JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
*J. Softw. Evol. and Proc.* 0000; **00**:1–37
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

# Comparing text-based and dependence-based approaches for determining the origins of bugs

Steven Davies, Marc Roper & Murray Wood
Computer and Information Sciences
University of Strathclyde
Glasgow, UK
{Steven.Davies,Marc.Roper,Murray.Wood}@cis.strath.ac.uk

SUMMARY

Identifying bug origins — the point where erroneous code was introduced — is crucial for many software engineering activities, from identifying process weaknesses and improvements to gathering data to support tool development for bug detection. Unfortunately this information is not usually recorded when fixes are made, and recovering it later is challenging. Recently the text approach and the dependence approach have been developed to tackle this problem. Respectively, they examine textual and dependence-related changes that occurred prior to a bug fix. However, only limited evaluation has been carried out, partially due to the lack of data-sets linking bugs to origins, and a lack of available implementations. To address this, origins of 174 bugs in three open-source projects were manually identified and compared to a simulation of the approaches. Both approaches were partially successful across a variety of different bugs — achieving precision of 29%–79% and recall of 40%–70%. Results suggested the precise definition of program dependence could affect performance, as could the decision on whether to identify a single or multiple origins. A number of potential improvements are explored in detail and serve to identify pragmatic strategies for combining the techniques along with relatively simple modifications. Even after adopting these improvements there remain many challenges: large commits, unrelated changes and long periods between origins and fixes all reduce effectiveness.


Received ...

KEY WORDS: software maintenance, bug origins, mining software repositories, program dependence graph, version control, bug tracking systems

## 1. INTRODUCTION

Given the suggestion that software developers can spend nearly half their time fixing bugs [1], and the well-documented impact of bugs on software development cost and effort, it is not surprising that many techniques and tools have been developed which attempt to prevent bugs from being introduced in the first place. Often this is done by inferring common patterns from code which contains bugs and then finding similar code. However, accurately identifying bug *origins* — the point in the codebase that a bug was introduced — is particularly challenging due to factors such as the length of time between introducing and reporting a bug, and the potentially substantial changes that may have been made to the code in the meantime. Being able to automatically identify which changes actually introduced a bug could be useful in improving the accuracy of any of these bug prevention tools or enabling the introduction of more advanced techniques altogether.

Software developers and managers would also stand to benefit from identifying when bugs were introduced, and use such data to pinpoint potential weaknesses in their processes. Furthermore, identifying such changes also opens possibilities for researchers to more closely examine the nature of changes that can introduce bugs.

Recently, two approaches have been proposed that attempt to identify the origin of a bug based on the changes made to fix it. Both start from the version of code containing the bug fix and progressively examine preceding versions until they find the one that introduced the code responsible for the bug. They differ primarily in how they examine the changes between versions and the number of files returned: the text approach [2] uses only the change in the text itself and returns all files deemed to be relevant, while the dependence approach [3] uses changes in the relationships between control and data in the code and returns only the first relevant file. Unfortunately, as yet no implementations of either approach are readily available, which presents a significant challenge to assessing their effectiveness or even their viability. Furthermore, the effort required to create an implementation in both cases is non-trivial, and the dependence approach in particular poses a number of implementation challenges, as discussed in Section 2.4.

As a necessary precursor prior to embarking on an expensive implementation exercise, this paper presents the results of an investigation into the accuracy of both approaches and provides insights into their respective strengths and weaknesses. Three open-source systems were chosen, the origins of a number of bugs associated with each system were identified, and the application of both the text approach and the dependence approach was then manually simulated. Comparing the results of this manual simulation with the true bug origins revealed a large variation in both the precision and recall of the approaches and also identified some key issues relating to factors such as unrelated changes, the number of versions between the bug introducing change and the fix, multiple files, and the potential variability introduced by dependence graph implementations. In addition to these challenges a range of possible improvements to both techniques are identified and a number of these explored in detail on a small subset of bugs. The results of this analysis show that, with some simple modifications, it is possible to increase the effectiveness of both approaches substantially at a relatively small cost and also combine the approaches in various ways that trade off effectiveness and efficiency. However, many of the key issues identified above still remain.

## 2. BACKGROUND AND RELATED WORK

### *2.1. Identifying Bug Origins*

Two components of modern development are crucial to help identify the origin of a bug: software configuration management (SCM) and bug tracking systems (BTSs).

SCM allows multiple developers to work on a project by coordinating their changes. Developers *check out* a copy of the code from a central repository, make changes on their own machine, then *commit* their changes back to the repository. When committing, developers usually include a comment describing the changes they have made. Each commit results in a new *version* of the code being stored in the SCM repository.

BTSs store information about problems with the code, called *issues*. Usually the information recorded for each issue includes: an ID; a description; how to reproduce the issue; and the current status, e.g. *fixed*, *invalid*, *in progress*, etc. Additionally, the BTS often records the type of issue. Many classifications exist but two of the most common are *enhancements*, which are requests for new functionality, and *bugs*, which are areas where the software does not do what it is supposed to.

While these systems are not usually integrated, techniques exist to combine their information [2]. When fixing bugs, developers often include the bug ID in their commit comment [2, 4]. Bugs can therefore be linked to the commits that fix them by extracting the



*Prepared using* smrauth.cls
DOI: 10.1002/smr

```
int shift = isCarbon ? -25 : -10;
light = getColor(BACKGROUND);
dark = new Color(disp,
   max(0, light.red() + shift),
   max(0, light.green() + shift),
   max(0, light.blue() + shift));
textCol = getColor(FOREGROUND);
```

(a) v1.39

```
1.17 int shift = isCarbon ? -25 : -10;
1.7  light = getColor(BACKGROUND);
1.8  dark = new Color(disp,
1.8     light.red() + shift,
1.8     light.green() + shift,
1.8     light.blue() + shift);
1.24 textCol = getColor(FOREGROUND);
```

(b) v1.38 (Version numbers on left)

```
int shift = carbon ? -25 : -10;
light = getColor(BACKGROUND);
dark = new Color(disp,
   light.red() + shift,
   light.green() + shift,
   light.blue() + shift);
taskCol = getColor(FOREGROUND);
```

(c) v1.8

```
light = getColor(BACKGROUND);
if (carbon)
  dark = new Color(disp,230,230,230);
else
  dark = new Color(disp,245,245,245);
taskCol = new Color(disp,120,120,120);
```

(d) v1.7

Figure 1. Eclipse Bug 63216 — `NewProgressViewer.java`. Names and formatting altered for clarity.

comments from the SCM repository and searching for bug IDs. Whilst other techniques for this task have been proposed [5], they have not been applied in this work.

Unfortunately for this work, the relationship between bugs and commits is not a simple one:

- Bug fixes may be relatively simple and may involve just a line or two in the system and the corresponding commit may involve just one file.
- Alternatively, bug fixes may involve a wider set of far more substantial changes. Consequently, a commit may be made up of one or more changes to one or more files.
- A commit may also involve changes that are unrelated to the bug in question: they may be associated with a different bug, or with no bug at all (taking the form of opportunistic refactorings or enhancements for example).
- A bug report may contain details about more than one bug.
- Developers may make more than one attempt to fix a bug, in which case there will be multiple commits associated with one bug report.
- A commit may fix a bug, but make no mention of the bug ID at all.

```
  void applyResult(DResult result){
1:  Descriptor [] d =
      result.getDescriptors();

3:  for(int i=0; i<desc.length; i++){
4:    if(d[i] != null)
5:      desc[i] = d[i];
    }

  }
```

(a) v1.6

```
  void applyResult(DResult result){
1:  Descriptor [] d =
      result.getDescriptors();
2:  if(d != null){
3:    for(int i=0; i<desc.length; i++){
4:      if(d[i] != null)
5:        desc[i] = d[i];
      }
    }
  }
```

(b) v1.7

Figure 2. Eclipse Bug 66653 — `DecorationBuilder.java`. Names and formatting altered for clarity.



DOI: 10.1002/smr

Given this complex relationship, the assumption for this work — that commits and bugs can be automatically linked via IDs — may be regarded as a rather bold one, but it is also one that provides the greatest opportunity for automatically processing large numbers of bugs and their associated fixes. Many other researchers, dating back to Fischer et al. [6], have used this assumption as the basis of their work, although there is some evidence that these links are not entirely accurate [4].

### *2.2. Text Approach*

Figure 1 shows four versions of the file `NewProgressViewer.java`. In version 1.39 the developers fixed a bug where, in certain scenarios, negative values were passed to the `Color` constructor causing an `IllegalArgumentException`. To remove the error three of the lines were updated to wrap parameters with calls to `Math.max(0, ...)` and so avoid the negative numbers. Using the text approach first proposed by Śliwerski et al. [2], to determine when this bug was introduced the *cvs diff* command is run on version 1.39. This command identifies all the lines that were added, removed or changed in that commit. The *cvs annotate* command is then run on the previous version, 1.38. This command displays the last version to change each line of code. For each line altered in the fix the version it was previously altered in is considered a possible origin of the bug. In this example each of the updated lines was last changed in version 1.8. Comparing version 1.8 to version 1.7 shows that this was indeed when the bug was introduced, as previously these values were set to specific numbers.

Various proposed improvements to the original approach are detailed in Section 2.5. Most pertinent to this study is that changes in formatting, whitespace and comments can be ignored, as they are unlikely to have been involved in causing or fixing a bug [7].

Unfortunately, the text approach is not suitable for all bugs. Figure 2 shows a bug involving a `NullPointerException` in the file `DecorationBuilder.java`. The bug was fixed by surrounding existing code with an `if`-statement that checks whether the variable is null. As these added lines did not exist in the previous version of code, *cvs annotate* cannot be used and the text approach cannot therefore identify any origin for this type of bug. A potentially more serious flaw is that there is no guarantee that a bug was actually introduced at the same location as it was fixed.

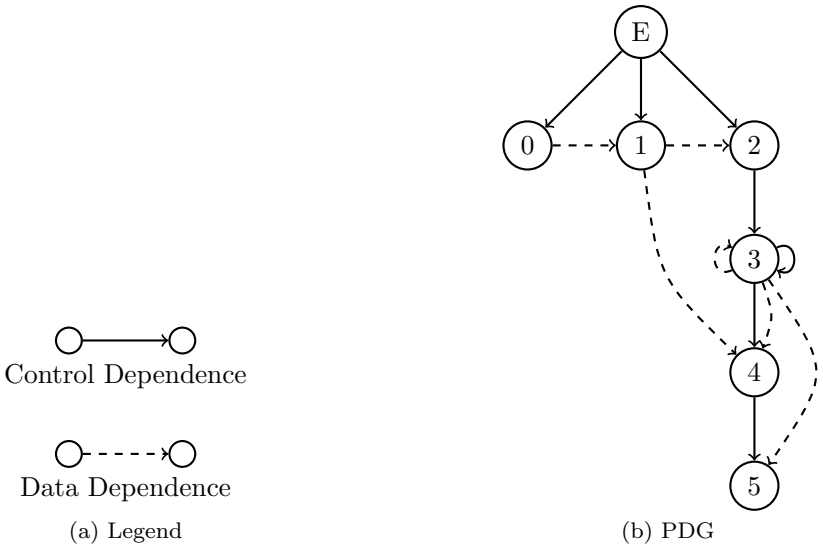
### *2.3. Dependence Approach*

The dependence approach [3] attempts to address some of the text approach's shortcomings by examining changes in the behaviour of the code rather than simply the text, using a program dependence graph (PDG). The idea of using a PDG within software engineering was first proposed by Ottenstein and Ottenstein [8]. In this simplest definition of a dependence graph, each statement is represented by a node and an additional unique *entry* node is added to identify the entry point of the program. Directed edges connect the nodes to identify the control dependences and the data dependences. A control dependence from node $X$ to node $Y$ (shown as $X \rightarrow Y$) exists if the execution of $Y$ is conditional on the outcome of the execution of the predicate at $X$. A data dependence from node $X$ to node $Y$ (shown as $X \dashrightarrow Y$) exists if $Y$ references a variable which is defined in $X$. An example of a simple method dependence graph for version 1.7 of the code in Figure 2 is shown in Figure 3. There are two points worth mentioning in relation to this illustration: firstly, an additional node (labelled *0*) has been introduced to represent the `result` parameter; and secondly the method contains references to an instance variable `desc`, but as this is external to the method there are no data dependences associated with it.

To determine the origin of the bug in Figure 2 the dependence approach compares the PDG for the fixed version to the PDG for the previous version. The approach first identifies any removed dependences. Added dependences are only examined if no dependences were removed. As shown in Figure 4, the control dependence of line 3 on method entry has been removed and the line now has a control dependence on the new `if`-statement. The




DOI: 10.1002/smr

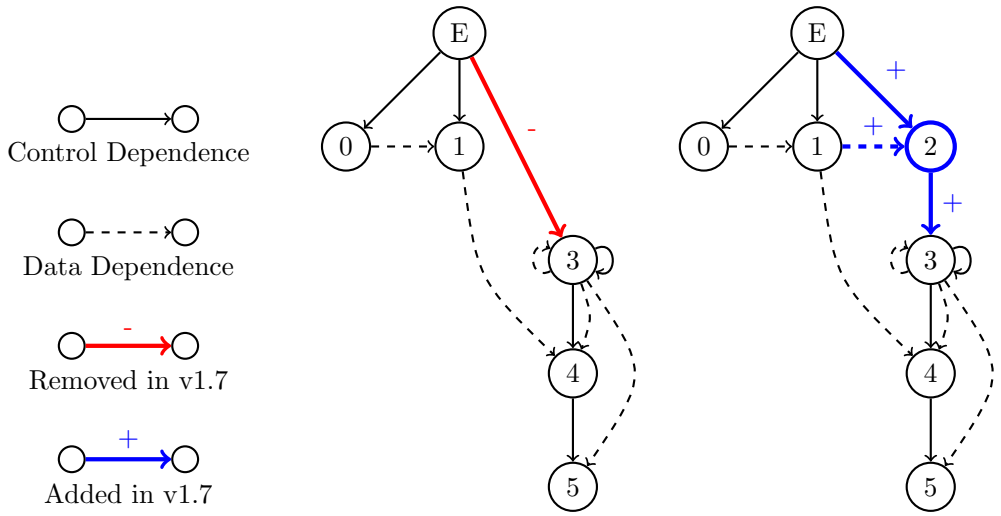



(a) Legend (b) PDG

Figure 3. PDG for `DecorationBuilder.java` v1.7 (See Figure 2)

approach therefore builds the PDG for each preceding version until it finds when the removed control dependence was added. In this case the dependence, and the bug, was introduced in version 1.5 when the method was created. If there are multiple dependences removed in the fix the dependence approach returns the most recent version in which one of these dependences was introduced.

Figure 4 also shows that a new line 2 has been introduced, with two new corresponding control dependences and a new data dependence from line 1 to line 2. As discussed, the approach prioritises removed dependences but if no dependences had been removed the approach would have built PDGs for preceding versions until it found the most recent one that altered either the source or target line of any of the new dependences.




(a) Legend (b) v1.6 (c) v1.7

Figure 4. PDGs for `DecorationBuilder.java`



DOI: 10.1002/smr

The dependence approach returns only a single version, the most recent version involved. The original rationale for this was that if multiple dependences were involved in a fix, each of which was added at a separate time, then the bug would not have actually exhibited itself until the last of these was added. This is in contrast to the text approach, where multiple versions can be returned, potentially a different one for each line in the fix. The rationale for this is that each line contributed to the bug initially. Both of these could be considered valid interpretations of the origin of the bug, but it should be noted that these assumptions are not actually inherent to the approaches in questions. Either approach could be modified to return a single or multiple versions.

This work uses the decision of Sinha et al. [3], taking the last change involved to have been the one that introduced the bug. Note however that one bug report does not necessarily equate to one bug and so for a single bug report there can still be multiple versions in which a bug was introduced.

The dependence approach is not appropriate for all bugs. Using the approach described here the bug fix in Figure 1 would not result in any change to the method's dependences. The approach could not therefore identify the origin of the bug.

### *2.4. Dependence Graphs*

Since the introduction of the PDG there have been numerous additions and refinements made. One of the first notable modifications was made by Horwitz et al. [9] who extended the representation to handle programs with multiple procedures (the original PDG was only designed to handle monolithic programs). Termed the system dependence graph (SDG), and created with the goal of inter-procedural slicing in mind, its aim was to capture both the direct and the transitive dependences that can occur as a consequence of procedure calls. This was achieved by the inclusion of additional graph elements to identify the *callsites* (the points in the program from which the procedure may be called) and the mapping between the formal and the actual input and output parameters, which results in the creation of four nodes for every parameter ($actual_{in}, formal_{in}, actual_{out}$ and $formal_{out}$). These parameters may be further linked via *summary edges*, which define dependences between parameters such as when an $actual_{in}$ node may influence the value of an $actual_{out}$ node. The result is a collection of PDGs representing each procedure, linked via these additional control and data dependences from callsites to procedures.

The advent of popular object-oriented languages introduced further features that need to be captured by the SDG, most notably inheritance, dynamic binding and polymorphism, which create further possible transitive dependences. For instance, a reference in a program may be to an interface or to a class in an inheritance hierarchy which means that any message sent via this reference at run time may be received by one of many possible objects. This binding is not known at the time the graph is created, so to capture the transitive dependences the SDG needs to be extended to include *every* possible receiving type. Early representations of graphs for C++ were created by Larsen and Harrold [10] (and updated by Liang and Harrold [11]) and adapted by Kovács et al. [12] and Zhao [13] to capture Java-specific features. Walkinshaw et al. [14] describe these developments and the key components of the SDG for Java in more detail.

Although the SDG for even a modest sized Java program is a complex structure, it is still capturing just control and data information and its key components may be summarised as follows:

- The atomic construct is the statement which is either a simple expression or a callsite if method invocations are involved.
- Statements are collected together into methods, each of which is represented by its own graph and also includes formal and actual parameter information, and return value information, in a similar vein to procedures in the traditional SDG.




DOI: 10.1002/smr

- Methods are linked to classes, which may also include data members. Any dependences between these data members and statements within methods (either in the containing class or elsewhere) are identified.
- Classes may exist within inheritance hierarchies and any data dependences between these need to be identified.
- Interfaces are modelled as separate graphs to identify the parameter and return type dependences between the interface and any potential implementation.
- Callsites identify all receiver types (although in practice this may not be feasible due to the possibly very large number of potential receivers).
- The treatment of type information is deserving of a special mention as it is often overlooked in the definitions of both the PDG and SDG. User-defined types (references to classes elsewhere in the system) will be captured via data dependences, but simple types and references to library classes will only appear as labels within the nodes which represent the variable declarations. The strategy taken by Apiwattanapong et al. [15] (upon whose work the graph differencing employed by Sinha et al. [3] is closely based) is to augment the name of scalar variables with type information (so a variable `x` of type `int` is identified as `x_int`). User-defined types and library classes are referred to by their globally qualified name (such as `java.lang.String`). This ensures that any changes to the type of identifiers will be reflected in a change to the node label.

For the purposes of this paper the focus of the analysis is on the method level which restricts the control and data dependences considered to just those *within* the method. This includes the formal parameters (but for input purposes only — any data dependences exported via a parameter are not modelled) but excludes any calls to other methods or references to variables outside the method, such as class member data. The original paper [3] was evaluated using a similar implementation, although it did describe the implementation of a full approach using SDGs.

Although tools to build the SDG of a Java system are not widely available, constructing them by combining the AST representation of a system available via the Eclipse JDT [16] along with a framework for performing dataflow analysis such as Wala [17] or Crystal [18] is now a realistic proposition (see for example the work of Luo et al. [19]). However, in this work it was decided to proceed with a manual analysis. Even though constructing a tool to perform the dependency analysis was a feasible option it would still probably have taken more time than to perform the analysis by hand. It must be stressed that the aim of this work is to gain insights into the approaches and, in particular, any problems associated with them, and without such insights it is not possible to build an accurate analyser. Should the approaches prove both valuable and free from fundamental problems then investment in the construction of an analyser would be worthwhile.

### *2.5. Additional Related Work*

Several improvements have been suggested to the approaches. Williams and Spacco [20] built on the text approach using a Java-syntax aware differencing tool, DiffJ [21]. This tool reports changes in terms of operations to Java elements, such as renaming or reordering. As well as allowing changes that have no semantic effect to be ignored, this also means that some changes, such as parameter reordering, can be disregarded as being unlikely to fix a bug. They also developed the concept of line mapping [7] further. Jung et al. [22] detailed common patterns that can identify individual changes within a fixing commit which were not involved in the fix and proposed a tool for automatically detecting such patterns. These patterns included simple syntactically detectable patterns such as the addition or removal of unnecessary brackets and semicolons, or the renaming of methods. However they also included those requiring further semantic analysis such as the replacement of constant values with variables of the same value or the addition and removal of temporary variables. The approaches also bear some resemblance to the concept of delta debugging [23], where failing test cases from a later version of code are used repeatedly on earlier versions in order




DOI: 10.1002/smr

to identify the last version where the test passed. This was extended further with iterative delta debugging [24], in which fixes for other bugs are backported to older versions to lessen the chances of the test cases failing for unrelated reasons.

Many studies have been carried out that attempt to examine and categorise bugs. The most relevant to this work have studied the actual cause of bugs [25–27], but while these have yielded many insights into the huge variety of causes and fixes of bugs, they did not seek to find the originating version of code. Chou et al. [28] examined warnings found by twelve automated bug checkers across versions of Linux, but did not examine user-reported bugs. For all checkers they found the median difference between the introduction and the fix to be around 1.25 years. However, this often varied by the type of warning: those related to potential deadlocks could last significantly longer.

A number of practical tools have been built on top of the text approach. HATARI [29] is an Eclipse plugin which determines how risky it is to change an area of source code. For each bug in a project's BTS it identifies the lines involved in the fix, using the techniques described earlier. Lines where a high proportion of changes lead to later fixes are identified as risky, and a visual indication of risk is presented to the developer. The developer can also view the history of each line, and examine which changes lead to which bug. FixCache [30] has a similar aim. A cache of elements, either files or methods, is maintained. Whenever a bug is fixed, the cause of the bug is identified and those elements are updated in the cache. The cache is then used to predict how likely changing an area is to cause another bug.

## 3. EVALUATION

As stated previously, limited evaluation has so far been performed on the approaches. The original text approach [2] did not validate whether the commits identified by the approaches actually caused the bugs in question. Subsequent work on other refinements has focused on evaluating the difference between the results returned by the original approach and those of the refined approach. In particular, there has been no evaluation of false negatives: commits which introduced bugs but which are not identified by the approaches. Identifying whether a commit caused a bug is time-consuming and subjective, as there is no one definition of *bug* and there may well be multiple causes that could be said to introduce a bug. Additionally, implementations of the approaches are not readily available, and are challenging to complete. There is a notable paucity of reliable and well-known implementations of tools to support the creation and analysis of Java PDGs or SDGs. Therefore this section will evaluate a manual *simulation* of the approaches. This allows their expected effectiveness to be assessed prior to any implementation.

### *3.1. Subjects*

The systems under examination are Eclipse [31], an IDE and development platform, and Rachota [32], a time tracking application. These were selected as they vary considerably in size, maturity and usage, although both are open-source, written in Java and use CVS [33]. Bug and commit data for Eclipse was obtained from *Promise* [34], and contains a collection of fixes, each of which is a commit linked to a bug as described in Section 2.1. The set contains every fix which could be linked to a bug and which occurred from six months before the 3.0 release of Eclipse until six months afterwards. However, bugs may have been introduced in earlier releases and all previous versions were included in the evaluation. A series of scripts was used to select a random sample of 100 bugs, out of 4136 in total. These were linked to 301 separate commits (from a total of 10402). While this sample size may be considered small, the time required for evaluation prohibited the study of a larger sample.

The data for Rachota was obtained from FLOSSMetrics [35]. By a manual examination, all issues raised before 3rd September 2008 and fixed before 15th March 2011 were classified into bugs and enhancements, as the Rachota BTS does not record this information. Then 242




DOI: 10.1002/smr

fixing commits were identified for each of the 66 bugs in Rachota. A number of these commits only modified plain text files. While the text approach can be used on these files, the dependence approach cannot. Therefore, only the 130 commits involving Java files will be considered in this evaluation.

The sampling of bugs for analysis was driven primarily by the availability of the data and it probably best categorised as purposive sampling. It is emphatically non-probabilistic, and as such cannot be considered to be a representative sample of all bugs in all software systems (which means that it would not make sense to calculate confidence intervals etc.). This issue is considered in Section 6. Considering the bugs from the two systems in detail, all the bugs from Rachota were selected, and 100 out of the 4136 linked bugs from Eclipse were chosen.

Further evaluation was carried out on a sample of jEdit [36] bugs provided by Dit et al. [37]. jEdit is a text editor for programmers maintained by the free software community, which has been used as a case study in numerous research studies. jEdit was used by the authors to develop a common understanding of the application of the text and dependence approaches and to check that a third case study found the same kind of issues that arose with Eclipse and Rachota. It is used here to provide examples of qualitative insights into the application of the two techniques and also to provide a standardised process to document bugs and the outcome of the application of techniques in future studies. jEdit also uses Bugzilla, but unlike Eclipse or Rachota uses SVN as its SCM. The approaches are applicable to any SCM, but there is one slight difference. In CVS, each file is committed individually, and a version number is maintained for each file individually. In SVN, all modified files are committed together, and a revision number is used which identifies the state of the entire project, not just a single file.

All authors applied both approaches to eight selected jEdit bugs and then worked through each in detail to identify key decisions that needed to be made to ensure consistency of application, particularly with respect to the dependence approach. Some of the issues that arose are discussed in Sections 4 and 5, with some jEdit examples being used as detailed examples throughout the paper.

### *3.2. Research Procedure — A Worked Example*

This section describes the research procedure used to simulate the text and dependence based approaches. To demonstrate both of these a detailed example of the analysis of jEdit bug 1965114 will be used.

The bug was related to the SVN revision that fixed the bug through the use of the bug ID in the commit comment, as provided by the original sample [37]. Additional scripts were written to parse the provided dataset into an HTML page with links to jEdit's Bugzilla [38] and ViewVC [39] servers to aid analysis. An example of the output from these scripts for bug 1965114 is shown in Figure 5.

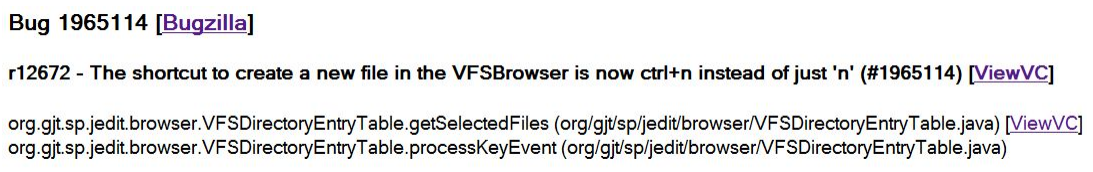



Figure 5. Bug summary relating Bugzilla description to SVN revision

The *Bugzilla* hyperlink in Figure 5 opens the bug description shown in Figure 6 and the *ViewVC* hyperlink opens the revision description shown in Figure 7. The bug is described as: "Pressing 'n' with the file system browser dockable open, and the file list focused, opens a new file." The problem is that typing an `n` character in a filename opens a new file, making it impossible to open files with the letter `n` in their filename using the keyboard. The shortcut to open a new file is supposed to be `Ctrl+n`, not just `n`.




DOI: 10.1002/smr

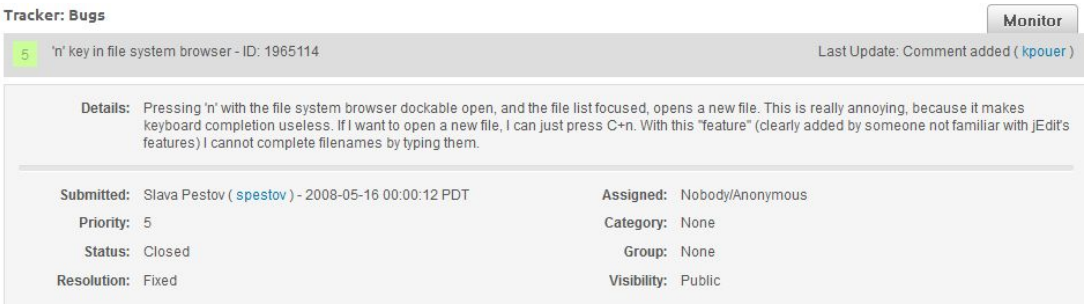



Figure 6. Bugzilla: Description for bug 1965114


Figure 7. ViewVC: SVN revision description for 'fix' to bug 1965114

The SVN revision description (Figure 7) identifies the jEdit commit that 'fixed' the bug (12672) and all the files that were modified during the fix. Often there will be multiple source code (`.java`) files modified for one fix. In this example the text file `CHANGES.txt` (the jEdit release history) has been updated to include a textual description of the bug fix and the single file `VFSDirectoryEntryTable.java` has been modified.

There are now three stages of analysis to be performed using the SCM system: identifying the jEdit revision where the bug was introduced; simulating the text approach; and simulating the dependence approach.

### *3.2.1. Origin Identification*

To identify the origin of the bug the BTS entry is read to see if there is any information that could aid understanding of the nature of the bug and its potential source. For bug 1965114 this helps since it indicates that the origin code is likely to be concerned with file shortcut options. This helps to focus the examination of the modified code.

Next, the commit that fixed the bug, 12672, is examined. First, the `CHANGES.txt` file was checked in case it contained helpful information; no further information was gained in this case. Second, the source file involved was examined using the annotated view provided by ViewVC. This provides a side-by-side SVN diff comparison against the preceding version of this file highlighting each line added, removed or changed. Figure 8 shows a snapshot of the annotated view for revision 12672 compared to its predecessor 12671. The lines highlighted in yellow (326–328) have changed between revisions and those in green (332) were added (any removed lines would be shown in red). Line 327 appears to be an added conditional to fix this bug.

In revision 12672 of `VFSDirectoryEntryTable.java` there were many other changes: ten different lines were changed in total, distributed throughout the file from line 128 to




DOI: 10.1002/smr

| # | Line 322 | Line 323 |
|---|---|---|
| 323 | ea = ac.getAction("vfs.browser.delete"); | ea = ac.getAction("vfs.browser.delete"); |
| 324 | ac.invokeAction(evt, ea); | ac.invokeAction(evt, ea); |
| 325 | break; | break; |
| 326 | case KeyEvent.CTRL_MASK \| KeyEvent.VK_N: | case KeyEvent.VK_N: |
| 327 | | if ((evt.getModifiersEx() & InputEvent.CTRL_DOWN_MASK) == |
| 328 | | InputEvent.CTRL_DOWN_MASK) { |
| 329 | evt.consume(); | evt.consume(); |
| 330 | ea = ac.getAction("vfs.browser.new-file"); | ea = ac.getAction("vfs.browser.new-file"); |
| 331 | ac.invokeAction(evt, ea); | ac.invokeAction(evt, ea); |
| 332 | | } |
| 333 | break; | break; |
| 334 | case KeyEvent.VK_INSERT: | case KeyEvent.VK_INSERT: |
| 335 | evt.consume(); | evt.consume(); |

Figure 8. ViewVC: Bug JEdit 1965114 side-by-side comparison of revisions 12671 and 12672 (the corresponding dependence graphs may be seen in Figure 13)

line 624. All of these changes must be considered as potential contributions to the bug fix. In this example these other changes appear to be concerned with opportunistic cleaning of the code — replacing a concrete `LinkedList` by the `List` interface and removing many unnecessary brackets. Such incidental changes can have unfortunate consequences for the two approaches.

Having determined that the fix is likely to be the changes at lines 326–332 in revision 12672 and that the erroneous code is therefore on line 326 of revision 12671, the task now is to identify the origin of this bug — in what revision was the bug introduced? By examining the SVN annotated view of revision 12671 (shown in Figure 9) the last revisions to change the context around line 326 are highlighted.

| | | | |
|---|---|---|---|
| 325 | vanza | 10326 | `case KeyEvent.CTRL_MASK \| KeyEvent.VK_N:` |
| 326 | ezust | 7998 | `evt.consume();` |
| 327 | | | `ea = ac.getAction("vfs.browser.new-file");` |
| 328 | | | `ac.invokeAction(evt, ea);` |
| 329 | | | `break;` |
| 330 | | | `case KeyEvent.VK_INSERT:` |
| 331 | | | `evt.consume();` |
| 332 | | | `ea = ac.getAction("vfs.browser.new-directory");` |
| 333 | | | `ac.invokeAction(evt, ea);` |
| 334 | | | `break;` |
| 335 | ezust | 7035 | `case KeyEvent.VK_ESCAPE:` |

Figure 9. ViewVC: Annotated revision view — the numbers in column 3 are hyperlinks to the last revisions that modified the lines in column 1

Opening the more recent revision 10326 associated with line 325 shows that there was only a change of layout in this revision, with no changes to the code. Opening revision 7998 reveals that all the code associated with this case branch was added in this revision, as shown in Figure 10. Revision 7998 is therefore deemed to be the origin of bug 1965114.

### *3.2.2. Text Approach*

The second stage is to now simulate the text approach. This means going back to revision 12672 which was identified as containing the bug fix. The evaluation was done as if each version of the code had been run through a preprocessor to standardise whitespace and brace formatting, and remove comments. This meant changes in whitespace, formatting and comments were disregarded. Adding or removing comment markers around lines of codes was therefore treated as adding or removing those lines. Changes to import statements were also ignored as these were either accompanied by other changes or were unused and so unrelated to the bug. Otherwise, the text approach assumes that all other changes made in



*Prepared using* smrauth.cls
DOI: 10.1002/smr

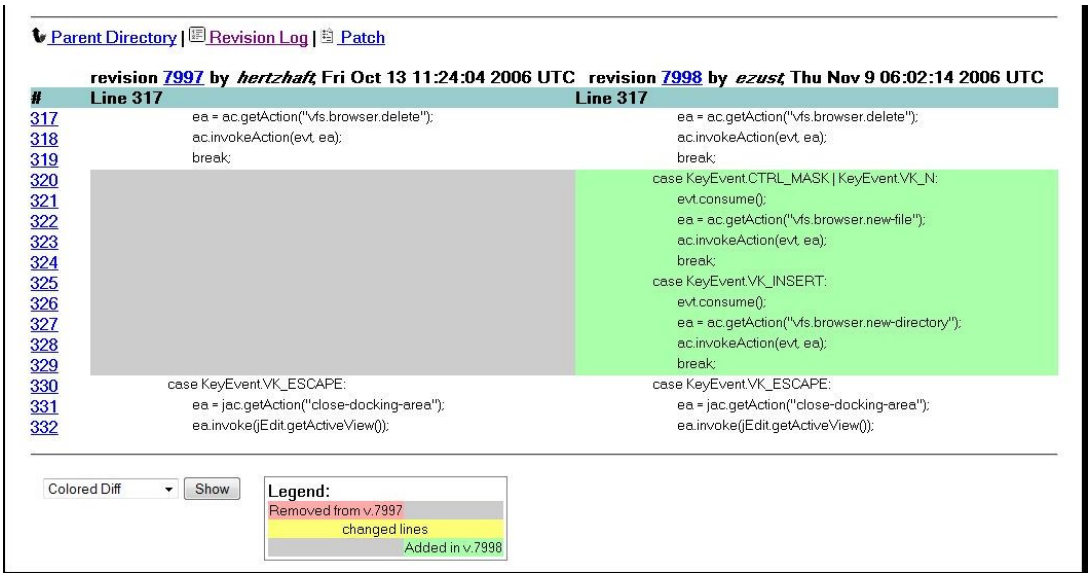



Figure 10. ViewVC: Code containing bug 1965114 was added in revision 7998

the revision that fixes the bug are part of the bug fix. Furthermore, as mentioned earlier, the text approach cannot deal with fixes that involve adding lines of code and so the focus is only on removed and changed lines.

The text approach works by taking each line that has been changed or deleted in the revision that fixes the bug (subject to the exclusions just mentioned) and tracking back through the revision history to identify the revisions that previously altered these lines. The set of revisions identified is considered the potential origin of the bug in the text approach.

Revision 12672 included ten separate line changes and one line addition. The ten line changes are therefore all traced back through the revision history as potential origins of bug 1965114. This is similar to the process described above to identify the origin of the bug. The problem is that the text approach cannot distinguish amongst any of the ten changes, as was done above where an understanding of code semantics was used to focus on the changes made around line 326 as being the actual bug fix. Tracking back for each of the 10 changes leads to a set of possible bug origins, potentially one origin for each change. In this case the set of revisions identified is 4640, 4648, 5179, 5217, 7170, 7998, 9596, 10275 and 11009. This is a clear example of one of the limitations of the text approach. Notice that the set *does* include the origin, 7998, as a result of finding the revision that introduced the change at line 326.

### *3.2.3. Dependence Approach*

The third stage is to simulate the dependence approach. The first step is to map methods between versions based on their name and signature. Although it did not occur in this case, this allows the dependence approach to handle methods which are relocated within the class (not true of the text approach).

Next, each method of the bug fix version is compared to the preceding version and any removed or added dependences are noted. In the case of removed dependences, each preceding version of an updated method is manually compared until the most recent version to add one of the dependences is found. If no dependences are removed then the same process is performed searching for the most recent version to have altered either the source or target of any added data/control dependences. In contrast to the text approach, rather than identifying the full set of potential bug origins, the dependence approach identifies



*Prepared using* smrauth.cls
DOI: 10.1002/smr

only the most recent revision associated with a removed/added dependence and assumes that revision is the single bug origin.

The dependence approach therefore starts with the SVN diff between revision 12672 and its predecessor 12671. Of the ten changes and one addition, eight are concerned with the removal of unnecessary brackets and one is the addition of a closing brace matching the new conditional at line 326. This leaves only two changes that could affect the data or control dependences. Only dependences on other lines within the same method were considered; dependences on other methods and on any fields were ignored.

The first change is line 128 which was altered from

```
LinkedList<VFSFile> returnValue = new LinkedList<VFSFile>();
```

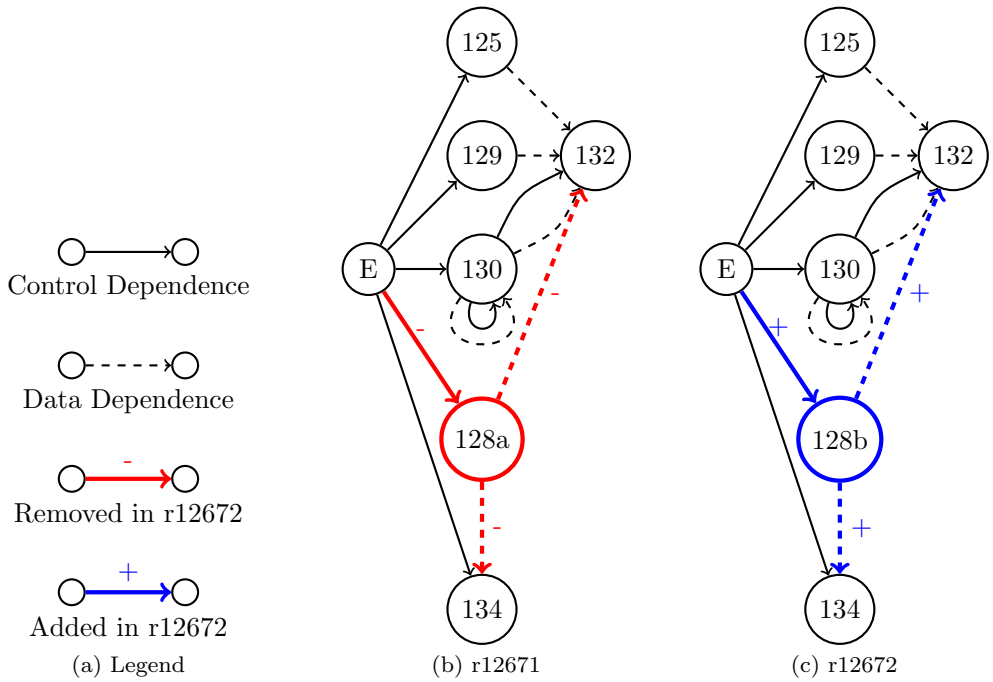
to

```
java.util.List<VFSFile> returnValue = new LinkedList<VFSFile>();
```

The dependences within this method are shown in Figure 11 (b) and the relevant sections of code in Figure 12. As shown, line 128 has a control dependence onto method entry and lines 132 and 134 have data dependences on line 128.

When comparing two methods, the first step of the dependence approach is to examine the graph for each version and determine which nodes in the first graph correspond to which nodes in the second graph, if any. This mapping is based on the similarity of the two lines and the lines surrounding them. Generally this was straightforward to do but where there were ambiguities these were recorded and re-examined at the end in order to ensure that similar cases were treated in the same manner. Specifically, if changes were made to an object's type, or a method was added to or removed from a chain of method calls, then the two nodes were considered to not map to one another. Changes that were made to the condition of an `if`-statement were regarded as the nodes mapping to one another. These decisions are somewhat arbitrary, but are based on suggestions in the original description [3], and ensure consistency in the evaluation.

As such, as this is a change of type, the two nodes representing line 128 are not considered to match. Therefore both the control and data dependences have been removed (and new




(a) Legend (b) r12671 (c) r12672

Figure 11. PDGs for `VFSDirectoryEntryTable.getSelectedFiles()`



DOI: 10.1002/smr

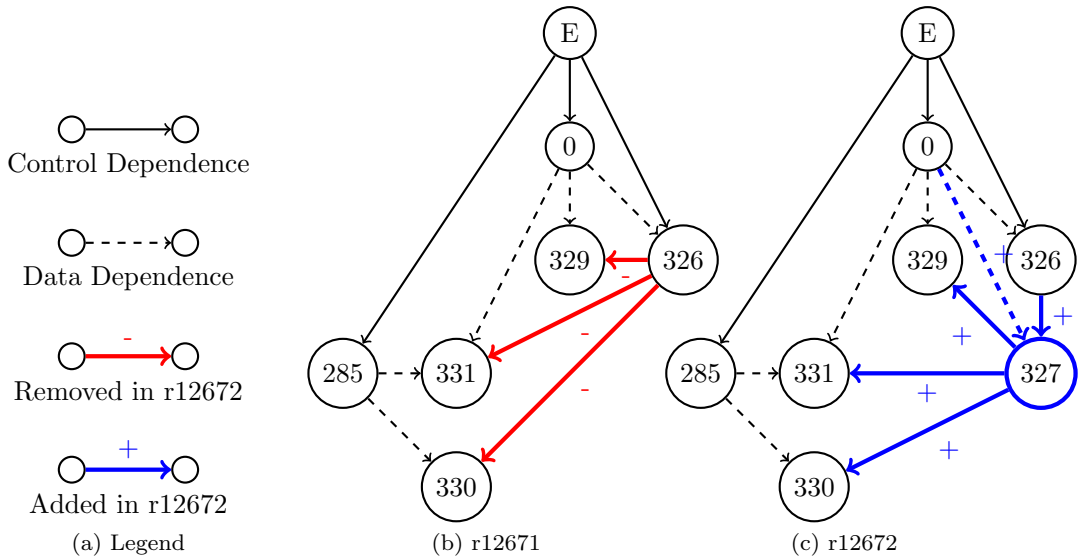
| # | revision 12671 by *ezust*, Tue Apr 22 23:12:43 2008 UTC<br>Line 125 | revision 12672 by *kpouer*, Fri May 16 11:56:05 2008 UTC<br>Line 125 |
|---|---|---|
| 125 | VFSDirectoryEntryTableModel model | VFSDirectoryEntryTableModel model |
| 126 | = (VFSDirectoryEntryTableModel)getModel(); | = (VFSDirectoryEntryTableModel)getModel(); |
| 127 | | |
| 128 | LinkedList<VFSFile> returnValue = new LinkedList<VFSFile>(); | java.util.List<VFSFile> returnValue = new LinkedList<VFSFile>(); |
| 129 | int[] selectedRows = getSelectedRows(); | int[] selectedRows = getSelectedRows(); |
| 130 | for(int i = 0; i < selectedRows.length; i++) | for(int i = 0; i < selectedRows.length; i++) |
| 131 | { | { |

Figure 12. ViewVC: Difference between revisions 12671 and 12672 in JEdit (corresponding PDGs shown in Figure 11

ones added), as shown by the highlighted edges in Figure 11. The dependence approach then searches the previous revisions to determine where any of those dependences were originally added. This process is performed in a similar manner to the text approach described above, using ViewVC to track back through the jEdit revisions. In this case, the generic parameter `VFSFile` was introduced in revision 9596, which would be seen as the introduction of the dependence, although prior to that the original data dependence was introduced in the baseline revision of jEdit, 4631. Had the two nodes representing line 128 instead been considered to match one another, then the two graphs would have been seen as identical: no dependences would have been seen as added or removed. This change would therefore not have led to any origin. The effects of these decisions are explored in more detail in Section 5.

The only other change is the code associated with the bug fix shown in Figure 8. Here the control dependences of the three assignments (starting `evt.consume();`) onto line 320 are removed and replaced with control dependences onto the new `if`-statement, as shown in Figure 13. In addition a new control dependence from the `if`-statement to the `case`-statement is introduced. As stated above, changes within conditionals are regarded as lines mapping to each other in the PDG, so the change on line 320 can be disregarded. Therefore, the focus here is on the removed control dependences and the revision(s) in which they originated. Since this is the code associated with the bug fix, tracking back to identify when these were introduced shows that they were all added in revision 7998.

As mentioned above, the dependence approach only identifies one revision: the most recent based on removed dependences (or added dependences if none are removed). Here




(a) Legend (b) r12671 (c) r12672

Figure 13. PDGs for `VFSDirectoryEntryTable.processKeyEvent()` which corresponds to the code in Figure 8 (Many unchanged nodes and dependences have been omitted)



DOI: 10.1002/smr

the dependence approach will return revision 9596. Note though that if the two versions of line 128 had been considered to be the same node, by ignoring type changes, then the approach would have returned the actual origin, 7998. This example therefore highlights the major impact that subtle changes in the definition of PDG differences have on the dependence approach, particularly since it only returns a single revision. This is explored further in Section 5.

## 4. RESULTS AND ANALYSIS

Before looking at the results of the Eclipse and Rachota evaluation, it is useful to highlight the breadth of issues raised by just the small sample of jEdit bugs.

### *4.1. Qualitative Analysis — jEdit*

A small, qualitative analysis was carried out using jEdit bugs, which is part of ongoing research. This was used by the authors to cross-check and develop an agreed understanding of the application of the text and dependence approaches and also to develop a standard process for summarising the bugs, their origin and the outcome of applying the techniques. The eight bugs discussed in Table I were specifically selected to highlight some of the subtleties that arose in the application of the approaches, particularly the dependence approach, across all three case studies. All authors independently applied both approaches to eight selected jEdit bugs. One of the jEdit bugs (1965114) was used as a detailed example in Section 3.2. Some of the issues that arose are discussed in more detail in Section 5.

Table I summarises the eight jEdit bugs. For each bug it: shows the jEdit bug identifier; gives a brief description of the bug; states the identified origin; lists the revision(s) identified by the text approach; lists the revision identified by the dependence approach; and highlights any interesting observations.

The jEdit study, though small, clearly demonstrates consistent findings with the main quantitative studies on Eclipse and Rachota which follow this section:

- There are examples where each approach clearly works as intended.
- Both approaches are impacted by intervening changes between the origin and the bug.
- The dependence approach is sensitive to the definition of program (and system) dependence graph used — it is not clear, however, that one specific definition will consistently provide the best results.
- Most bug fixes had many *opportunistic* modifications as well as the bug fix, this means that the text approach often returns many revisions (including the correct one), while the dependence approach is more likely to be misled because it only returns the most recent revision.
- Bug fixes which involve changing surrounding context rather than the problematic lines can mislead the text approach; the dependence approach has more potential to identify the origin in these cases.
- The text approach cannot deal with changes that only add code.

### *4.2. Quantitative Results — Eclipse and Rachota*

For each approach the predicted origins were compared to the manually identified origins. The predictions that the approach made correctly were recorded as true positives (TP). False positives (FP), versions that the approach predicted but which were not correct, and false negatives (FN), origins which were manually determined but the approach did not predict, were also recorded.*

*Full results are available from `https://personal.cis.strath.ac.uk/s.davies/`



*Prepared using* smrauth.cls
DOI: 10.1002/smr

Table I. 8 jEdit bugs

| | |
|---|---|
| **Bug** | 1965114 — Shortcut to create a new file in `VFSBroswer` is `n` rather than `Ctrl+n` |
| **Origin** | Revision 7998 — Shortcut code originally added |
| **TA** | 9 revisions including 7998 |
| **DA** | 9596 |
| **Notes** | Text approach identifies many revisions due to multiple opportunistic changes in bug fix file. Subtle questions raised for dependence approach: do type and generic type parameter changes cause PDG changes? |
| **Bug** | 1584436 — Off-by-one error, local variable `i` should have been decremented |
| **Origin** | Revision 5216 — Updated one use of `i` but not the others |
| **TA** | 5173 |
| **DA** | 5173 |
| **Notes** | Bug caused by change in context which misleads text approach. No change to data/control dependences in actual fix but changes elsewhere mislead dependence approach. |
| **Bug** | 1834620 — Lexer problem - only seems to recognise one character after $ |
| **Origin** | 4826 — Original functionality added here |
| **TA** | Nothing |
| **DA** | 4826 |
| **Notes** | Text approach can't deal with only added code. For dependence approach, data and control changes both lead to origin |
| **Bug** | 1541372 — Caret positioning on text folding. |
| **Origin** | Revision 3791 — Functionality introduced |
| **TA** | 4 revisions including 3791 |
| **DA** | 4567 |
| **Notes** | Intervening changes between origin and fix cause difficulties for both approaches. Fix includes changes to multiple files — one file relevant, one not. |
| **Bug** | 1600401 — Long lines cause editor to hang when `enter` pressed on them |
| **Origin** | 8103 Where the original condition added without length check |
| **TA** | 8107 |
| **DA** | Nothing |
| **Notes** | Intervening change causes difficulty for text approach. Questions raised about dependence approach: if adding conjunct to `if`-statement causes control change then it finds the wrong revision; if data change only then finds correct revision |
| **Bug** | 1571629 — Issue with filtering files in Windows, looks like was a problem with Windows from original |
| **Origin** | 6808 — Problem with Windows OS since beginning |
| **TA** | 6808 |
| **DA** | 6808 |
| **Notes** | Good example of both approaches performing correctly — all changes lead to origin. |
| **Bug** | 1548902 — Formatting empty paragraph throws exception, size isn't checked |
| **Origin** | revision 5196 — Move caret position introduced |
| **TA** | Many revisions including 5196 |
| **DA** | 5324 |
| **Notes** | Opportunistic changes, e.g. changing StringBuffer to StringBuilder, mean that text approach gathers many revisions. Dependence approach depends on PDG subtleties — type change gets wrong revision, otherwise correct. |
| **Bug** | 1542026 — Manipulating menus using keyboard — characters going into text file. |
| **Origin** | 5114 — Original coded added |
| **TA** | Nothing |
| **DA** | 5114 |
| **Notes** | Dependence approach works as intended and text approach fails because there is only added code. |



*Prepared using* smrauth.cls
DOI: 10.1002/smr

Table II shows a summary of where the origin of the bug lies for each of the bug fix commits in Eclipse and Rachota. The majority of commits fixed a single instance of a bug where the origin and the fix lay within the same file, shown as *Single*. Some commits contained fixes for more than one instance of the same bug, shown as *Multiple*. These bugs may have all originated at the same time or may have been introduced in different commits, but in all cases the origin and the fix lay within the same file. A number of commits contained fixes for bugs for which the origin actually lay in a commit made to a separate file and these are classed as *Elsewhere*. Finally some commits which claimed to be bug fixes were not actually fixes, but were instead consequences of the bug fix or unrelated changes, and these were classed as *Related* or *Unrelated*. Examples explaining the distinction between categories in greater detail will be given later in this section. For 20 commits the origin could not be determined, as the changes made by developers were complex and it was not possible for the authors to understand them or to identify if they were related to the bug in question. These commits were classed as *Unclear*. Note that these 20 files relate to just 3 bugs. All files for these bugs have been omitted from the remainder of the evaluation.

Only for bugs classed as *Single* or *Multiple* could the two approaches potentially return the correct answer. Identifying the origin of *Elsewhere* bugs would require an analysis over the whole system, not just on one individual file. Commits classed as *Related* and *Unrelated* claim to be bug fixes but there is actually no bug in the file; for these bugs there is no origin and therefore any answer returned by either of the two approaches would be an error.

Table III shows the results obtained by the text approach (TA) and dependence approach (DA). In order to help compare the performance of each the commonly used measures of precision (P) and recall (R) are also shown, along with their harmonic mean $F_1$-Score (F). As can be seen, for Eclipse the text approach identifies more correct versions than the dependence approach. However, the text approach also generates a much larger number of false positives, as it can return multiple origins for each commit while the dependence approach only returns a single origin. As to be expected from the lower number of false positives, the precision of the dependence approach is higher, at 44% compared to 29%, although the recall is only 40% compared to 48% for the text approach.

A similar pattern is seen for Rachota but the proportion of false positives compared to true positives is vastly reduced compared to Eclipse. This may be because of the relative simplicity of changes in Rachota. The bugs, and their fixes, often seemed simpler than those in Eclipse and there were often fewer versions between the origin of the bug and the fix. Unsurprisingly, given the smaller number of false positives, the precision and recall for Rachota are higher than for Eclipse.

### *4.3. Analysis of Eclipse and Rachota*

The approaches successfully found origins for a variety of different types of bug. Of the 68 bugs in Eclipse for which at least one origin was successfully identified (out of a total of 100 bugs), around a third resulted in exceptions that either crashed the application, displayed

Table II. Count of Commits by Classification of Origin

| | **Eclipse** | **Rachota** |
|---|---|---|
| Single | 161 | 88 |
| Multiple | 19 | 21 |
| Related | 57 | 6 |
| Elsewhere | 6 | 1 |
| Unrelated | 38 | 14 |
| Unclear | 20 | 0 |
| Total | 301 | 130 |



DOI: 10.1002/smr

Table III. Overall Results

| | | TP | FP | FN | P | R | F |
|---|---|---|---|---|---|---|---|
| Eclipse | TA | 91 | 220 | 100 | 0.29 | 0.48 | 0.36 |
| | DA | 77 | 98 | 114 | 0.44 | 0.40 | 0.42 |
| Rachota | TA | 89 | 39 | 38 | 0.70 | 0.70 | 0.70 |
| | DA | 85 | 23 | 42 | 0.79 | 0.67 | 0.72 |

an error to the user or appeared in logs. However, bugs in other areas were also successfully identified:

**UI** Bug 63753 — "Checkbox being ignored"

**Tests** Bug 74229 — "Failing automated tests"

**Code Reviews** Bug 57670 — "Wrong subclass of `InputStream` was being used"

**Performance** Bug 64531 — "Find/Replace operation using 100% CPU"

There was a similar diversity of bugs identified in Rachota, although there was a greater proportion of user interface bugs and incorrect output rather than exceptions. Overall the approaches did not seem to be more or less effective for any particular type of bug, but there were a number of factors that impacted on their performance. This section will highlight some of these with a number of illustrative examples of bugs.

*4.3.1. Graph matching, multiple bugs, age of origin and large fixes*

**Eclipse Bug 49561** — "Commit should only lock parent's folder"

Some bugs demonstrated a wide range of issues.

Although the description only mentions commit operations, the actual bug involved a number of different operations locking the entire Eclipse workspace, preventing other operations from accessing it, when it was only actually necessary to lock individual folders. One of the changes made to fix the bug was version 1.156 of `CompilationUnit.java`, in which a number of lines were changed from `runOperation(operation, monitor)` to `operation.runOperation(monitor)`. This was done as the method `JavaElement.runOperation` had been moved, and during the move changed, to `JavaModelOperation.runOperation`. Each of the calls was used as part of a different operation and therefore each represents a separate instance of the bug. Some of these bugs were introduced in version 1.1 of the file, while others were introduced in version 1.98.

**Graph matching** The text approach correctly identified both origins of this bug, as these lines had not been materially altered since the bug was introduced. However, as in jEdit bug 196511, the graph matching technique used has a major impact on what the dependence approach returns. If the line `runOperation(operation, monitor)` is considered to be the same node as `operation.runOperation(monitor)`, then the dependence approach would not return anything at all. However, as the evaluation considered those two lines to be different, the dependence approach returned the most recent version to introduce one of the relevant dependences, version 1.140. The technique used to match nodes between different graphs has a major impact on the conclusions reached by the dependence approach. However, as later examples will show, there is not one definitive strategy which will always give the right answer.



*Prepared using* smrauth.cls
DOI: 10.1002/smr

Table IV. Results by origin (Eclipse)

| | TA | | | DA | | |
|---|---|---|---|---|---|---|
| | **TP** | **FP** | **FN** | **TP** | **FP** | **FN** |
| Single | 79 | 75 | 82 | 73 | 41 | 88 |
| Multiple | 12 | 5 | 18 | 4 | 4 | 26 |
| Related | 0 | 88 | 0 | 0 | 29 | 0 |
| Elsewhere | 0 | 10 | 0 | 0 | 4 | 0 |
| Unrelated | 0 | 42 | 0 | 0 | 20 | 0 |

**Multiple bugs** Even if the dependence approach had returned *a* correct answer, it always only returns a single answer, so could not have hoped to identify both origins. This bug is an example of a number of bug reports in the Eclipse sample where one bug report actually represents multiple real bugs. Since these were often introduced at different times, the dependence approach fared much worse at identifying these, as shown by the row for *Multiple* bugs in Table IV.

**Age of origin** Quite a large number of versions passed between the bug being introduced and being fixed. While for this bug, and some others, the text approach was still successful, this was not generally true — the more versions passed between the introduction and the fix the less likely it was the origin could successfully be found, as will be shown in Section 5.4.

**Large fixes** `CompilationUnit.java` was only one of the files involved in this fix. In total, there were 13 files whose commit comments indicated they were related to this bug. However, the knock-on effects of moving the `runOperation()` method described above were responsible for most of these. In 3 of the files, there was no actual incidence of a bug, although the code changed was clearly involved in the fix. These commits were classed as *Related*. In an additional 7 files code that was not involved in the bug at all had to be altered and these commits were classed as *Unrelated*. For both these types of change, since there was never any bug in the file, any result returned by an approach would be wrong. The ideal outcome in fact would be for the approaches to return nothing. This did not happen however: these changes resulted in 19 false positives for the text approach and 7 for the dependence approach. This was often true of other bugs; as can be seen in Table IV these types of commits are responsible for a significant number of false positives.

#### *4.3.2. Unrelated changes*

**Eclipse Bug 49891** — "Problems launching a program, when using linked resources in CLASSPATH, which are not set"

This bug is an example of a fairly straightforward fix for which the origin is easily identified, but which can still result in issues for both approaches. In version 1.37 of `RuntimeClasspathEntry.java` the line

```
return res.getLocation().toOSString();
```

was altered to check first whether the value of `res.getLocation()` was null. This possibility had existed since version 1.1, the origin of the bug, but version 1.8 of the file had changed the line to its current version from the previous

```
return new String[] {res.getLocation().toOSString()};
```

Both approaches therefore returned version 1.8 of the file as the incorrect origin. Unrelated changes often tripped up both approaches, although in general the dependence approach appeared to handle them slightly better. In particular because it explicitly mapped methods




DOI: 10.1002/smr

between versions it could handle code being relocated within a file where the text approach could not.

It is difficult to see here how either approach could account for this problem however. The change in version 1.8 is indeed a semantic change and cannot simply be disregarded, as it is entirely possible that the same scenario could be the fix for another bug under different circumstances.

#### *4.3.3. Bugs caused by changes to other files*

**Eclipse Bug 54538** — "Bundle-SymbolicName value has changed"

Not all bugs were caused by a change made earlier in the same file. This bug was raised as the class `BundleManifest.java` was not correctly parsing the value of a `String`. It was fixed in version 1.6 of the file. However, the bug was originally caused by a change that was made elsewhere in the system to change the format of the `String`, not in the `BundleManifest.java` itself. At that time, a number of classes which parsed this value were updated, but this class was incorrectly omitted, and this bug then raised at a later date when the problem was discovered. As the origin lies in another class, this commit was therefore classed as *Elsewhere*, and neither the text approach nor the dependence approach could correctly identify the origin.

#### *4.3.4. Coincidental correctness*

**Eclipse Bug 65354** — "[Perspectives] Perspective bar incorrectly updated when opening new perspective"

The fix for this bug was in version 1.11 of `PerspectiveBarManager.java`. Here, two lines were updated from

```
if (coolBar != null) LayoutUtil.resize(coolBar);
```

to

```
if (getControl() != null) LayoutUtil.resize(getControl());
```

However, as neither `coolBar` or `getControl()` are local variables, their dependences are not included in either graph and the intra-procedural dependence approach would therefore not be able to determine their origin (although it is possible an inter-procedural approach could have).

An additional change was also made before those lines to remove a call to `updateCoolBar();` (that method was also removed). However, the `updateCoolBar()` method did not actually do anything: its entire contents had been commented out in an earlier version. The removed control dependence on the call however causes the dependence approach to return the correct origin for the bug. In effect, the approach is only correct due to the developer making an unnecessary change.

As this bug shows, developer behaviour has a major effect on the quality of the two approaches. They may be more effective for projects which have a strict policy on making one commit per logical change and less so for projects where developers tend to commit a large variety of changes all in one go. It also raises the question: even if a way could be found to ignore changes which are incidental or unrelated to the main fix, should they be, or are these still useful clues to help discover the origin? Here for instance, it was only due to the unrelated change that the dependence approach worked. After all, developers usually only change code if it is at least tangentially related to the code they need to fix; they are less likely to make changes to code which is totally disconnected from that involved in the bug.

It is also useful to note that the intended use of the approaches will have an effect on how useful they are perceived to be. If a developer for example selects a bug report and wants to know what caused it, issues like these will trip up the approaches. If on the other hand the developer identifies an individual change known to be the bug fix and wants to know the same thing, the approaches may be more useful. For many bugs, although not this one, the




DOI: 10.1002/smr

approaches could often identify the correct version if not for the presence of other changes in the file.

#### *4.3.5. Number of changes*

**Eclipse Bug 55640** — "[FastView] Screen cheese after Fastview resize"

Firstly, this bug highlights a major problem with trying to interpret developer changes through bug and commit data. The commit comment for this fix, version 1.56 of `ViewPane.java` was "fix for bug 55640: views with custom titles were wasting space". The developer is fairly unequivocally claiming to have fixed a bug with the correct ID and the code is clearly in the same area as that described by the bug. However, the description of the commit and the description of the bug do not entirely match up and it is not clear whether they are definitely referring to the same thing. It is indeed possible that the developer used the wrong bug number when committing the fix or that the bug report was not a true description of what the problem actually was. Alternatively, this could simply be a problem of understanding caused by the analysis not being performed by a developer of the system.

Regardless, it was fairly straightforward to identify the origin of the bug fixed by the developer as version 1.33 of the file. The fix itself involved the deletion (or commenting out) of 32 lines of code and modification of another 2, spread throughout the class. However, these lines of code were last changed in a variety of different versions: 1 in 1.22, 27 in 1.33, 1 in 1.34, 1 in 1.38, 1 in 1.48 and 3 in 1.53. This results in the text approach identifying the correct answer, but with an additional 5 false positives. The dependence approach identifies the most recent of these changes, 1.53, as the source and so gets the incorrect answer. The interesting point here is that the source of the bug was *not* the most recent change to the code (nor was it the oldest), but it was the version in which the majority of lines had last been changed.

#### *4.3.6. Non-local dependences*

**Eclipse Bug 63753** — "Team -> Tag as Version w/ "Move tag if it already exists" option does not work"

Again this bug had a fairly straightforward fix, for which the text approach could correctly identify the origin. A single line

```
if (confirmDialog.getReturnCode() == IDialogConstants.OK_ID) {
```

was changed to

```
if (confirmDialog.getReturnCode() == IDialogConstants.YES_ID) {
```

Like previous bugs, if these two lines are interpreted as *not* mapping to one another between dependence graphs then the dependence approach would also return the correct answer. If however, as was assumed here, these lines do map, then the dependences *within* the method have not changed at all. The only change that has been made is a dependence on a static variable. This highlights additional possibilities for altering the dependence approach. Even if a full inter-procedural analysis is not feasible, extending the dependence analysis to take into account constant or field declarations may still improve the recall of the approaches. It is not clear however how many bugs would be affected this way and there may be bugs where doing so would cause the wrong version to be returned.

#### *4.3.7. Added dependences*

**Eclipse Bug 50549** — "[Dialogs] Pref Page Workbench/Keys/Advanced does not use dialog font"

It is not at all clear whether the prioritisation of removed dependences over added dependences in the dependence approach is necessarily the correct technique. In the fix for this bug, version 1.66 of `KeysPreferencePage.java`, there were 2 lines removed, with corresponding control and data dependences removed. The most recent of these was introduced in version 1.63, which is what the dependence approach returned as the incorrect




DOI: 10.1002/smr

answer. However, there was also 1 line added. The corresponding control dependence, onto method entry, would have led the approach to return version 1.60, the actual origin of the bug.

### *4.4. Summary*

A number of key points can be taken from the examples given here and earlier:

- For many bugs only one of the approaches gains the correct answer, or indeed any answer at all.
- The decisions taken by the dependence approach on graph matching, depth of analysis and priority of dependences has the potential to hugely affect its performance.
- Large fixes, and the number of versions between a bug and a fix, can reduce the effectiveness of the approaches.
- Unrelated changes have an impact on the results returned, but this can sometimes be to the approaches' benefit.
- Taking the most recent version as that which introduced the bug may not be the most effective technique.

## 5. POTENTIAL IMPROVEMENTS

It is obvious from the sample, and from the examples highlighted above, that there are a number of areas in which the approaches could be improved. It is worth considering first why the approaches, or variations on them, should work at all. After all, it is not hard to construct theoretical examples of bugs for which neither approach would ever be successful, but these bugs do not appear to be frequent in the sample that was examined. Almost every bug fix involves either a textual change or a correction of an aberrant dependence. Using a theoretical extension of the dependence approach, with a richer variation of an SDG, even bugs which are fixed in other classes or by changes to inheritance hierarchies etc. could be seen as dependence changes. If the approaches were to consider *every* change, whether textual, added dependence or removed dependence and then to consider *every* previous version which affected these lines or dependences, it would seem very likely that the full set would include the bug origin. To do so would of course have huge costs, both in the large number of false positives and in the prohibitive time that would be required to run such a technique.

In a sense however, the various decisions taken by the two approaches can be seen as attempts to reduce both of these costs:

- Choosing to use just one of the approaches rather than both.
- Ignoring changes to whitespace or formatting.
- Only returning the most recent version.
- Ignoring added dependences if dependences have been removed.

It is not clear however if these are the most appropriate ways to improve the approaches. This section will present a number of other possible improvements or changes that could be made.

### *5.1. False Negatives*

The idealised description of the approaches given previously is just that: an ideal. In reality, there are a large number of false negatives returned by each approach. In a very small minority of cases these occurred when a bug had multiple origins of which the approaches only identified some, but for 70 commits in Eclipse classed as *Single* or *Multiple* the text approach returned nothing at all. As stated earlier, this was usually due to fixes which



DOI: 10.1002/smr

Table V. Combining Approaches

| | | TP | FP | FN | P | R | F |
|---|---|---|---|---|---|---|---|
| Eclipse | TA | 91 | 220 | 100 | 0.29 | 0.48 | 0.36 |
| | TA,DA | 116 | 231 | 75 | 0.33 | 0.61 | 0.43 |
| | DA | 77 | 98 | 114 | 0.44 | 0.40 | 0.42 |
| | DA,TA | 96 | 127 | 95 | 0.43 | 0.50 | 0.46 |
| Rachota | TA | 89 | 39 | 38 | 0.70 | 0.70 | 0.70 |
| | TA,DA | 109 | 43 | 18 | 0.72 | 0.86 | 0.78 |
| | DA | 85 | 23 | 42 | 0.79 | 0.67 | 0.72 |
| | DA,TA | 95 | 34 | 32 | 0.74 | 0.75 | 0.74 |

only involved lines being added. For the dependence approach the equivalent figure was 58 commits, where the fix usually contained no changed dependences.

The original authors proposed that in the cases where the dependence approach found no altered dependences it would fall back to using the text approach [3]. However, this could equally be applied in the other direction, with the text approach given first preference. In fact this may well be preferable due to the extra computational effort required for the dependence approach. The original paper reported the dependence approach to take around 7.2 times as long as the text approach on average, in the worst case taking over 12 hours to analyse 129 fixing commits where the text approach took around 1 hour.

A technique of using both strategies may well be viable. The most common result was for the two approaches to return the same outcome. However, in a significant number of cases one approach identified the correct version while the other did not, as already discussed. Obviously, however, it is also possible for one approach to return the incorrect answer where the other returned nothing.

Table V shows the effect of returning the result of the second approach if the first approach returns nothing. For both Eclipse and Rachota the change to the dependence approach is the same: a slight increase in both true and false positives, with a corresponding rise in recall but drop in precision. The change for the text approach is more pronounced however, increasing both precision and recall. The difference in $F_1$-Score between the approaches has now reduced and in fact for Rachota applying the text approach first gives the highest value. Given the potential time saving and the similarity in effectiveness, using the text approach first could benefit some applications.

There are perhaps more effective ways of combining the two approaches and of taking further evidence into account. As the earlier examples should have illustrated, there are often a number of decisions that need to be taken in each approach; there is not one correct way for every bug. Sometimes the most recent change is the one that introduced the bug, sometimes it is the first. Sometimes dependences are removed to fix the bugs, sometimes they are added. Sometimes type changes cause the bug, other times they are irrelevant. One potential way to approach handling this would be to aggregate this information and combine it in a probabilistic manner.

As a simplistic example, a fix in 1.7 involving 10 lines of code, 2 of which were changed in version 1.5 and 8 in 1.4, could imply that the bug was 80% likely to have been introduced in version 1.4. However, the fact that 1.5 was more recent could suggest that the bug is 66% likely to have originated in version 1.5. Multiplying these probabilities would say that the cause was version 1.4 with 65% probability. Obviously the examples given here are arbitrary, but the exact values could be determined empirically and additional evidence could also be taken into account.

In a way, some of the current decisions taken by the approaches can be seen as a blunt form of this: choosing to favour the most recent change is saying that there is a 100% probability it was in that version and 0% probability it was in any preceding version.



*Prepared using* smrauth.cls
DOI: 10.1002/smr

### *5.2. Inter-procedural analysis*

Even after combining the two approaches, there are still 62 commits for which no result would be returned. In some cases this is desirable — where the file did not actually have a bug in it — but often this is due to the dependence approach not being capable of detecting the dependences that have been changed. However, there are a number of additions that could be made to it in order to increase the number of dependences it considers.

Starting from the basic intra-procedural approach used in the evaluation, a simple step would be to take into consideration dependences on fields, static values or constants. Slightly more complex would be to add in dependences on other methods within the same class, then more complex still to include other classes. Finally, other information could be added in until there is a full SDG as presented earlier: synchronization, exception handling, inheritance hierarchies etc. Each of these layers adds on more information and brings the system closer to that described at the beginning of this section, both in terms of being more likely to find the origin and of having more false positives and a longer runtime. However, it may be that there is a level between the basic one used here and the full SDG which proves to be the most balanced of solutions.

Another way to increase the number of potential results discovered by the dependence approach is to consider *all* the dependences, rather than stopping at the one which had the most recent change. This would in effect make it more similar to the text approach, with similar downsides.

Another possible investigation is into the graph matching techniques used. The original source does not fully explain the technique used [3], but is suggested as being similar to that used in JDiff [15]. In that technique, graphs are broken down into smaller subgraphs called *hammocks* and nodes are matched by their structural properties or textual equality — nodes which are textually similar but not exactly equal are not considered to match. However, there are many other graph or tree differencing techniques. In ChangeDistiller [40] for example, n-grams are used to determine whether the labels of leaf nodes are above a certain threshold of similarity. Whether a node in one version is matched to a node in a subsequent version or not has a major effect on how these techniques work, as two nodes which don't match can result in a lot of removed dependences. Using a technique which gives a more accurate decision on whether a line has been updated or instead entirely removed and a new one added could give fewer false positives. Conversely however, using a *less* accurate technique may actually increase the number of dependences altered, which if multiple dependences are taken into account may give the approach more chance of finding the correct answer.

### *5.3. False Positives*

A significant proportion of the responses for the text approach were false positives. While most occurred when the approach could not identify the correct answer, a total of 41 false positives occurred where the approach identified the correct version along with one or more false positives, shown by the highlighted cells in Table VI. For example, there were 6 instances of commits which caused 1 true positive and 2 false positives, and 98 commits which caused 0 of each; nothing at all was returned for these commits. In addition, the bold

Table VI. Commits Classified by Number of True and False Positives for Text Approach (Eclipse)

| **FP** | **0** | **1** | **2** | **3** | **4** | **5** | 6 |
|---|---|---|---|---|---|---|---|
| **TP** | | | | | | | |
| 0 | 98 | 44 | **25** | **12** | **7** | **3** | **1** |
| 1 | 59 | 11 | **6** | **4** | **0** | **1** | **0** |
| 2 | 4 | 1 | **0** | **0** | **0** | **0** | **0** |



DOI: 10.1002/smr

cells show the large number of cases where more than one false positive occurred, for a total of 164 false positives.

One technique to reduce the number of these responses would be to return a single version, similar to the dependence approach. Doing so would reduce the maximum possible number of false positives per commit to one and would hopefully also result in discarding the false positive in favour of the true positive. Two ways to do so are to return the most recent version in which anything changed or to return the version in which the majority of the lines last changed.

Table VII shows the effect of such a change. As shown the false positives have decreased significantly. Unfortunately, the true positives have also decreased, as the approach no longer returns some of the correct versions it previously would have. This is especially true for bugs classed as *Multiple*, where it is no longer possible to correctly identify all of the origins. However, selecting the version in which most lines were last changed does significantly increase the precision at the cost of a smaller decrease in recall. Selecting the most recent version is similar but for a lesser benefit and larger downside.

### *5.4. Unrelated Changes*

In total there were 34 fixes containing unrelated changes, of which 20 fixed multiple different bugs. An automated tool could potentially ignore these commits by checking the commit message for multiple bug IDs. Unfortunately, 13 of these commits were for a bug that was classed as *Unclear* and the remaining sample did not allow conclusions to be drawn about such a change.

One situation where this could be improved is where multiple commits are made to fix a bug, often because the first attempt was incorrect. One improvement would be to ignore changes made in versions that were linked to the bug other than the latest. This is in effect similar to the suggested improvement to remove all versions after the bug was raised as possible origins [7].

Possibly due to the increased chance of unrelated changes, the approaches tended to get less effective as the time between the bug being introduced and being fixed increased. Figure 14 shows the effects on F if fixes more than a given number of versions after the origin were to be ignored. Larger values have been omitted from the chart as the values remain largely stable, but the maximum difference between a bug being introduced and being found in Eclipse was 267 versions. Note that these figures only include bugs for which an origin actually existed so will not correspond with those given earlier in Table III.

Figure 14 also illustrates that bugs were often fixed shortly after being introduced. For Eclipse, 12.8% of bugs were fixed within 1 version after the bug was introduced with 50% of bugs being fixed within 12 versions.

Another, more complex, way of removing false positives would be to ignore changes to code that had no semantic effects. A number of common patterns were identified by Jung et al. [22], from simple renames or deletions of lines with no effect, to deep semantic analysis. Some of these are already carried out, whilst others could perhaps be easily achieved by using abstract syntax trees rather than straight textual differencing, but detection of other patterns would be challenging. While Jung et al. only considered ignoring these changes in the fix itself, it would also be possible to apply such techniques to changes in intervening versions.

Table VII. Returning Single Version for Text Approach (Eclipse)

| | **TP** | **FP** | **FN** | **P** | **R** | **F** |
|---|---|---|---|---|---|---|
| TA | 91 | 220 | 100 | 0.29 | 0.48 | 0.36 |
| Majority of Lines | 75 | 103 | 116 | 0.42 | 0.39 | 0.41 |
| Most Recent | 66 | 112 | 125 | 0.37 | 0.35 | 0.36 |



DOI: 10.1002/smr

Table VIII. Number of Files Per Bug (Eclipse)

| | Number of Bugs | | Number of Bugs |
|---|---|---|---|
| 1 | 55 | 7 | 3 |
| 2 | 16 | 10 | 2 |
| 3 | 7 | 13 | 1 |
| 4 | 4 | 21 | 1 |
| 5 | 5 | 22 | 1 |
| 6 | 1 | 24 | 1 |

### *5.5. Large Commits*

Eclipse Bug 49561 showed a large number of false positives being returned when there were many files involved in the bug fix. This appeared to be common: often bug fixes which involved many files had a large number of unrelated changes, leading to an increase in false positives. One proposal is to ignore fixes that change a large number of files [7]. The number of bugs of each size in Eclipse is shown in Table VIII.

Figure 15 illustrates F if bugs with more than the given number of commits were to be ignored. Similar trends are present for P and R. As seen here, the vast majority of bugs were small and the scores are much better for smaller commits. For the few larger bugs the scores decrease. The same was not necessarily true of Rachota; there the largest bugs were smaller and the scores stayed largely constant.

Given the results for Eclipse, ignoring bugs with more than a certain number of commits may increase effectiveness without reducing applicability significantly. This might be appropriate for some use cases but further study would be needed to determine a threshold and it is likely this may vary by project.

### *5.6. Incorporating users*

Whilst automated methods to reduce false positives are desirable, they are unlikely to be highly accurate and in some cases they may not be necessary. Depending on the desired use of the approaches, it may be that users could be available to manually remove false positives and to assist the approaches in other ways.

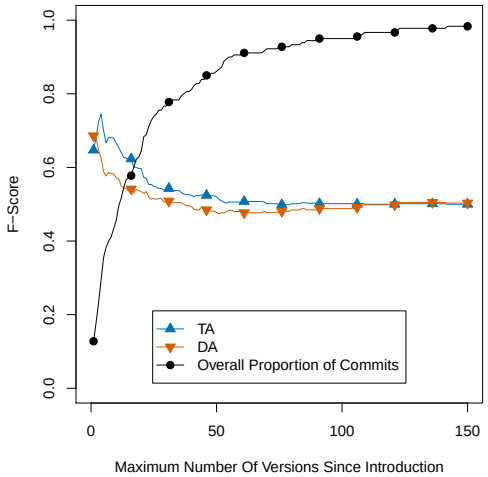



Figure 14. Age of Origin (Eclipse)



DOI: 10.1002/smr

If the approaches are being used to assist a developer with finding the origin of a specific bug then a reasonable first step would be to present the developer with a list of all changes in the fix. From there, the developer could select any changes which are obviously not related to the bug. This could easily reduce the number of false positives quite significantly as a number of irrelevant changes could be ignored. This process could also be carried out throughout the search by presenting the user with a list of proposed origins. Again, the user could select some as being irrelevant and the approaches could then follow the trail back through older versions.

As well as potentially being more useful for a developer, such a technique could also lead to a better understanding of what type of changes are responsible for causing or fixing bugs. This information could then be incorporated into the approaches to improve them, learning from the developer's feedback. As an example, there is a variable in Rachota which is displayed on screen to the user and indicates the version of the software. This variable is updated with every release and alone is responsible for 14 false positives in the Rachota sample. Being able to learn to eliminate these types of change would improve their accuracy.

### *5.7. Analysis of Potential Improvements*

As a way of assessing the likely impact of several of these suggested improvements, a diverse set of bugs was selected in order to investigate in detail the consequences of a number of possible changes. Five bugs were randomly chosen from the set of Rachota bugs, two from Eclipse, and all eight of the JEdit bugs previously examined. Although this is a small set, and not one which could be argued as representative, the differences between the bugs and the insights obtained from the process give some very clear pointers about which approaches are likely to yield substantial benefits and which are not worth considering. Selection of the potential improvements was based on what could be feasibly investigated and objectively evaluated and consequently tended to include changes in the applications of the two strategies and exclude larger, more amorphous, amendments such as investigating user involvement (such modifications would need to be the subject of a larger, separate study). The modifications for the text approach were to:

- Return just the most recent revision
- Return just the largest revision (that with the most substantial number of modifications in terms of added/deleted/changed lines)
- Consider the impact of additions by looking at where the surrounding block was introduced (i.e. treat the enclosing block as if it was the subject of some change)

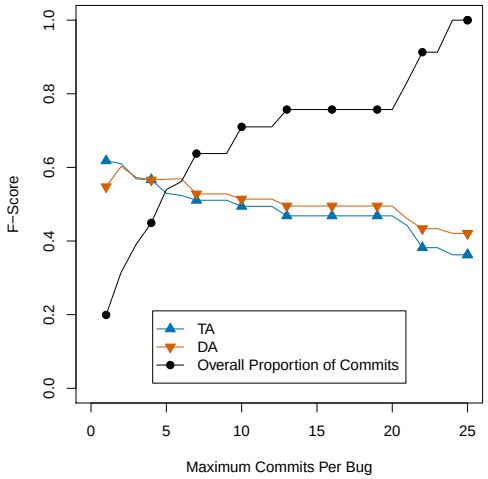



Figure 15. Performance of Different Sized Fixes (Eclipse)



DOI: 10.1002/smr

- Return just the most recent revision taking into account any extra revisions identified from considering the impact of additions
- Return just the largest revision taking into account any extra revisions identified from considering the impact of additions

The modifications for the dependence approach were to:

- Return *all* revisions that could be identified by examining changes in dependences
- Return just the largest revision (that with the most substantial number of modificiations)
- Take in account additions even when there are removed dependences and return the most recent revision
- Return all revisions taking into account any extra ones identified after considering additions
- Return just the largest revision taking into account any extra ones identified after considering additions
- Widen the scope of dependences considered to include:
  - intra-class dependences
  - inter-class dependences
  - system-wide dependences (library classes etc.)

  and for each case again consider the impact of returning all revisions and just the largest revision, as well as the standard approach of just the most recent revision.

Some of these changes are fairly trivial to implement (for example, selecting either the most recent or the largest revision for the text approach) whereas others (such as those considering text addition, and widening the scope of dependences) involved a considerable amount of additional analysis of code to identify the associated revisions. All these modifications were applied manually by the second and third authors to the selected bugs: five bugs were analysed independently by both researchers and subsequently discussed in order to ensure agreement on the application of the modifications, and the remainder were investigated by one or other researcher alone.

The results of applying these potential improvements to the selection of bugs are shown in Table IX and Table XI. These tables present the results (in terms of TP, FP and FN) for each of the standard (i.e. original) approaches, followed by the results of examining each of the potential improvements. In the tables a dash (-) indicates that there was no additional information to consider. This covers cases such as:

- The text approach only returning one result, so there are no other revisions to classify as either largest or most recent.
- There are no additions for the text approach to consider.
- The dependence approach has already considered additions (in the case where there are no removed dependences) or there are no added dependences to consider.

Where there is any new information to consider the results are reported, even if they don't produce any different values. This strategy clearly distinguishes between the consequence of considering the improvement and the case where the improvement cannot be applied. Table X and Table XII summarise the results from Table IX and Table XI by using the $F_1$Score. Individual table cells are also shaded, relative to the standard approaches, to visualise the benefits or otherwise of the potential improvements: in cases where there is no change the cells remain the same shade as the standard approach, but where there are improvements (deteriorations) in performance then the table entries are lighter (darker) than the standard approach.

The results of widening the scope of dependences are not shown in the table as overall this had very little impact. Any additional dependences tended either to identify no new revisions on top of those identified by the local changes, or resulted in a large number of



*Prepared using* smrauth.cls
DOI: 10.1002/smr

false positives (and a great deal more analysis). There were no cases where consideration of a widened dependence resulted in the identification of a new true positive. Investigating this further it would appear that the only case where a remote change is likely to be the source of a bug fixed locally is where the invoked method causes a side-effect, either by modifying some class-level state (a field for example) or by mutating a parameter in some way. None of the fifteen cases investigated involved such side-effects, which is not to suggest that this would never happen; on the contrary, Section 5.2 suggests that there are several cases where the inability to follow dependences results in nothing being returned. From this limited analysis a full-blown SDG-based analysis would not appear to be worthwhile when extending the analysis to consider only class-level or global state manipulation may be just as effective. This is something which may differ for projects which have a different coding style; perhaps as a consequence of the functionality demands of the system, the preference of the developers, or the choice of implementation language for example.

Considering the improvements to the text approach it is apparent that taking into account additions has a clear benefit in correctly identifying a further four revisions (with few false positives). This is a clear deficiency of the text approach so it is perhaps unsurprising that the impact should be so substantial. The fixes for three of the bugs (R1551008, J1542026 and J1834620) took the form of pure additions and so moved from returning nothing to correctly identifying the origin of the bug. For bug E50549 an addition was one of several fixes and so considering this correctly identified the origin. Bug R1356348 was similar, but the correct location was masked by intermediate fixes which prevented it from being identified. For bugs where the text approach had already correctly identified the source, considering additions tends to either have no detrimental effects or slightly increases the numbers of false positives. Filtering these results further by looking at either the largest or most recent revision does not produce any clear benefits (losing a correct result in one case), and filtering the original standard results yields substantial losses – findings that are in line with the earlier analysis in Section 5.3. Only in one case (bug R1734799) is there a substantial gain in the form of a large reduction in false positives from considering just the largest change. However, this is a slightly unusual bug in that the majority of changes are unrelated modifications to property files, and an example of where user involvement would be by far the most effective mechanism for filtering out such irrelevant changes.

For the dependence approach the biggest benefit comes from returning all revisions identified rather than the most recent, yielding an additional six correct revisions in addition to the seven intially identified by the original approach. This naturally results in an increase in the number of false positives, but not a substantial one in this sample. Considering just the largest revision from this wider set is successful in four cases but loses the correct revision in the remaining two, so if false positives were a significant issue this could represent an efficient, but more risky, approach. Taking into account additional dependences even where there are deleted ones would not appear to be worthwhile; the consequences are a marginal increase in the number of false positives in one case and no observable changes in the other two. The origins of two bugs remained undetected in spite of all these improvements. For bug R1356348, as with the text approach, the presence of intermediate changes prevented the correct location from being identified. For this bug there were also a number of additional wider dependences to consider, but none of these returned any new possible sources. Bug J1584436 (as noted when the JEdit bugs were originally considered) involved the failure to decrement a local variable and the fix did not involve any changes to either data or control dependences; such cases will always be missed by the dependence approach.

Considering both approaches together, on this sample the dependence approach performs far better than the text approach and detects 7 of the 15 bugs using the standard approach against the 4 returned by the text approach (which are also in the set detected by the dependence approach). Furthermore, 3 of the 5 revisions that would be correctly identified by the improvements in the text approach were already detected by the standard dependence approach; these three are the cases where the fix involved straightforward additions so it is no




DOI: 10.1002/smr

Table IX. Text Approach Improvements

| | Standard | | | Standard (recent) | | | Standard (largest) | | | Std. w/ Additions | | | Additions (recent) | | | Additions (largest) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bug ID | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN |
| R1266799 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| R1356348 | 0 | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| R1522751 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| R1551008 | 0 | 0 | 1 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| R1734799 | 1 | 22 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 22 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| J1600410 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| J1571629 | 1 | 0 | 0 | - | - | - | - | - | - | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| J1548902 | 1 | 4 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 4 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| J1542026 | 0 | 0 | 1 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| J1965114 | 1 | 8 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | - | - | - | - | - | - | - | - | - |
| J1584436 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | - | - | - | - | - | - | - | - | - |
| J1834620 | 0 | 0 | 1 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| J1541372 | 1 | 3 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 3 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| E50549 | 0 | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 2 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| E55640 | 1 | 4 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | - | - | - | - | - | - | - | - | - |

surprise that the dependence approach would also detect these (in general one would expect the dependence approach to detect all textual changes except where the modification is minor and there is no change in dependences, such as in the case of bug J1584436 mentioned above).

Combining this information would suggest that the following 3-step strategy would be most likely to provide the best results for the least cost:

1. Apply the standard dependence approach and inspect the revision returned.
2. If the revision identified by step 1 is incorrect then apply standard dependence (largest) and inspect the revision returned.
3. If the revision identified by step 2 is incorrect then apply standard dependence (all) and inspect the revision returned.

For the bugs considered here, this would correctly identify 13 (7 from step 1, plus 4 from step two, plus 2 from step 3) of the 15 revisions at a cost of examining only 19 false positives (8 in step 1, 4 in step 2, and 7 in the final step). However, other strategies could give similar results for slightly different costs: for example, applying standard text, followed by standard dependence and finally by largest dependence would also produce the same number of revisions at the cost of examining a larger number of false positives but with the benefit of having to carry out less in the way of expensive dependence analyses. Furthermore, this combined approach offsets some of the dangers of investing in a single strategy: as mentioned in Section 5.1, 70 of the commits in Eclipse returned nothing at all (being typically just additions) and 58 contained no changes to dependences. An approach which combines techniques would appear to be ultimately more effective, if slightly more expensive, in the light of such figures. Such trade-offs also have to take into consideration the technology available to support the analysis. Finally, it is important to stress that these findings are not readily generalisable from this small sample and require wider empirical investigation in the future.

### *5.8. Recommendations*

Even with this additional analysis the question of which approach is appropriate for use is not a simple one and may come down more to the intended use and time available rather than accuracy. Generally speaking, if the most important issue is accuracy then combining the two approaches in some way and including the fairly straightforward improvements



DOI: 10.1002/smr

Table X. F-Scores for Text Approach Improvements (shading highlights direction of improvement)

| Bug ID | Standard | Standard (recent) | Standard (largest) | Std. w/ Additions | Additions (recent) | Additions (largest) |
|---|---|---|---|---|---|---|
| R1266799 | 1 | - | - | - | - | - |
| R1356348 | 0 | 0 | 0 | 0 | 0 | 0 |
| R1522751 | 1 | - | - | - | - | - |
| R1551008 | 0 | - | - | 1 | 1 | 1 |
| R1734799 | 0.0833’ | 0 | 1 | 0.0833’ | 0 | 1 |
| J1600410 | 1 | - | - | - | - | - |
| J1571629 | 1 | - | - | 0.666’ | 1 | 0 |
| J1548902 | 0.333’ | 0 | 0 | 0.333’ | 0 | 0 |
| J1542026 | 0 | - | - | 1 | 1 | 1 |
| J1965114 | 0.2 | 0 | 0 | - | - | - |
| J1584436 | 0 | 0 | 0 | - | - | - |
| J1834620 | 0 | - | - | 1 | 1 | 1 |
| J1541372 | 0.4 | 0 | 1 | 0.4 | 0 | 0 |
| E50549 | 0 | 0 | 0 | 0.5 | 0 | 1 |
| E55640 | 0.333’ | 0 | 0 | - | - | - |

Table XI. Dependence Approach Improvements

| | Standard | | | Standard (all) | | | Standard (largest) | | | Std. w/ Additions (recent) | | | Std. w/ Additions (all) | | | Std. w/ Additions (largest) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bug ID | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN |
| R1266799 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| R1356348 | 0 | 1 | 1 | 0 | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 2 | 1 | 0 | 1 | 1 |
| R1522751 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| R1551008 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| R1734799 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - |
| J1600410 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| J1571629 | 1 | 0 | 0 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| J1548902 | 0 | 1 | 1 | 1 | 4 | 0 | 0 | 1 | 1 | - | - | - | - | - | - | - | - | - |
| J1542026 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| J1965114 | 0 | 1 | 1 | 1 | 2 | 0 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - |
| J1584436 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | - | - | - | - | - | - | - | - | - |
| J1834620 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| J1541372 | 0 | 1 | 1 | 1 | 6 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 6 | 1 | 1 | 0 | 0 |
| E50549 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - |
| E55640 | 0 | 1 | 1 | 1 | 3 | 0 | 0 | 1 | 1 | - | - | - | - | - | - | - | - | - |

presented here (primarily considering additions for the text approach and extending the dependence approach to include multiple results, which both result in quite substantial gains), is likely to be the most effective tactic. However, if only one can be used then the dependence approach (again with some of the identified improvements) tends to generate far fewer false positives whereas the text approach has speed advantages, both in use and in likely implementation time. Even though the dependence approach appears to perform well in this detailed analysis (particularly taking into consideration the improvements), the results of the evaluation in Section 4 would suggest that it does not appear to be sufficiently more accurate than the text approach to justify its sole use in most cases (note that this is based on the dependence approach as it was simulated here). There does appear to be more scope for improvement in the dependence approach than in the text approach which



*Prepared using* smrauth.cls
DOI: 10.1002/smr

Table XII. F-Scores for Dependence Approach Improvements (shading highlights direction of improvement)

| Bug ID | Standard | Standard (all) | Standard (largest) | Std. w/ Additions (recent) | Std. w/ Additions (all) | Std. w/ Additions (largest) |
|---|---|---|---|---|---|---|
| R1266799 | 1 | - | - | - | - | - |
| R1356348 | 0 | 0 | 0 | 0 | 0 | 0 |
| R1522751 | 1 | - | - | - | - | - |
| R1551008 | 1 | - | - | - | - | - |
| R1734799 | 0 | 0.666’ | 1 | - | - | - |
| J1600410 | 1 | - | - | - | - | - |
| J1571629 | 1 | - | - | 1 | 0.666’ | 0 |
| J1548902 | 0 | 0.333’ | 0 | - | - | - |
| J1542026 | 1 | - | - | - | - | - |
| J1965114 | 0 | 0.5 | 1 | - | - | - |
| J1584436 | 0 | 0 | 0 | - | - | - |
| J1834620 | 1 | - | - | - | - | - |
| J1541372 | 0 | 0.222’ | 1 | 0 | 0.222’ | 1 |
| E50549 | 0 | 0.666’ | 1 | - | - | - |
| E55640 | 0 | 0.4 | 0 | - | - | - |

could lead to it becoming sufficiently more accurate in the form of widening the scope of the analysis. However, these changes were not particularly fruitful and are only likely to exacerbate the differences in time required. Widening the dependence scope to consider field variables, statics and constants may be an effective compromise.

## 6. THREATS TO VALIDITY

There are are a number of threats to the reliability and generalisability of the findings from this evaluation. These include the manual application of both the text and dependence approaches, challenges in interpreting the dependence approach, researcher consistency, the accurate determination of bug origins, and potential limitations caused by the particular software systems and bugs selected for analysis.

For the main evaluation both the text and dependence approaches were manually applied as described in Section 3.2. As well as having the potential for human error in their application there were also occasional difficulties in determining precisely how they would perform when automated, particular with regard to the dependence approach. As discussed earlier, there are different possible interpretations of data and control dependence, including issues associated with intra- and inter-procedural dependences, type changes and changes within conditional statements. Furthermore, the handling of constructs such as fields, `synchronized` blocks and `try-catch` blocks were not discussed in the original work. To maintain consistency scenarios involving such constructs were noted as encountered and then revisited as a group at the end of analysis.

The analysis of Eclipse and Rachota was carried out by one researcher (the first author). To try to ensure clear and consistent application of the research procedure all three researchers independently applied the procedure to eight jEdit bugs Section 4.1. The researchers then met and performed a detailed group walkthrough of the analysis of each of these bugs to clarify all the issues raised and to ensure consistency of application. The correct identification of the origin of each bug is also a potential threat, especially since none of the researchers were developers of the software systems that were being studied.



*Prepared using* smrauth.cls
DOI: 10.1002/smr

This threat could not be avoided altogether, but was lessened by noting related and similar commits and again revisiting them at the end of analysis. Also, as described in Section 3.2, where possible, commit logs and bug report comments were used to support identification of bug origins.

For Eclipse and Rachota the analysis only ever finds a bug origin in the file where the fix was applied. However, as discussed, the origin to a bug sometimes lay elsewhere. This limitation is partially due to the SCM, CVS, used in these studies which considers each file in a commit as a separate transaction. This threat could partially be addressed by further studying projects that use other SCMs which assign revision numbers to the entire project, such as with the use of SVN in the jEdit study, or to use techniques that reconstruct project versions [41].

Particular threats to the generalisability of the results are associated with the systems studied, the relatively small number of bugs analysed and the particular bugs that were selected for analysis. With regard to the number of bugs selected from Eclipse (100 from a possible set of just over 4000), if a medium effect size is assumed (clearly a matter of judgement but arguably reasonable from inspection of the various graphs in the paper) then this sample of 100 from 4000 is enough to give any statistical test carried out a power of just less than 0.95. The manual analysis of bug origins and application of both approaches is quite time-consuming — about 30–60 minutes per bug on average. The only criteria used for selecting bugs to be analysed was that they could be linked via a bug identifier to a fixing commit, and so these may not be representative of all bugs in the three systems [4]. Similarly, the three software systems studied Eclipse, Rachota and jEdit, and their bugs, may not be representative of all software systems.

For the analysis of improvements, both the text and dependence approaches were again manually applied as described in Section 3.2, but also took into account a number of possible improvements. In terms of application of the approaches, the most significant of these was consideration of the wider set of dependence approaches which typically involved a good deal more analysis. For the detailed examination of all these potential improvements to be feasible it was only possible to explore their impact on a small set of bugs, and consequently these findings are only indicative and not necessarily generalisable.

In considering the above threats it is important to note that the aim of this study was to gain a deeper insight into how the text and dependence approaches may actually perform when applied to real software systems. The goal has not been to investigate whether one approach is 'better' than the other but to identify their apparent strengths and weaknesses, the key issues that arise with their use in practice and to determine the extent which bug origins might be automatically identified. Some confidence in the findings can be derived from the fact that same key issues repeatedly arose across the wide variety of bugs that were studied.

## 7. CONCLUSIONS

The aim of this study was to investigate the practical application of the text and dependence approaches for discovering the origins of bugs. The study was performed by manually simulating both of the approaches on sets of bugs from three case studies: Eclipse, Rachota and jEdit. The research goals included: identifying the strengths and weaknesses of both approaches when applied to real systems; determining how effective both approaches are in identifying bug origins; and suggesting refinements that could improve the overall effectiveness of the two approaches. The motivation for this research is that automated bug origin discovery can then feed into research on bug prediction and prevention based on common patterns or code locations, and analysis of the development process that leads to the introduction of bugs.

The study manually investigated the origins of 174 bugs and analysed the potential of the text and dependence approaches to correctly identify these origins. The quantitative




DOI: 10.1002/smr

evaluation based on Eclipse and Rachota found that both approaches often performed as intended, identifying the bug origin with a precision in the range 29%–79% and a recall in the range 40%–70%. The evidence suggests that the text approach is more likely to identify the correct origin than the dependence approach but at the cost of reduced precision — this is due to the text approach, as originally described, identifying *all* possible origins while the dependence approach seeks to identify a *single* origin. This choice of all versus single is actually independent of the approach used and either approach could choose to adopt one strategy or the other.

Other key lessons derived from this study include:

- The accuracy of using both approaches together was at least as good as, and mostly better than, using either on its own.
- Both approaches are likely to provide inaccurate results when other 'opportunistic' changes are made coincident to the fix. The text approach can accommodate this, at a cost to precision, because, as originally specified, it returns all possible origins.
- Unrelated changes made between the origin and the fix often caused incorrect results — this is a particularly difficult issue for both approaches.
- Both approaches became less accurate as more versions passed between the bug origin and the fix, though, in the systems studied, most bugs were fixed soon after their origin.
- Most bugs required changes to only a few files; bugs which required many changes could often not be resolved, particularly those where the fixes involved multiple files.
- The dependence approach is sensitive to the definition of program (and system) dependence graph used. It is not clear that one particular definition will consistently provide the best results. Nor is it clear that inter-procedural dependence provides benefits that outweigh the associated complexity and, likely, high execution time.
- The text approach, as originally defined, cannot cope with bug fixes that only add code.
- Bug fixes that modify the context of the origin rather than the same lines as the origin are more likely to mislead the text approach than the dependence approach.

There are a number of threats to these findings mainly concerned with the manual simulation of the approaches by one individual applied to only three case studies and a involving a relatively small number of bugs. However, the fact that the key findings listed above were repeatedly discovered across different bugs and the different systems, supported by a documented, repeatable research procedure should provide confidence in these results.

The level of precision and recall discovered for the systems studied would seem reasonable for some applications, especially if it is sufficient to return a relatively small set of files which is likely to contain the bug origin. Other applications may demand higher levels of precision and recall, and to investigate this a number of refinements were identified and their impact explored by the manual application of the techniques on a small range of previously examined bugs. The findings of this detailed analysis were as follows:

- The two approaches may be used together and, along with the suggested improvements, combined in various ways. The authors of the dependence approach [3] suggested that dependence might be followed by text when there were no altered dependences. The research here also suggests that it is likely to be more efficient, and only marginally less effective, to apply text first and then dependence. Alternatively, both approaches might be applied simultaneously and results determined based on either the union or intersection of their results.
- The dependence approach, as originally defined, only returns one potential origin, the most recent. When it works as intended, this will help precision; however, the bug fixes studied here suggest this is often unlikely. If high recall is important then the dependence approach should be altered to return a range of possible origins. This is




DOI: 10.1002/smr

a relatively simple extension (although there is the cost of the additional dependence analysis), but one which yielded noticeable benefits in the analysis of improvements.
- A fundamental weakness of the text approach, as originally defined, is that it cannot deal with fixes that only add code. A simple extension to the text approach which considered the enclosing block to be the subject of a change proved very effective on the small number of bugs examined in detail.
- Extending the dependence approach to include all dependences within the SDG does not appear to be worthwhile from this limited study. Widening the strategy to include references to class-level state may be worthwhile but needs futher investigation.

This research has also identified a number of other refinements that may be worth exploring, either manually using the procedure adopted here, or via an implementation:

- Both approaches, and potential refinements, are misled by unrelated changes made simultaneous to the bug fix. In an environment where automatic discovery of bug origins was of some importance, it would help if developers submitted bug fixes and other changes in distinct commits.
- More ambitiously, precision may be improved by exploring techniques which could distinguish changes that had semantic impact and disregarded syntactic only change (part of the motivation for the dependence approach).
- A further refinement is to ignore bug fixes that involve multiple files, which causes problems for both approaches. However, this would mean disregarding a well-defined category of bugs potentially skewing further bug-related analyses.
- Finally, there is very good potential to improve the approaches by involving users in the process. Relatively easily, users could quickly eliminate changes in the bug fix files that are clearly not associated with the fix. Similarly, as the search for the origin proceeds the user could interactively exclude versions that are clearly not the source of the bug.

To conclude, there is scope for further research in this area prior to experimenting with fully-functional implementations of the approaches. The refinements identified could be manually explored following the research procedure adopted in this study, although experience tells us that this is a time-consuming activity and any larger-scale analysi is best supported by some form of implementation. If the dependence approach is to be pursued then further work is required to methodically explore the range of possible program and system dependence graphs (although the initial findings here suggest that this is best investigated incrementally, rather than starting with a full-blown SDG implementation) and the precise definition of change to data and control dependences and their potential impact. Finally, if these refinements are to be pursued, then the case studies used here — Eclipse, Rachota and jEdit — are useful starting points but there would also be value in exploring other systems and their bugs to ensure consistency of findings, and to aid the longer term understanding of the bug life cycle.

## ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/P505747/1]. We would like to thank the anonymous reviewers for their constructive comments.

## REFERENCES

1. Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models : A Study of Developer Work Habits. In *Proc. ICSE*, 2006. doi:10.1145/1134285.1134355.



DOI: 10.1002/smr

2. Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. MSR*, 2005.
3. Vibha Singhal Sinha, Saurabh Sinha, and Swathi Rao. BUGINNINGS: identifying the origins of a bug. In *Proc. ISEC*, 2010.
4. Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and Balanced?: Bias in Bug-Fix Datasets. In *Proc. ESEC/FSE*, 2009.
5. Christopher S. Corley, Nicholas A. Kraft, Letha H. Etzkorn, and Stacy K. Lukins. Recovering traceability links between source code and fixed bugs via patch analysis. In *Proc. TEFSE*, 2011.
6. Michael Fischer, Martin Pinzger, and Harald C. Gall. Populating a Release History Database from version control and bug tracking systems. In *Proc. ICSM*, 2003. doi:10.1109/ICSM.2003.1235403.
7. Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead Jr. Automatic Identification of Bug-Introducing Changes. In *Proc. ASE*, 2006.
8. Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 1984.
9. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.
10. Loren Larsen and Mary Jean Harrold. Slicing Object-Oriented Software. In *Proc. ICSE*, 1996.
11. Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proc. ICSM*, 1998.
12. Gyula Kovács, Ferenc Magyar, and Tibor Gyimóthy. Static slicing of java programs. Technical Report December, 1996.
13. Jianjun Zhao. Applying program dependence analysis to Java software. In *Proceedings of Workshop on Software Engineering and Database*, 1998.
14. Neil Walkinshaw, Marc Roper, and Murray Wood. The Java system dependence graph. In *Proc. SCAM*, 2003.
15. Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1), December 2006. doi:10.1007/s10515-006-0002-0.
16. [online]Available from: http://www.eclipse.org/jdt/.
17. [online]Available from: http://wala.sourceforge.net/wiki/index.php/Main_Page.
18. [online]Available from: http://code.google.com/p/crystalsaf/.
19. Zhi Da Luo, Linda Hillis, Raja Das, and Yao Qi. Effective Static Analysis to Find Concurrency Bugs in Java. In *Proc. SCAM*, September 2010. doi:10.1109/SCAM.2010.20.
20. Chadd C. Williams and Jaime Spacco. SZZ revisited: verifying when changes induce fixes. In *Proc. DEFECTS*, 2008.
21. [online]Available from: http://www.incava.org/projects/1042574828.
22. Yungbum Jung, Hakjoo Oh, and Kwangkeun Yi. Identifying static analysis techniques for finding non-fix hunks in fix revisions. In *Proc. DSMM*, 2009. doi:10.1145/1651309.1651313.
23. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, 28(2), 2002.
24. Cyrille Artho. Iterative delta debugging. *LNCS*, 5394, March 2009. doi:10.1007/978-3-642-01702-5_13.
25. Albert Endres. An analysis of errors and their causes in system programs. *ACM SIGPLAN Notices*, 10(6), June 1975. doi:10.1145/390016.808455.
26. Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *CACM*, 27(1), January 1984. doi:10.1145/69605.2085.
27. Mark Sullivan and Ram Chillarege. A comparison of software defects in database management systems and operating systems. In *FTCS-22. Digest of Papers.*, 1992. doi:10.1109/FTCS.1992.243586.
28. Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proc. SOSP*, volume 35, October 2001. doi:10.1145/502034.502042.
29. Thomas Zimmermann and Andreas Zeller. HATARI: Raising Risk Awareness. In *Proc. ESEC/FSE*, 2005. doi:10.1145/1095430.1081725.
30. Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting Faults from Cached History. In *Proc. ICSE*, May 2007. doi:10.1109/ICSE.2007.66.
31. [online]Available from: http://www.eclipse.org.
32. [online]Available from: http://rachota.sourceforge.net.
33. [online]Available from: http://www.nongnu.org/cvs/.
34. Gary Boetticher, Tim Menzies, and Thomas Ostrand. PROMISE Repository of empirical software engineering data, 2007. Available from: http://promisedata.org/?p=17.
35. [online]Available from: http://flossmetrics.org/.
36. [online]Available from: http://jedit.org/.
37. Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *JSME*, November 2011. doi:10.1002/smr.567.
38. [online]Available from: http://www.bugzilla.org.
39. [online]Available from: http://www.viewvc.org/.
40. Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald C. Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE TSE*, 33(11), November 2007. doi:10.1109/TSE.2007.70731.




DOI: 10.1002/smr

41. Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE TSE*, 31(6), June 2005. doi:10.1109/TSE.2005.72.



DOI: 10.1002/smr