes can fix the bug.*

4.2.4. The most common change subtype pairs

A subtype pair is defined as a pair of change subtypes occurring together in the same function. In practice, we may focus more on

Table 5
Pearson's correlation between the frequencies of change subtypes in different projects.

System	Bash 3.2	Libgcrypt 1.2.0	Make 3.79	Gimp 2.0.0	Vim 6.0	Subversion 1.2.0	Gstreamer 1.0.0	Gst-plugins- base1.0.0	Wine 1.6.rc1	Libav 0.8	Apr 1.4.2
Bash3.2	–	0.503	0.716*	0.778*	0.914**	0.748*	0.711*	0.748*	0.816**	0.813**	0.678*
Libgcrypt1.2.0		–	0.650	0.869**	0.743*	0.835**	0.851**	0.865**	0.874**	0.705*	0.855**
Make3.79			–	0.878**	0.786*	0.745*	0.832**	0.838**	0.831**	0.798**	0.737*
Gimp2.0.0				–	0.927**	0.894**	0.967**	0.976**	0.979**	0.843**	0.925**
Vim 6.0					–	0.849**	0.887**	0.909**	0.931**	0.784*	0.882**
Subversion1.2.0						–	0.932**	0.952**	0.961**	0.905**	0.940**
Gstreamer1.0.0							–	0.993**	0.966**	0.838**	0.976**
Gst-plugins-base1.0.0								–	0.983**	0.852**	0.979**
Wine1.6.rc1									–	0.903**	0.943**
Libav0.8										–	0.765*
Apr1.4.2											–

* Correlation is significant at the 0.05 level (2-tailed).
** Correlation is significant at the 0.01 level (2-tailed).

Table 6
The distribution of faulty functions involving different number of change subtypes.

System	Number of involved change subtypes							
	One		Two		Three		More	
	#Func	Percentage	#Func	Percentage	#Func	Percentage	#Func	Percentage
Bash3.2	17	28.3%	23	**38.3%**	12	20%	8	13.3%
Vim 6.0	67	27.2%	78	**31.7%**	28	11.4%	73	29.7%
Subversion1.2.0	11	25%	7	15.9%	2	4.5%	24	**54.5%**
Gstreamer1.0.0	62	**44.3%**	39	27.9%	19	13.6%	20	14.3%
Gst-plugins-base1.0.0	85	**41.5%**	45	22%	25	12.2%	50	24.4%
Libgcrypt1.2.0	40	**47.6%**	16	19%	10	11.9%	18	21.4%
Make3.79	5	11.6%	25	**58.1%**	6	14%	7	16.3%
Gimp2.0.0	140	28.7%	220	**45.2%**	48	9.9%	79	16.2%
Wine1.6.rc1	194	**47.5%**	108	26.5%	52	12.7%	54	13.2%
Libav0.8	168	24.2%	167	24.1%	162	23.4%	196	**28.3%**
Apr1.4.2	32	30.5%	55	**52.4%**	6	5.7%	12	11.4%

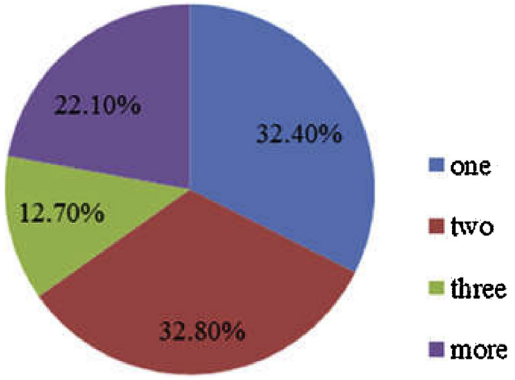

Fig. 5. The average frequency of involved change subtype number.

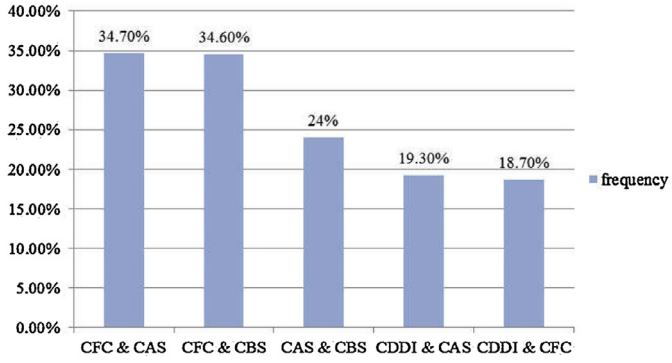

Fig. 6. Five most frequently observed change subtype pairs.

those changes that are most likely to occur together. As such, Fig. 6 illustrates the five most frequently observed change subtype pairs across all the studied systems. Among all the subtype pairs, CFC & CAS (34.7%), CFC & CBS(34.6%), CAS & CBS(24%), CDDI & CAS(19.3%), and CDDI & CSC(18.7%) are the top five pairs. More specifically, on average, function-call statements and assignment statements are both modified in the bug-fixing behaviors of 34.7% faulty functions (CFC & CAS). Similarly, function-call statements and branch statements are changed together in about 34.6% functions (CFC & CBS). Compared to these two pairs, the other pairs are relative less frequent. To understand this better, we manually examined a set of faulty functions which involve either "CFC & CAS" or "CFC & CBS". We found that, in a number of them, function calls are written inside assignment statements or branch statements, and many bugs in these statements are just caused by incorrect function calls. As mentioned in Section 2.3, when a statement involving function-calls is changed, it will be categorized according to the kind of the statements and whether the function-calls are also changed. In these cases, the changes are categorized into function-call category (CFC), as well as the certain type according to the kind of the statement itself (CAS or CBS), which increases the occurrences of CFC & CAS or CFC & CBS. This may be a possible reason why function calls are very likely to be changed together with assignment statements or branch statements.

Finding 4*: Function call statements are very likely to be changed together with assignment statements or branch statements.*

Implication 4*: When changing assignment statements or branch statements to fix bug, developers should pay attention to the function-call statements which may also be involved.*

Overall, based on the findings from the above four perspectives, we are able to answer RQ1:

There indeed exist general characteristics of change types across projects. The common characteristics are that, in most studied

Table 7
The frequency of each subtype for each domain.

System	#Faulty Func	CDDI	CAS	CFDD	CFC	CLS	CBS	CRGS	CPD	CO
(a) For GNU										
Bash3.2	60	6 (10.0%)	29 (48.3%)	1 (1.7%)	26 (43.3%)	6 (10.0%)	**39 (65.0%)**	14 (23.3%)	13 (21.7%)	0 (0.0%)
Make3.79	43	8 (18.6%)	11 (25.6%)	3 (7.0%)	**35 (81.4%)**	17 (39.5%)	27 (62.8%)	7 (16.3%)	4 (9.3%)	0 (0.0%)
Libgcrypt1.2.0	84	35 (41.7%)	33 (39.3%)	22 (26.2%)	**49 (58.3%)**	7 (8.3%)	26 (31.0%)	6 (7.1%)	2 (2.4%)	1 (1.2%)
Gimp2.0.0	487	122 (25.1%)	201 (41.3%)	31 (6.4%)	**389 (79.9%)**	30 (6.2%)	260 (53.4%)	19 (3.9%)	20 (4.1%)	4 (0.8%)
Average	–	23.80%	38.60%	10.30%	**65.70%**	16.00%	53.00%	12.70%	9.40%	0.50%
(b) For Apache										
Subversion1.2.0	44	18 (40.9%)	31 (70.5%)	3 (6.8%)	**38 (86.4%)**	7 (15.9%)	22 (50.0%)	17 (38.6%)	1 (2.3%)	1 (2.3%)
Apr1.2.1	98	26 (26.5%)	37 (37.8%)	7 (7.1%)	**73 (74.5%)**	4 (4.1%)	23 (23.5%)	13 (13.3%)	7 (7.1%)	0 (0.0%)
Apr1.3.0	24	8 (33.3%)	17 (70.8%)	2 (8.3%)	**23 (95.8%)**	0 (0.0%)	9 (37.5%)	7 (29.2%)	5 (20.8%)	0 (0.0%)
Apr1.4.2	105	23 (21.9%)	57 (54.3%)	2 (1.9%)	**83 (79.0%)**	4 (3.8%)	28 (26.7%)	10 (9.5%)	6 (5.7%)	0 (0.0%)
Average	–	30.70%	58.30%	6.00%	83.90%	6.00%	34.40%	22.60%	9.00%	0.60%
(c) For Tool										
Gstreamer1.0.0	140	26 (18.6%)	60 (42.9%)	5 (3.6%)	119 (85.0%)	4 (2.9%)	51 (36.4%)	17 (12.1%)	3 (2.1%)	0 (0.0%)
Gst-plugins-base1.0.0	205	55 (26.8%)	99 (48.3%)	6 (2.9%)	169 (82.4%)	17 (8.3%)	86 (42.0%)	29 (14.1%)	13 (6.3%)	2 (1.0%)
Wine1.6.rc1	408	98 (24.0%)	168 (41.2%)	23 (5.6%)	246 (60.3%)	24 (5.9%)	172 (42.2%)	57 (14.0%)	13 (3.2%)	2 (0.5%)
Vim 6.0	246	46 (18.7%)	149 (60.6%)	29 (11.8%)	168 (68.3%)	30 (12.2%)	149 (60.6%)	29 (11.8%)	61 (24.8%)	5 (2.0%)
Libav0.8	401	209 (30.2%)	296 (42.7%)	76 (11.0%)	482 (69.6%)	86 (12.4%)	447 (64.5%)	331 (47.8%)	3 (0.4%)	5 (0.7%)
Average	–	23.70%	47.10%	7.00%	73.10%	8.30%	49.10%	20.00%	7.40%	0.80%

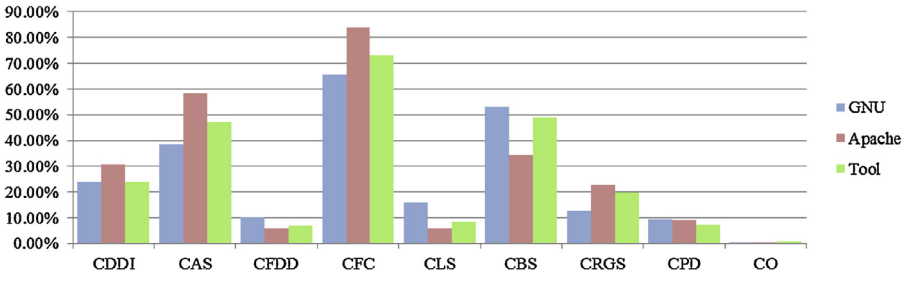

Fig 7. The comparison between different domains.

projects, (1) interface-related changes are the most commonly observed changes; (2) the frequencies of change types are significantly similar; (3) most faulty functions can be fixed by only one or two kinds of changes; and (4) function call statements are very likely to be changed together with assignment statements or branch statements.

4.3. RQ2: across domains

To investigate whether the frequencies of change subtypes vary substantially across different domains, we study three groups of systems, including 4 GNU systems, 2 Apache systems, and 5 Tool systems. The observations are as follows.

Table 7 summarizes the frequency of each change subtype. The first and second columns list the system name and the number of faulty functions. Each value in the 3rd to 11th columns indicates the number (percentage) of faulty functions that involve the corresponding change subtype. For each domain, we also list the average frequencies. The largest percentage in each row is marked in bold. With respect to GNU, CFC is the most common in most projects, accounting for 58.3 ∼ 81.4%. Only in Bash3.2, the most frequent changes are CBS, with a percentage of 65.0%. In the contrary, in terms of Apache and Tool, CFC in all the studied projects definitely has the highest frequency compared with other change subtypes. While on average, similar trends of change types on the three domains can be observed from Fig. 7, such as CFC is the most, and CAS and CBS are the second most or the third most common changes. These 3 types of changes are related to logical decisions and key computation of typical code. We attempted to count the number of different types of statements, and found that CFC, CAS and CBS tend to be more widely used than other statements. Therefore their chance of being changed may be higher if every statement has equal chance of being faulty.

Table 8
Pearson's correlation between the frequencies of change subtypes in different domains.

Domain	GNU	Apache	Tool
GNU	–	0.897	0.976
Apache		–	0.962
Tool			–

We also observe that the Apache projects tend to involve function call faults with a higher average frequency of 83.9%, as well as assignment faults accounting for 58.3%. Additionally, compared with other two domains, the branch issues occur in the GNU projects with a relatively higher average percentage of 53%. Overall, in spite of some subtle differences, there are also a lot of common characteristics about the bug fixes among the three domains, such as their top three frequent changes are the same, and the trends of the distribution of change types are similar. To further confirm this, we compute the Pearson's correlations between the average frequencies of change subtypes in different domains. As shown in Table 8, despite the difference in application domains, the average frequencies of change subtypes are significantly similar, as the correlation values are consistently over 0.89 with the p-values all less than 0.001. Furthermore, we also examine the frequency similarity across studied domains using Spearman's correlation, and the results consistently confirm the significant similarity.

These observations are consistent with Finding 1 and Finding 2, indicating that the frequencies of change subtypes share similar trends across different domains. The possible reason may be that, although the projects are in different domains, the frequencies of statement types may be similar, and each kind of statements may carry certain fault-prone possibility [24,43]. As such, the probabil-

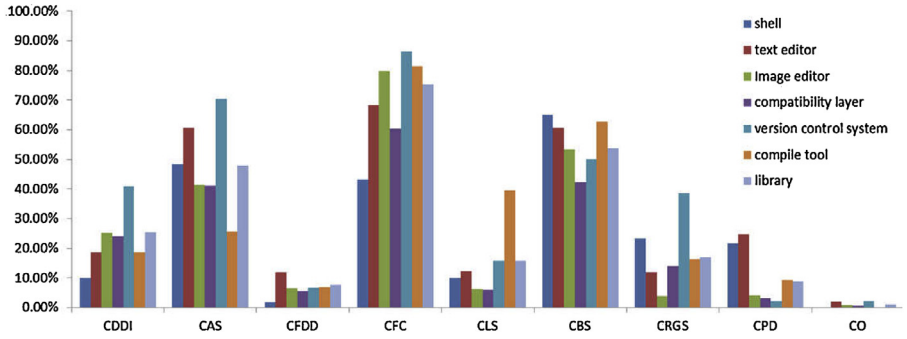

Fig. 8. The comparison between different domains.

ity of a kind of statements being faulty tends to be similar across domains, and the frequencies of change types indicate the frequencies of faulty code types. This conjecture certainly should be further confirmed in the future works.

Finding 5*: Changes on function call statements are the most commonly observed changes, and changes on assignment and branch statements are often either the second or the third most frequent changes regardless of the domain of the studied projects.*

Implication 5*: Function call statements should be highly concerned regardless of the application domain when developing software or fixing bugs.*

Finding 6*: The frequencies of change subtypes are not domain-specific, and exhibit significantly similar trends across studied domains.*

Implication 6*: The distribution of bugs may be similar across domains. The bug fixing data in other domains may be informative to understand the bug distribution in the current domain.*

Besides, to make our conclusion more comprehensive, in addition to the above three domains, we also consider another more fine-grained application view. We divided the projects into 7 domains, i.e. shell, test editor, image editor, compatibility layer, version control system, compile tool and library. The preliminary results are shown in Fig. 8. Consistently, similar trends of change types on most domains can be observed, such as CFC is the most, and CAS and CBS are the second most or the third most common changes.

Overall, based on the findings above, we are able to answer RQ2:

The frequencies of change subtypes do not vary substantially. Instead, changes on function call statements are consistently the most commonly observed changes in most domains, and the average frequencies of change subtypes are significantly similar across studied domains.

4.4. RQ3: across versions

In order to answer RQ3, we study two groups of versions, one from Apr and the other from Libav. We examine whether different versions have significant similarity in frequencies of change types. The key observations are as follows.

For Apr, Fig. 9 presents the comparison results of the frequencies of change subtypes. For each version, CFC and CAS account for the largest and the second largest percentages respectively. There is a subtle variance in the third most frequent subtype, i.e. Apr1.2.1 involves more CDDI, while the other two versions involve more CBS. However, the percentages of CDDI and CBS in Apr1.2.1 do not differ too much (26.5% vs 23.5%). Similar trends can be observed in the other subtypes. To further examine the statistical similarity, Table 9 shows the Pearson's correlation between the frequencies of change subtypes in different versions. As can be seen from Table 9, the correlation values are all above 0.96 with p-value < 0.001, which indicates that the three versions of Apr have significantly similar distribution of fixing code changes. What is more, we further use Spearman's correlation to confirm the similarity, and the results are consistent with Table 9.

Table 9
Pearson's correlation between the frequencies of change subtypes in different versions of Apr.

	Apr1.2.1	Apr1.3.0	Apr1.4.2
Apr1.2.1	–	0.963	0.9770
Apr1.3.0		–	0.9780
Apr1.4.2			–

Table 10
Pearson's correlation between the frequencies of change subtypes in different versions of Libav.

	Libav0.7	Libav0.8	Libav9	Libav10	Libav11
Libav0.7	–	0.987	0.988	0.985	0.983
Libav0.8		–	0.985	0.959	0.968
Libav9			–	0.955	0.994
Libav10				–	0.957
Libav11					–

For Libav, Fig. 10 and Table 10 respectively present the frequency comparison results and the Pearson's correlation between the frequencies of change subtypes. Fig. 10 reveals that the distributions of change subtypes do not change too much across different versions. For example, the percentage of CBS only varies from 60.3% to 68.9%. Except Libav0.8, the most frequent one is CBS, and the second most frequent is CFC. While Libav0.8 just has the opposite order on CBS and CFC. Another variance among the 5 versions is that the third most frequent subtype of Libav0.7 and Libav10 is CAS, while the third in other versions is CRGS. In spite of the subtle variance, Table 10 shows that all the studied versions of Libav have similar trends on frequencies of change types, as the correlation values are consistently over 0.95 with the p-values all less than 0.001. Similarly, we also employ Spearman's correlation to further verify this, and the results consistently reveal the similarity of change type frequencies across studied versions.

For each version, the changes in the subsequent releases are mainly performed for three purposes, i.e. function enhancement, bug fixing, or refactoring [21,22]. If a version is released purely for bug fixing, only a small portion of functions are modified. For example, there are totally 1946 functions in Bash3.2, while only 62 of them are faulty. Therefore, there must be high similarity in source code between bug version and fix version. Besides, they are developed in a similar environment with similar coding styles, and hence tend to have similar frequencies of bug types. This may lead to the similar trends of their change types.

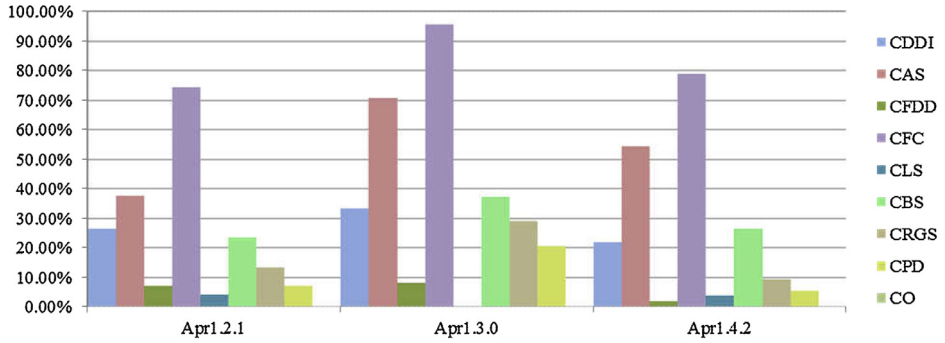

Fig. 9. The comparison between different versions of Apr.

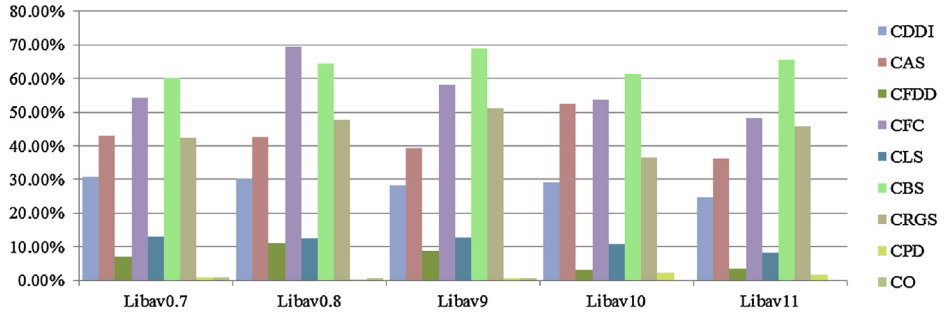

Fig. 10. The comparison between different versions of Libav.

Table 11
The kinds of changes most likely to occur together in terms of Apr.

System	#Faulty func.	Most frequent pair	Number of occurrence	Percentage	Second most frequent pair	Number of occurrence	Percentage
Apr1.2.1	98	CAS & CFC	24	24.5%	CFC & CDDI	17	17.3%
Apr1.3.0	24	CAS & CFC	17	70.8%	CFC & CBS	9	37.5%
Apr1.4.2	105	CAS & CFC	52	49.5%	CFC & CBS	20	19.0%

Table 12
The kinds of changes most likely to occur together in terms of Libav.

System	#Faulty func	Most frequent pair	Number of occurrence	Percentage	Second most frequent pair	Number of occurrence	Percentage
Libav0.7	401	CBS & CRGS	154	3.4%	CFC & CBC	135	33.7%
Libav0.8	693	CFC & CBC	320	46.2%	CBS & CRGS	288	41.6%
Libav9	486	CBS & CRGS	217	44.7%	CFC & CBC	210	43.2%
Libav10	93	CBS & CRGS	33	35.5%	CFC & CBC	31	33.3%
Libav11	122	CBS & CRGS	48	39.3%	CFC & CBC	34	27.9%

Finding 7*: Considering systems Apr and Libav, the frequencies of each subtype share similar trends across different versions. Also, the top most frequent ones are always the same.*

Implication 7*: The bug-fixing behaviors in previous versions are often valuable reference to fix the bugs in the current version.*

Besides, Tables 11 and 12 respectively reveal the specific kinds of changes that are more likely to occur together in Apr and Libav. The third and sixth columns respective show the most frequent and the second most frequent change type pairs, followed by the concurrence and percentage in all faulty functions. It can be easily observed from Table 11 that the pair of CAS and CFC is the most frequent change pair across Apr versions, ranging from 24.5% to 70.8%. The second pair is CFC and CBS in Apr1.3.0 and Apr1.4.2. The observations are generally consistent with Finding 4, except the case that the second most frequent pair in Apr1.2.1 is the pair of CFC and CDDI. While in terms of Libav, interestingly, CBS & CRGS and CFC & CBC are two most frequent pairs with similar percentages across all the studied versions of Libav. It indicates that branch statements in Libav are likely to be changed together with return/goto statements or function call statements. From the results of Tables 11 and 12, we can see that, except subtle differences, there are many common characteristics across versions of Apr and Libav.

Finding 8*: For Apr and Libav, there are similar trends on subtype pairs across different versions. Especially, the most frequent change subtype pairs always tend to be the same.*

Implication 8*: The information about code changes that always occurred together in previous versions could be valuable reference to understand characteristics of the current version.*

Overall, based on the observations above, we are able to answer RQ3:

Change subtypes occur with similar frequencies in different versions of the same system. Although there exist some obvious differences between Apr and Libav, different versions of the same system have many similar characteristics.

4.5. Observations summary

In summary, to understand change types in bug fixing code, we study 17 versions of 11 C systems varying in software size and

Table 13
The key findings and implications.

Finding 1: For most studied systems, interface related code changes are the most frequent bug-fixing changes, accounting for 74.6% on average, and specifically contributes percentages ranging from 45% to 87% across all the studied systems.	**Implication 1**: Interface-related bugs are commonly observed. Practitioners should pay more attention to interface issues when developing software or fixing bugs.
Finding 2: The frequencies of change types across most studied systems are significantly similar.	**Implication 2**: The distribution of different kinds of bugs tends to be similar across projects. The existing bug fixing information hence can be used for reference when reviewing a newly developed system.
Finding 3: A significant number of faulty functions (65.2% on average) in studied projects can be fixed by only one or two kinds of changes.	**Implication 3:** It is reasonable to assume that most bug fixing behaviors are relatively uncomplicated. When dealing with a new bug, it may be more efficient to first examine whether one or two kinds of changes can fix the bug.
Finding 4: Function call statements are very likely to be changed together with assignment statements or branch statements.	**Implication 4**: When changing assignment statements or branch statements to fix bug, developers should pay attention to the function-call statements which may also be involved.
Finding 5: Changes on function call statements are the most commonly observed changes, and changes on assignment and branch statements are often either the second or the third most frequent changes across the studied domains.	**Implication 5**: Function call statements should be highly concerned regardless the application domain when developing software or fixing bugs.
Finding 6: The frequencies of change subtypes are not domain-specific, and exhibit significantly similar trends across studied domains.	**Implication 6**: The distribution of bugs may be similar across domains. The bug fixing data in other domains may be informative to understand the bug distribution in the current domain.
Finding 7: Considering systems Apr and Libav, the frequency of each subtype shares similar trends across different versions. Also, the top most frequent ones are always the same.	**Implication 7**: The bug-fixing behaviors in previous versions are often valuable reference to fix the bugs in the current version.
Finding 8: For Apr and Libav, there are similar trends on subtype pairs across different versions. Especially, the most frequent change pairs always tend to be the same.	**Implication 8**: The information about code changes that always occurred together in previous versions could be valuable reference to understand characteristics of the current version.

application domain, and thoroughly analyze thousands of faulty functions. Our key findings and implications are summarized in Table 13. More specifically, (1) Finding 1, 5 and 7 consistently reveal that interface related code changes are most commonly observed in bug fixing activities, which suggests that interface-related bugs should be highly concerned when developing new projects and the interface-related source code should have higher priority for review when fixing bugs. (2) Finding 2, 6 and 7 show the frequency similarity of change types across studied projects, domains and versions, which indicates that the bug fixing behaviors of previous projects despite their domains or versions, have common features. Developers could learn from the previous bug fixing activities and avoid making similar mistakes again, and the knowledge on the similarity should be a valuable reference to prioritize source code to be reviewed, debugged, or tested. (3) Finding 3 shows most faulty functions could be repaired by one or two kinds of code changes, which indicates that, instead of complicating the issue as many developers usually do, simplification may be more efficient to solve the problem. Besides, (4) Finding 4 and 8 reveal that what kinds of changes are most likely to occur together, which indicates the potential correlation between change kinds and guide developers to check the other possible influence when changing one kind of code. Overall, these findings provide useful guidelines for software quality enhancement, like software development, code review, testing or debugging management, and resource allocation.

We believe our findings reveal the characteristics of changes in bug fixing code in many software systems; however we cannot draw absolutely general conclusions on all software. This is an inherent problem for all studies in empirical software engineering, not unique to us, as many factors cannot be considered completely. Consequently, our results should be taken with the change type taxonomy, system types and their programming language in mind.

5. Threats to validity

In this section, we discuss the most important threats to the construct, internal, and external validity of our study. Construct validity is the extent to which the dependent and independent variables accurately measure the concept they purport to measure. Internal validity is the extent to which the conclusions can be drawn about the causal effect of independent variables on the dependent variables. External validity is the extent to which the conclusions can be generalized to the population under study and other research settings.

The most important threat to construct validity is the limitations in our change taxonomy. We propose a classification scheme for fine-grained source code changes in bug fixes, which highlights the nature of changes, and hence visualizes the characteristics of bugs. We believe the findings based on our taxonomy could provide a better understanding of how bugs were fixed. However, the taxonomy is only based on the type of statements being modified, while ignoring the dynamic and contextual information. We haven't provided direct means of identifying the underlying reason behind the code change. For example, one "logic/control" change from "if $(n > 1)$ {...}" to "if $(n \geq 1)$ {...}", could be caused by a simple oversight or the revision of a complicated algorithm, which can't be recognized yet by our taxonomy. To bridge this gap, in future work, the taxonomy will be further extended for semantic contexts and comments analysis to identify the actual reason behind the bugs. Another potential threat is related to fix versions. As pointed out above, through mining the history database, we identify the corresponding patch data or latest fix version for each bug version. However, it is possible that there still exist some bugs not yet exposed and hence not fixed. If so, the faulty functions we locate might just be a subset of all the real faulty functions in the bug version. However, we believe it will not impact our findings too much. The reason is that the total number of faulty functions in our study is not trivial (nearly four thousand). Even if they are not the whole faulty functions, they can be regarded as a representative sample of the trends of bug-fixing behaviors. This threat can be mitigated by using more complete patch data or fix versions to find more faulty functions in the future work. Besides, if there exist other changes not related to bugs, such as changes for adding features or refactoring, it will be a possible threat to the construct validity of our study. However, we believe such changes are very rare in our studied code changes. As described in Section 3.2, to recognize the changes that fix a bug, we first search history database, identify the purpose of each release, and then find bug fixing releases or the fixing patch data, which are released purely for fixing bugs. Therefore, we suppose that the changes between the bug version and fix version are used only for bug fixing.

There are several important threats to internal validity. The first one is the exclusion of few faulty functions when detecting change patterns. The reason for the exclusion is that they could

not be parsed successively by Coccinelle. As we employ Coccinelle to identify change types, one essential condition which must be satisfied is that the C code should be syntactically correct. To perform our experiments, we ignore those functions which Coccinelle fails to parse. Even so, we believe it will not have substantial influence on our findings, as only a quite small proportion of functions (less than 4%) are excluded as shown in Section 4.1. Another threat is the effectiveness of the used tool. We develop an automatic tool CTforC based on Coccinelle to categorize code changes. Although Coccinelle has been used in a lot of previous works [14–17,23], it may still be not absolutely perfect. However, these deficiencies will not have a great impact on our conclusion. As shown in Section 4.1, we have manually checked the classification results, and our tool can achieve high accuracies (consistently above 98%). But since we evaluated CTforC manually, subjectivity is inevitable. To minimize the subjectivity, each change was examined at least three times by different checkers individually, and changes with different opinion were further discussed and re-checked to reach final consensus. Therefore, we believe the evaluation results, which suggest the effectiveness of CTforC, are believable. Of course, this threat could be further reduced by employing more advanced tools. The validity of our studied datasets with change types might have been threatened by the limitations of Coccinelle. To mitigate this threat and encourage replications, we made the datasets and CTforC tool publically available.[2]

Besides, in terms of external validity, the threat includes the generality of our findings. In this study, we studied 11 widely used systems involving 17 versions, with a total of nearly four thousand faulty functions. The projects we studied are representative and the scale of our experiments is considerable. Although we believe our results reveal well the characteristics of bug fixing code changes in many software systems, we do not intend to draw general conclusions on all software systems, as all of the studied projects are open-source software and developed in C language. Consequently, our findings may not be generalized to commercial systems or software written in other programming languages. More comprehensive and more rigorous experiments are certainly needed to enhance the understanding of change types. To mitigate this threat, there is a need to replicate our study using a wide variety of systems in the future work.

6. Related work

Fault classification: There exist a number of studies on fault classification [25–35]. One commonly used taxonomy is orthogonal defect classification(ODC) proposed by Chillerage et al., which classifies defects into eight types based upon the description about the symptoms, semantics and root causes of the defects [45]. Their classification scheme is cause-driven (what caused this bug) and not directly associated with source code. For example, a defect caused by a communication problem between modules should be an “interface” defect in terms of ODC, possibly requiring a series of changes to fix it. The modifications may involve multiple kinds of source code, which is not considered in ODC. While in this study we attempt to study defects from a different perspective, i.e. the nature of changes in bug fixing code, which is closer to the program text itself. In this example, the modifications on different kinds of code will be distinguished in terms of our taxonomy. What is more, for many systems, the description information of defects needed for ODC is often missing. While our taxonomy is based on source code, such missing will not affect the classification. In addition to ODC, there are other studies on fault classification [25–35]. Such as, Offutt and Hayes proposed a semantic model for fault categorization based on the syntactic and semantic size of a fault to analyze the characteristics of program faults [35]. Xia et al. proposed a text mining solution to categorize defects into two fault trigger categories: Bohrbug and Mandelbug, according to its natural-language description available in the corresponding bug report [5]. Vetro et al. classified 78 defects using the ISO/IEC 9126 quality main characteristics and sub-characteristics, and a set of proposed extended guidelines [27]. Tan et al. designed bug taxonomies in three dimensions, i.e. Root Cause (the fault), Impact (the failure caused by the bug), and Component (the location of the bug), and manually examined bug reports to classify real world bugs [31]. Overall, the taxonomies in above works tend to be cause-driven, impact-driven, or document-driven, while the taxonomy proposed in our study is source code driven which is more concrete and intuitive to understand the characters of bugs.

In our taxonomy, there are totally five main *change types* (i.e. Data, Computation, Interface, Logic/control, and others) and nine subtypes (i.e. CDDI, CAS, CFDD, CFC, CLS, CBS, CRGS, CPD, and CO). Some of them have been more or less revealed in previous literatures. For example, IEEE Standard Classification for Software Anomalies [46] involved *Data, Interface*, and *Logic* types, and some of the error examples are also related to our subtypes. Such as the data error like *Incorrect data type or column size* can be fixed by CDDI, the Interface errors like *Incorrect or insufficient parameters passed* can be fixed by CFC, and the Logic error like *Missing else clause* can be fixed by CBS. Michael [7] proposed a Code-fault Taxonomy, which inspired the proposition of our taxonomy. It includes Data, Computation, Interface, Logic/control and four more types. More specifically, the subtypes such as data definition, incorrect equation, incorrect loop attributes, and incorrect input parameters are also responding to our CDDI, CAS,CLS,and CFC respectively.

Automatic classification tool: There are a few studies on automatic classification of defects. Thung et al. proposed an automatic approach to classify defects into three supercategories of ODC (Orthogonal Defect Classification) defect types [2]. Their approach was able to label defects with an average accuracy of 77.8%. Huang et al. presented a tool AutoODC for automating Orthogonal Defect Classification by casting it as a supervised text classification problem [36]. Their results indicated that AutoODC was accurate in classifying defects with an accuracy of 80.2%. Chawla and Singh proposed an approach based on fuzzy set theory for automatic categorization of bug reports [37]. They conducted experiments on issue repository of three open source software systems, and achieved an accuracy of 87%, 83.5%, and 90.8% respectively. Nagwani et al. presented a software bug classification algorithm using bug attribute similarity to form clusters and then the cluster labels are matched with bug taxonomic terms [38]. Their method maintained more than 80% accuracy. In our study, we also develop an automatic tool CFforC for change classification, which is highly effective with accuracies consistently above 98%.

Source code driven classification: Some previous works also attempted to study faults based on source code. Duraes classified 668 faults into several categories based on ODC types to perform fault injection in a simulation model [39]. Their work assumes that all the changes are bug fixes, but it is not always true as some code may be modified for adding new feature or refactoring. In contrary, we mine the history database and use the real fixing code changes for experiments. Fluri and Gall [40] presented a taxonomy of source code changes according to tree edit operations in the abstract syntax tree, and implemented an Eclipse plugin CHANGEDISTILLER based on the algorithm proposed by Chawathe et al. [41]. Their types are fine-grained with more than thirty types. Among them, four types only identify the insert, update, delete, or reordering of a statement, regardless of the statement type. Kidwell et al. extend these four change types in the change taxonomy developed by Fluri et al., and evaluated the extended change types

[2] https://github.com/njuzhyy/datasets_ChangeTypes

using clustering [42]. They performed experiments on only two Java systems, which may be insufficient to show the general characteristics of change types.

Closest works: One of the closest works to ours is the work by Pan and Kim [10]. The similarities are that (1) the primary motivations are to study the source code differences between bug versions and fix versions; (2) the classification processes can be automated; and (3) some experimental results are somewhat similar, such as they found that the bug fix pattern frequencies tended to be similar across studied projects (consistent with our Finding 2), and the most common categories of bug fix patterns were Method Call (similar to our Finding 1). However, despite the above similarities, the main differences are that: (1) taxonomy difference: they defined 27 automatically bug fix patterns, while we define 5 change types with 9 subtypes. Their taxonomy is finer grained than ours. For example, if-related changes are further divided into 7 subtypes in their study. Since their taxonomy is fine grained, only a small portion of changes can be categorized into each subtype, giving the most common patterns in their study being MC-DAP at 14.9–25.5%, IF-CC at 5.6–18.6%, and AS-CE at 6.0–14.2%, which may not be as valuable as high-level change types for the purpose of code reviewing; (2) coverage difference: only 45.7%–63.6% of the bug fix changes were covered by their bug fix patterns, while our change types could cover all kinds of changes in source code; (3) language difference: their taxonomy is Java specific, while our defined change types are applicable to almost all programming languages; (4) experimental difference: they performed experiments based on 7 Java systems, while we investigate code changes using 17 versions of 11 C systems. Besides, we perform a more comprehensive investigation on the characteristics of change types across projects, domains and versions. Our findings should be more informative for both developers and researchers.

Another close work is the work by Hayes et al. [8]. They developed two taxonomies, one for code modules and another for code faults, and proposed an approach for examining relationships between the two. We learn from their fault taxonomy, and propose our own taxonomy for changes in bug fixing code. Furthermore, we provide concrete guidelines on classification. The important difference is that their taxonomy relies on bug reports or problem reports, while ours only relies on changed source code. Besides, they did not implement complete automation, while we develop an automatic classification tool for classification. They applied the taxonomy to two C/C++ open source software systems, and concluded that Control/Logic faults occurred most frequently for both systems, while we study more projects to investigate the general characteristics of change types.

7. Conclusions and future work

In this paper, we perform a comprehensive empirical investigation on general characteristics of change types. We first present a new taxonomy for categorizing code changes: Data, Computation, Interface, Logic/control, and Others, which are further subdivided into 9 types (CDDI, CAS, CFDD, CFC, CLS, CBS, CRGS, CPD, and CO). To support our experiment, we then develop an automatic classification tool CTforC based on Coccinelle. We manually evaluate the effectiveness of our tool, and the results indicate CTforC is reliable.

To gain deep insights into changes in bug fixing code, we organize our experiments based on 3 research questions. Based on 17 versions of 11 C systems with thousands of faulty functions, we observe the following key findings: (1) Across studied projects: The frequencies of change subtypes are significantly similar across most studied projects; on average, interface related code changes are often the most frequent bug-fixing changes; most of faulty functions can be fixed by one or two change subtypes; function call statements are very likely to be changed together with assignment statements or branch statements; (2) Across studied domains: The frequencies of change subtypes are similar. Changes on function call, assignment and branch statements are often the three most frequent changes; and (3) Across studied versions: The frequencies of change subtypes share similar trends, and the most common subtype pairs tend to be same. We believe our results are valuable to guide code review, help manage software testing and debugging, and improve the software development.

In the future work, we plan to extend our study in the following three directions: (1) identifying the underlying reason behind the code change. We will extend the taxonomy by combining the semantic contexts and the comments in the code to identify the actual reason behind the bugs, which will provide more insightful understanding of bugs; (2) replicating this study based on more systems. The purpose is to examine whether the findings from this study can be generalized to other kinds of systems, such as commercial systems or software written in other programming languages; and (3) predicting bug-fixing change types. Based on structural features and change types, we can build prediction models. Different from traditional fault-prediction models, change type prediction model can be used to predict the bug-fixing behaviors for a given module, and hence directly inform the process of code review and bug fixing.

Acknowledgments

The work in this paper is supported by the National Natural Science Foundation of China (61272082, 61432001, 91418202), the National Key Basic Research and Development Program of China (2014CB340702), and the National Natural Science Foundation of Jiangsu Province (BK20130014).

References

[1] F.A. Arshad, R.J. Krause, S. Bagchi, Characterizing configuration problems in java EE application servers: an empirical study with glassfish and JBoss, in: Proceedings of the 24th International Symposium on Software Reliability Engineering, 2013, pp. 198–207.
[2] F. Thung, D. Lo, L. Jiang, Automatic defect categorization, in: Proceedings of the19th Working Conference on Reverse Engineering, 2012, pp. 205–214.
[3] X. Xia, D. Lo, W. Qiu, X. Wang, B. Zhou, Automated configuration bug report prediction using text mining, in: Proceedings of the 38th Computers, Software & Applications Conference, 2014, pp. 107–116.
[4] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, K.S. Trivedi, Fault triggers in open-source software: an experience report, in: Proceedings of the 24th International Symposium on Software Reliability Engineering, 2013, pp. 178–187.
[5] X. Xia, D. Lo, X. Wang, B. Zhou, Automatic defect categorization based on fault triggering conditions, in: Proceedings of the 19th International Conference on Engineering of Complex Computer Systems, 2014, pp. 39–48.
[6] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secure Comput. 1 (1) (2004) 11–33.
[7] I.R.C. Michael, Fault Links: Identifying Module and Fault Types and Their Relationship, University of Kentucky Master's Theses, 2004.
[8] J.H. Hayes, C.M.I. Raphael, V.K. Surisetty, A. Andrews, Fault links: exploring the relationship between module and fault types, in: Dependable Computing-EDCC 5, Springer, Berlin Heidelberg, 2005, pp. 415–434.
[9] J.H. Hayes, I.R. Chemannoor, E.A. Holbrook, Improved code defect detection with fault links, Softw. Test. Verif. Rel. 21 (4) (2011) 299–325.
[10] K. Pan, S. Kim, E.J. Whitehead Jr, Toward an understanding of bug fix patterns, Empir. Softw. Eng. 14 (3) (2009) 286–315.
[11] A. Mockus, L.G. Votta, Identifying reasons for software changes using historic databases, in: Proceedings of the International Conference on Software Maintenance, 2000, pp. 120–130.
[12] Y. Padioleau, J. Lawall, R.R. Hansen, G. Muller, Documenting and automating collateral evolutions in Linux device drivers, in: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems, 2008, pp. 247–260.
[13] J. Brunel, D. Doligez, R.R. Hansen, J. Lawall, G. Muller, A foundation for flow-based program matching using temporal logic and model checking, in: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2009, pp. 114–126.
[14] J.L. Lawall, J. Brunel, N. Palix, R.R. Hansen, H. Stuart, G. Muller, WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process, Software 43 (1) (2013) 67–92.

[15] N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, G. Muller, Faults in Linux: ten years later, in: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 305–318.
[16] J. Lawall, B. Laurie, N.Palix R.R.Hansen, G. Muller, Finding error handling bugs in OpenSSL using coccinelle, in: Proceedings of the 2010 European Dependable Computing Conference, 2010, pp. 191–196.
[17] N. Palix, J.L. Lawall, G. Thomas, How often do experts make mistakes? in: Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2010, pp. 9–15.
[18] V.R. Basili, B.T. Perricone, Software errors and complexity: an empirical investigation, Commun. ACM 27 (1) (1984) 42–52.
[19] B. Marick, A Survey of Software Fault Surveys, University of Illinois at Urbana-Champaign, Department of Computer Science, 1990.
[20] K. Pearson, Contributions to the mathematical theory of evolution, Philos. Trans. R. Soc. London. A 185 (1894) 71–110.
[21] E.B. Swanson, The dimensions of maintenance, in: Proceedings of the 2nd International Conference on Software Engineering, 1976, pp. 492–497.
[22] G. Canfora, L. Cerulo, M M. Cimitile, M. Di Penta, How changes affect software entropy: an empirical study, Empir. Softw. Eng. 19 (1) (2014) 1–38.
[23] N. Palix, J.R. Falleri, J. Lawall, Improving pattern tracking with a language-aware tree differencing algorithm, in: Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering, 2015, pp. 43–52.
[24] X. Zhu, Q. Song, Z. Sun, An empirical analysis of software changes on statement entity in java open source projects, Int. J. Open Source Softw. Processes 4 (2) (2012) 16–31.
[25] Y. Zhou, Y. Tong, R. Gu, H. Gall, Combining text mining and data mining for bug report classification, in: Proceedings of the 22nd IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 311–320.
[26] D. Falessi, B. Kidwell, J. Huffman Hayes, F. Shull, On failure classification: the impact of getting it wrong, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 512–515.
[27] A. Vetro, N. Zazworka, C. Seaman, F. Shull, Using the ISO/IEC 9126 product quality model to classify defects: a controlled experiment, in: Proceedings of the 16th International Conference on Evaluation & Assessment in Software Engineering, 2012, pp. 187–196.
[28] D. Falessi, G. Cantone, Exploring feasibility of software defects orthogonal classification, in: Software and Data Technologies, Springer, Berlin Heidelberg, 2008, pp. 136–152.
[29] K. Henningsson, C. Wohlin, Assuring fault classification agreement - an empirical evaluation, in: Proceedings of the International Symposium on Empirical Software Engineering, 2004, pp. 95–104.
[30] R.A. Demillo, A.P. Mathur, A grammar based fault classification scheme and its application to the classification of the errors of TEX, Purdue University, 1995 Technical Report SERC-TR-165-P.
[31] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, Bug characteristics in open source software, Empir. Softw. Eng. 19 (6) (2014) 1665–1705.
[32] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, C. Zhai, Have things changed now? An empirical study of bug characteristics in modern open source software, in: Proceedings of Workshop on Architectural and System Support for Improving Software Dependability, 2006, pp. 25–33.
[33] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, ACM SIGPLAN Notices 43 (3) (2008) 329–339.
[34] M. Felderer, A. Beer, B. Peischl, On the role of defect taxonomy types for testing requirements: results of a controlled experiment, in: Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications, 2014, pp. 377–384.
[35] A.J. Offutt, J.H. Hayes, A semantic model of program faults, ACM SIGSOFT Softw. Eng. Notes 21 (3) (1996) 195–200.
[36] L.G. Huang, V. Ng, I. Persing, R. Geng, X. Bai, J. Tian, AutoODC: automated generation of orthogonal defect classifications, in: Proceedings of the International Conference on Automated Software Engineering, 2011, pp. 412–415.
[37] I. Chawla, S.K. Singh, An Automated approach for Bug Categorization using Fuzzy Logic, in: Proceedings of the 8th India Software Engineering Conference, 2015, pp. 90–99.
[38] N.K. Nagwani, S. Verma, Clubas: An algorithm and Java based tool for software bug classification using bug attributes similarities, J. Softw. Eng. Appl. 5 (6) (2012) 436–447.
[39] J.A. Duraes, H.S. Madeira, Emulation of software faults: a field data study and a practical approach, IEEE Trans. Softw. Eng. 32 (11) (2006) 849–867.
[40] B. Fluri, H.C. Gall, Classifying change types for qualifying change couplings, in: Proceedings of the International Conference on Program Comprehension, 2006, pp. 35–45.
[41] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom, Change detection in hierarchically structured information, ACM SIGMOD Record 25 (2) (1996) 493–504.
[42] B. Kidwell, J.H. Hayes, A.P. Nikora, Toward extended change types for analyzing software faults, in: Proceedings of the International Conference on Quality Software, 2014, pp. 202–211.
[43] X. Zhu, E.J. Whitehead, C. Sadowski, Q. Song, An analysis of programming language statement frequency in C, C++, and Java source code, Software: Practice and Experience 45 (11) (2015) 1479–1495.
[44] H. Cohen, C. Lefebvre, Handbook of Categorization in Cognitive Science, Elsevier, 2015.
[45] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, M.Y. Wong, Orthogonal defect classification-a concept for in-process measurements, IEEE Trans. Softw. Eng. 18 (11) (1992) 943–956.
[46] ISDW Group, 1044-2009-IEEE Standard Classification for Software Anomalies (2010).