A preliminary evaluation of text-based and dependency-based techniques 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*

***Abstract*—A crucial step in understanding the life cycle of software bugs is identifying their origin. Unfortunately this information is not usually recorded and recovering it at a later date is challenging. Recently two approaches have been developed that attempt to solve this problem: the text approach and the dependency approach. However only limited evaluation has been carried out on their effectiveness so far, partially due to the lack of data sets linking bugs to their introduction. Producing such data sets is both time-consuming and challenging due to the subjective nature of the problem. To improve this, the origins of 166 bugs in two open-source projects were manually identified. These were then compared to a simulation of the approaches. The results show that both approaches were partially successful across a variety of different types of bugs. They achieved a precision of 29%–79% and a recall of 40%–70%, and could perform better when combined. However there remain a number of challenges to overcome in future development — large commits, unrelated changes and large numbers of versions between the origin and the fix all reduce their effectiveness.**

***Keywords*-software maintenance, bug-introducing changes, mining software repositories**

I. Introduction

Previous studies have suggested that software developers can spend nearly half their time fixing bugs [1], and the impact of bugs on cost and effort is a well-documented problem faced by the entire industry. Many techniques and tools have been developed which attempt to prevent bugs from being introduced in the first place by producing warnings based on evaluating some aspect of the code. Often this is done by inferring common patterns from code which contains bugs and then finding similar code. Many of these tools solely examine the current version of code, using the number of bugs that have been identified in the past.

More sophisticated techniques may mine the history of the code. However, accurately identifying which changes originally caused a bug is 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.

Identifying which changes actually introduced a bug could be useful in improving the accuracy of any of these bug prevention tools or allowing the introduction of more advanced techniques altogether. Amongst others, possibilities such as measuring the lifetime of bugs and predicting whether a change is likely to introduce a bug have been identifed by Kim et al. [2].

Software developers and managers would also stand to benefit from identifying when their bugs were introduced. The data could be used, for example, to highlight periods of time when more bugs were introduced than normal or to assess how individual developers, departments or projects differ in their ability to detect and correct bugs. This could then be used to pinpoint weaknesses in their processes. Identifying such changes also opens possibilities for researchers to more closely examine the kinds of changes that can introduce bugs.

In recent years two approaches have been developed that attempt to discover the origin of a bug by examining the changes made to fix it. Both approaches start from the version of code that fixed the bug and progressively examine preceding versions until they find the version that introduced the bug. They differ in how they examine the changes between versions: the text approach [3] uses only the change in the text itself while the dependency approach [4] uses changes in relationships between lines of code.

Unfortunately implementations of the approaches are not yet readily available. The dependency approach in particular is complex, with a large number of implementation challenges. The text approach, while conceptually simpler, is still non-trivial. Before deciding whether to invest the large amount of time required to implement either approach it is reasonable to ask how accurate they would be, especially given their differences. To evaluate this, this work has manually identified the origin of a number of bugs and compared them to the results given by a simulation of the approaches.

II. Identifying Bug Origins

Two components of modern development are crucial to help identify the origin of a bug: software configuration

```
...
int shift = isCarbon ? -25 : -10;
light = display.getSystemColor(SWT.BACKGROUND);
dark = new Color(display, Math.max(0, light.getRed() + shift),
                          Math.max(0, light.getGreen() + shift),
                          Math.max(0, light.getBlue() + shift));
textColor = display.getSystemColor(SWT.FOREGROUND);
...
```

(a) v1.39

```
     ...
1.17 int shift = isCarbon ? -25 : -10;
1.7  light = display.getSystemColor(SWT.BACKGROUND);
1.8  dark = new Color(display, light.getRed() + shift,
1.8                            light.getGreen() + shift,
1.8                            light.getBlue() + shift);
1.24 textColor = display.getSystemColor(SWT.FOREGROUND);
     ...
```

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

```
...
boolean carbon = "carbon".equals(SWT.getPlatform());
int shift = carbon ? -25 : -10;
light = display.getSystemColor(SWT.BACKGROUND);
dark = new Color(display, light.getRed() + shift,
                          light.getGreen() + shift,
                          light.getBlue() + shift);
taskColor = display.getSystemColor(SWT.FOREGROUND);
...
```

(c) v1.8

```
...
boolean carbon = "carbon".equals(SWT.getPlatform());
light = display.getSystemColor(SWT.BACKGROUND);
if (carbon)
 dark = new Color(display, 230, 230, 230);
else
 dark = new Color(display, 245, 245, 245);
taskColor = new Color(display, 120, 120, 120);
...
```

(d) v1.7

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

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 [3]. When fixing issues, developers often include the issue ID in their commit comment. Issues can therefore be linked to the commits that fix them by extracting the comments from the SCM repository and searching for issue IDs. Whilst other techniques for this task have been proposed [5], they have not been applied in this work.

A. Text Approach

Fig. 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. [3], 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

```
  void applyResult(DecorationResult result){
    ...
1:  ImageDescriptor [] resultDescriptors = result.getDescriptors();
3:  for(int i = 0; i < descriptors.length; i ++){
4:    if(resultDescriptors[i] != null)
5:      descriptors[i] = resultDescriptors[i];
    }
    ...
  }
```

(a) v1.6

```
  void applyResult(DecorationResult result){
    ...
1:  ImageDescriptor [] resultDescriptors = result.getDescriptors();
2:  if(resultDescriptors != null){
3:    for(int i = 0; i < descriptors.length; i ++){
4:      if(resultDescriptors[i] != null)
5:        descriptors[i] = resultDescriptors[i];
      }
    }
    ...
  }
```

(b) v1.7

Figure 2. Eclipse Bug 66653 - `DecorationBuilder.java`

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 VI. 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 [2].

Unfortunately the text approach is not suitable for all bugs. Fig. 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.

B. Dependency Approach

The dependency approach [4] attempts to address some of the text approach's shortcomings by examining changes in the behaviour of the code rather than simply the text. A graph of a method's control and data dependencies is built, known as a program dependence graph (PDG). Informally, a line of code X is data dependent on another line Y if Y could potentially have been the last to set the value of a variable used by X. A line X is control dependent on another line Y if X may or may not be executed dependent on the result of Y. Lines of code are represented by nodes in the PDG while the dependencies are shown as two different types of edge.

To determine the origin of the bug in Fig. 2 the dependency approach compares the PDG for the fixed version to the PDG for the previous version. The approach first identifies any removed dependencies. Added dependencies are only examined if no dependencies were removed.

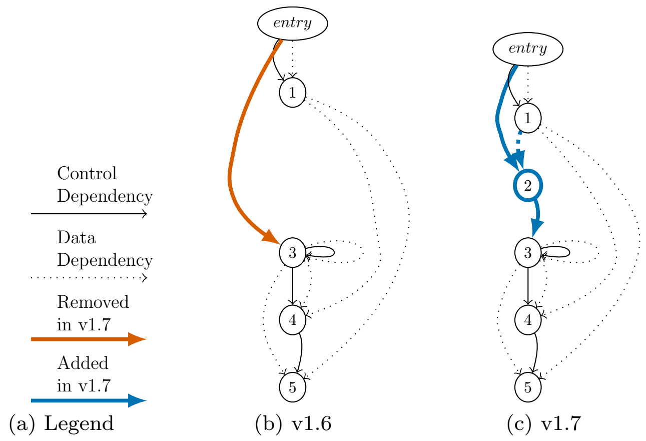

Figure 3. PDGs for `DecorationBuilder.java`

As shown in Fig. 3, the control dependency of line 3 on method entry has been removed and the line now has a control dependency on the new `if` statement. The approach therefore builds the PDG for each preceding version until it finds when this dependency was added. In this case the dependency, and the bug, was introduced in version 1.5 when the method was created. If there are multiple dependencies removed in the fix the dependency approach returns the most recent version in which one of these dependencies was introduced.

Fig. 3 also shows that a new line 2 has been introduced, with two new corresponding control dependencies and a new data dependency from line 1 to line 2. As discussed, the approach prioritises removed dependencies but if no dependencies 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 dependencies.

There are two possible variations on the approach. The *intraprocedural* approach, described above, only considers dependencies within a single method. The *interprocedural* approach analyses dependencies between methods, which may give better results but is more complex and challenging to build. The interprocedural approach was not used in this study due to its complexity.

The dependency approach is not appropriate for all bugs. Using the intraprocedural approach the bug fix in Fig. 1 would not result in any change to the method's dependencies. The approach could not therefore identify the origin of the bug.

III. Evaluation

As stated previously, limited evaluation has so far been performed on the approaches. The original text approach [3] did not validate whether the commits identified by the approaches actually caused the bugs in question. Subsequent work has focused on evaluating updated approaches relative to the original, by examining which results have been added or removed. In particular, commits which introduced bugs but which are not returned by the approaches have not been previously assessed. 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. Therefore this section will evaluate a manual *simulation* of the approaches. This allows their expected effectiveness to be assessed prior to any implementation.

A. Subject

The systems under examination are *Eclipse*[1], an IDE and development platform, and *Rachota*[2], a time track-

[1]http://www.eclipse.org
[2]http://rachota.sourceforge.net

ing application. These were selected as they vary considerably in size, maturity and usage, although both are open-source, written in Java and use *CVS*. Bug and commit data for Eclipse was obtained from *Promise* [6], and contains a collection of fixes, each of which is a commit linked to a bug as described in Section II. 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 *FLOSS-Metrics*[3]. 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 fixing commits were identified for each of the 66 bugs in Rachota. Only the 130 commits involving Java files will be considered in this evaluation, but problems related to other files are explored in Section IV-H.

B. Origin Classification

Before assessing the approaches the origins of each of the bugs had to be manually identified — a substantial undertaking. For each commit in the sample the changes that fixed the bug were examined[4]. Previous versions were then examined to determine which version the bug first appeared in. If code was moved between methods, or extracted to new methods, the origin was traced through the original methods, as long as the bug could still be said to exist in the older version. Additionally, information provided in the bug and commit comments was used to help locate the origin.

If the bug was introduced by a change to an earlier version of the file, the fixing commit was then classified as either *Single* or *Multiple*, depending on the number of instances of the bug in the file. For example, Eclipse Bug 62932 was a `NullPointerException` that occurred when elements containing a particular type of breakpoint were renamed. However, the bug occurred when any of a project, package or type was renamed or moved. Each of these followed a separate route through the code, and so each is considered a separate instance of the bug, with a separate origin. The commit is therefore classed as *Multiple*. Note that commits can be classed as *Multiple* whenever there is more than one instance of a bug even if each was introduced in the same version.

[3]http://flossmetrics.org/

[4]Full results are at http://personal.cis.strath.ac.uk/~spd/

Not all bugs had origins in the same file. When no versions of the file could be identified as causing the bug then the fixing commit was classed as one of: *Elsewhere* if the bug was introduced by a change to another file; *Unrelated* if the commit is not actually related to the bug in question; or *Related* if there was no bug in the file but it was still involved in fixing the bug.

Table I shows the number of commits for each classification. For some commits the origin could not be determined. These 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.

C. Approach Classification

To evaluate the text approach each commit was again analysed. Each line of code removed or updated was manually examined in the preceding version, using the *ViewVC* repository browser. Each of the versions identified by the annotated view was also examined in order to ensure that the version reported was correct. Each line in a fix could lead back to a different version; the full set of versions found was recorded.

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; the code was traced back until the last actual change was made. Adding or removing comment markers around lines of codes (as happened in Eclipse Bug 60768) 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. Such techniques could easily be automated [2].

To evaluate the dependency approach, the old and new versions of each updated method were manually compared and any dependencies removed were noted. Each preceding version of the method was examined until the most recent version to add one of the dependencies was found. If a fix involved multiple methods this was done for each and the most recent version to have altered any of them was recorded as the single result.

If no dependencies were removed in any method then any added dependencies were noted. Each preceding

Table I
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

version of the method was then examined in a similar manner, searching for the most recent version to alter either the source or target of the dependency.

The graph comparison of the dependency approach requires lines to be mapped between versions. This mapping was based on the similarity of the two lines and the lines surrounding them. Generally this was straightforward but any ambiguities were recorded and re-examined at the end in order to ensure that similar cases were treated in the same manner. Specifically, changes to an object's type or the addition or removal of a method in a chain of method calls were regarded as the two lines not mapping to one another. Changes that were made to the condition of an `if` statement were regarded as the lines mapping to one another.

Only dependencies on other lines within the same method were considered; dependencies on other methods and on any fields were ignored. In addition if the name of a method was altered during the fix this was treated as one method being added and another removed; no attempt was made to trace the dependencies. These behaviours, and the line mapping technique, attempt to match the original description [4] as closely as possible, but are explored further in Section IV.

As the dependency approach works by first mapping different versions of a method to one another based on their signature, it was assumed that it could handle methods being relocated inside a class, as happened with Eclipse Bug 81695. Because the text approach implementation relies on tools based on line numbers, it was assumed that it could not cope with this.

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.

D. *Results*

Table II shows the results obtained by the text approach (TA) and dependency 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 dependency 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 dependency approach only returns a single origin. As to be expected from the lower number of false positives, the precision of the dependency approach is higher, although the recall is lower.

Table II
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

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.

IV. Analysis

A. *Successful Results*

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, around a third resulted in exceptions that either crashed the application, displayed 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; their effectiveness appeared to depend more on the type of changes being made and on changes made at the same time or in intervening versions. This is explored in more depth in Section IV-E.

In general the approaches performed best when there were fewer versions between the origin and the fix, but there were exceptions. Eclipse Bug 49561 involved certain operations locking the entire workspace and required changes to numerous files to fix. One of the fixes was version 1.156 of `CompilationUnit.java`, which had multiple instances of the bug. The text approach correctly identified the origins of this bug even though one was introduced in version 1.98, 58 versions before, and the other was 155 versions earlier in the initial version.

There were a number of origins only identified by the text approach. Often these included changes to literal values or to formulas. For example, Bug 64531, where Eclipse would hang utilising all of the machine's CPU, was solved by changing a single line from `findReplacePosition = selection.x;` to `findReplacePosition = selection.x - 1;`. This had no effect on dependencies but was correctly identified by the text approach. Changes sometimes also altered lines more significantly without altering dependencies. Eclipse Bug 51593 was an exception thrown in the background during particular operations. The fix was to add a check on the length of an array, but to an `if` statement that was already checking whether the array was null. Therefore the dependencies were unaltered although the text approach could correctly identify the origin.

Conversely the dependency approach returned some origins the text approach could not, particularly those with added lines. Eclipse Bug 60246 was an exception caused by trying to create two classes with the same name in different cases. The fix involved adding an `else` block to a series of `if-else` statements. The dependency approach correctly traced this back to when the statements were added but the text approach could not identify any changes. The dependency approach also appeared to better handle unrelated changes in the intervening versions, such as in Eclipse Bug 63519. This was fixed by removing a check on a variable in an `if` statement but as the method had been moved in a previous version the text approach returned that version. The dependency approach is specifically designed to take this into account and so correctly returned the origin.

B. Type of Origin

Table III shows a more detailed view of how the various types of origin affect the the two approaches. As shown, the text approach produced more false positives than the dependency approach across every type of origin. However, as expected, it correctly identified the origins classed as *Multiple* more often than the dependency approach.

The results also show the existence of false positives for commits classed as *Elsewhere*. These bugs have no origin within the file that was fixed, and as both approaches look solely in that file it is unclear whether the origins of these bugs could ever be discovered by either. It *may* be possible for an interprocedural version of the dependency approach to identify the origin, but to compare the dependencies of an entire project is a daunting task. Other bugs could potentially be introduced by changes to the external environment or to libraries, and these origins may be impossible to discover.

The results also show that a significant proportion of false positives occurred for commits with no origin: those classed as *Related* or *Unrelated*. In these cases, any result returned was a false positive; ideally the approaches

Table III
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

should return nothing at all. It is not clear that either approach could ever be updated to identify such files and ignore them. Separating the files that were changed to fix a bug from the files that were changed as a consequence of the fix may not be possible without knowing the developers' intentions.

C. False Negatives

One obvious weakness is the 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 only involved lines being added. For the dependency approach the equivalent figure was 58 commits, where the fix usually contained no changed dependencies.

The original authors proposed that in the cases where the dependency approach found no altered dependencies it would fall back to using the text approach [4]. 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 dependency approach. The original paper reported the dependency 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 IV 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 dependency 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

Table IV
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

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.

For the dependency approach, another way to reduce the number of cases where nothing is returned would be to assess dependencies on external objects. For example, the fix for Eclipse Bug 63753 changed which static constant a line of code depended on. The dependency approach therefore saw these dependencies as unchanged and returned nothing. Adding these dependencies in, or implementing the full interprocedural dependency approach, would have allowed this origin to be correctly identified. 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.

D. 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 V. In addition, the bold 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 dependency 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 VI 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.

E. Unrelated Changes

False positives were often introduced when unrelated changes were made coincident to the fix. For example, version 1.32 of `AntModel.java`, part of the fix for Eclipse Bug 52040, was a straightforward update that both approaches could have handled correctly. However Bug 51347 was also fixed at the same time, altering many other lines. The approaches could not distinguish which lines were relevant to which bug and so returned the incorrect answer. In total there were 34 fixes containing unrelated changes, of which 20 fixed multiple 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.

Similarly both approaches had problems identifying the correct origin when unrelated changes were made between the origin of the bug and the fix. Eclipse Bug 49891 was a simple fix to avoid a `NullPointerException`. However during one previous release the return type of the method had changed and a cast had been altered on the line involved. Both approaches therefore returned this version as the source of the bug. Unfortunately there is not in general an easy way to adapt for such cases, as the same scenario of a cast being changed could conceivably be the source of a bug.

One situation where this could be improved is where multiple commits are made to fix a bug, often because the first attempt was incorrect. For example three versions of `LaunchView.java` were linked to Eclipse Bug 61928. When evaluating the later versions the earlier fix

Table V
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**

Table VI
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

attempts were incorrectly returned as the origin of the bug. While in some senses this could be considered the origin of *a* bug, as the fix was buggy, it was not the origin of the bug being examined. 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 [2].

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. Fig. 4 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 II.

Fig. 4 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.

F. *Large Commits*

As bug fixes increased in size the approaches became less likely to identify the correct origins. Fixing Eclipse Bug 61706 required changing commonly-used constructors and methods resulting in changes to 24 files. Only 4 of these directly related to the bug; the remainder were classed as *Related* or *Unrelated* based on whether they were involved in the bug. Both cases led to a number of false positives. This distribution was repeated on other large bugs and one proposal is to ignore fixes that change a large number of files [2]. The number of bugs of each size in Eclipse is shown in Table VII.

Table VII
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

Fig. 5 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 significantly better for smaller commits. For the few larger bugs the scores decrease. The same was not necessarily true of Rachota however; 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.

G. *Line Mapping*

The original dependency approach stated that identifying which line in one PDG was the same as which line in the previous PDG could be performed by any of several techniques. However, it became apparent throughout this work that how this mapping is done could have a significant impact on the results gained. Eclipse Bug 51593, detailed earlier, involved adding a new predicate to an existing `if` statement, but this predicate depended on a variable already referenced in the `if` statement. The

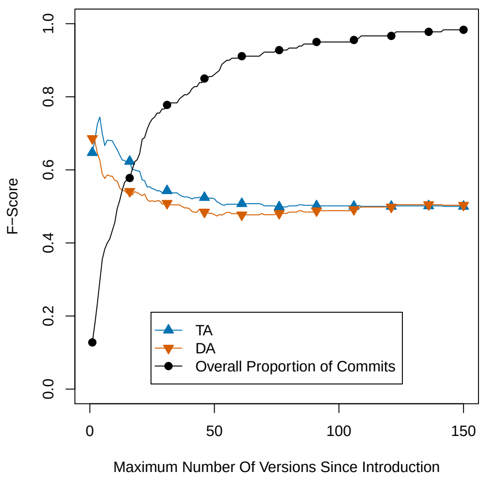

Figure 4. Age of Origin (Eclipse)

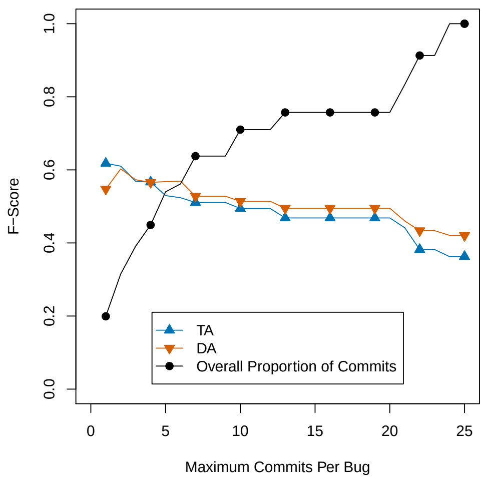

Figure 5. Performance of Different Sized Fixes (Eclipse)

dependency approach regards this as being the same line of code in both versions with no change in dependencies. As such, it returned nothing for this instance. If however the mapping algorithm used had said these lines were different then an entire node would have been removed from the PDG and a new one added. The removal of the old node and its corresponding dependencies would have caused the approach to behave completely differently.

The approach would also behave differently if the PDGs were built from individual statements, or clauses, rather than lines. Another aspect to take into consideration is that the approach only examines added dependencies if there were no removed dependencies, which may not always be appropriate. Whether altering any of these parameters would result in the approach becoming more or less effective has not been evaluated.

H. *Other Issues*

A number of other issues were also identified.

- One Rachota bug was fixed in a plain text file, which the text approach could have identified. However the majority of changes to text files were not related to bugs. Including text files would have introduced 83 false positives but only 4 true positives.
- Although Eclipse issues marked as *enhancement* were excluded, a number of issues remained that could still be seen as enhancements. In general, both approaches performed poorly on these. Their effectiveness may vary if issues are classified more accurately, possibly automatically [7].
- Two Eclipse bugs were specifically related to errors in comments. The text approach could have correctly identified the origin if changes to comments were taken into account but this would have created false positives elsewhere. Some projects may wish to identify the origin of such bugs, or of formatting errors and coding standard violations. This creates a trade-off between accepting false positives and correctly handling such bugs.

V. Threats to Validity

The evaluation has several potential shortcomings, of which the most major is the possibility of error. In particular the dependency approach could be difficult to visualise, especially regarding constructs such as fields, `synchronized` blocks or `try-catch` blocks, as the original paper did not discuss how these were handled. To maintain consistency related scenarios were noted as they were encountered and then revisited as a group.

There may also have been errors when determining the correct origin for each bug, especially as the evaluation was not carried out by a developer of the projects being studied. While this risk could not be avoided altogether, related or similar commits were noted and then revisited at the end to ensure consistency. Where possible commit logs and bug report comments were also used to give an indication of the origin.

The work only ever looked to find an origin in the file where the fix was applied. However as discussed the origin to a bug sometimes lay elsewhere. The evaluation technique used is partially due to the underlying SCM, CVS, which considers each file in a commit as a separate transaction. One possibility may be to repeat this evaluation for projects that use other SCMs which assign version numbers to the entire project, or to use techniques that reconstruct project versions [8].

A particular problem is that, due to the time-consuming nature of the work, only a small sample was studied and the results may not generalise to other systems. There may have been particular factors that influenced the results. As only bugs that could be linked to a fixing commit were examined they may in fact not even be representative of all bugs in the two systems [9]. Future studies would be needed to determine if these results are applicable to other projects. This work however can be used as a data set for better evaluation of the actual implemented techniques, and as a framework for future studies.

VI. Related Work

Several improvements have been suggested to the approaches. Williams and Spacco [10] built on the text approach using a Java-syntax aware differencing tool, *DiffJ*, to eliminate changes that have no semantic effect and developed the concept of line mapping [2] further. Jung et al. [11] 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. The approaches also bear some resemblance to the concept of iterative delta debugging [12], where failing test cases from a later version of code are used repeatedly on earlier versions in order to identify the last version where the test passed.

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 [13]–[15], although they did not seek to find the originating version of code. Chou et al. [16] examined errors in versions of Linux, finding the median difference between the origin and fix of a bug to be around 1.25 years.

A number of practical tools have been built on top of the text approach. HATARI [17] determines how risky an area of source code is based on the proportion of changes made to that area which introduced a bug. Similarly FixCache [18] uses a cache of elements that recently introduced a bug to predict how likely changing an area is to cause another bug.

VII. Conclusions

This paper has manually classified the origins of 166 bugs and compared the results to two approaches for identifying the origins of fixed bugs: the text approach and dependency approach. It has evaluated these approaches and found that they individually achieved a precision of 29%–79% and a recall of 40%–70%, suggesting they are at least partially successful.

The current precision and recall seems reasonable for some applications. Which system is more appropriate to implement is most likely a product of the time available for implementation and the willingness to accept false positives. Some amount of false positives may be acceptable if the approaches were to be used by developers and managers tracking their own bugs. Some of the variations described earlier could be built into the system and left to the user to choose whether to adopt or not.

If however the approaches were to be used for researching the life cycle of bugs they may not be effective enough. While the variations discussed above could improve either precision or recall, they may do so at the cost of the other. Additionally some variations would result in certain bugs being ignored, for example those which changed a large number of files, and this may introduce bias into the findings. Care would have to be taken before adopting these approaches in research.

The results found also indicated that:

- The accuracy of using both approaches together was at least as good as, and mostly better than, using either on their own
- Both approaches often gave incorrect answers when other changes were made coincident to the fix. How diligently a project separates fixes from other changes may influence the approaches' effectiveness
- Unrelated changes made between the origin and the fix often caused incorrect results
- The approaches became less accurate as more versions passed between the origin and the fix, although most bugs were fixed soon after their origin
- Most bugs required changes to only a few files, but bugs which required many changes could often not be classified correctly

At the very least, this paper proposes that further evaluation of these approaches is merited, particularly with a larger range of projects, in order to see if these findings are generally applicable and whether the proposed variations could improve effectiveness. Developing these approaches further could help significantly with accurately identifying the origin of bugs, and lead to a better understanding of their 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] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models : A Study of Developer Work Habits," in *Proc. ICSE*, 2006, pp. 492–501.

[2] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr., "Automatic Identification of Bug-Introducing Changes," in *Proc. ASE*, 2006, pp. 81–90.

[3] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. MSR*, 2005.

[4] V. S. Sinha, S. Sinha, and S. Rao, "BUGINNINGS: identifying the origins of a bug," in *Proc. ISEC*, 2010, pp. 3–12.

[5] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *Proc. TEFSE*, 2011, pp. 31–37.

[6] G. Boetticher, T. Menzies, and T. Ostrand. (2007) PROMISE Repository of empirical software engineering data. West Virginia University, Department of Computer Science. [Online]. Available: http://promisedata.org/?p=17

[7] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests," in *Proc. CASCON*, 2008.

[8] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, June 2005.

[9] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced?: Bias in Bug-Fix Datasets," in *Proc. ESEC/FSE*, 2009, pp. 121–130.

[10] C. C. Williams and J. Spacco, "SZZ revisited: verifying when changes induce fixes," in *Proc. DEFECTS*, 2008, pp. 32–36.

[11] Y. Jung, H. Oh, and K. Yi, "Identifying static analysis techniques for finding non-fix hunks in fix revisions," in *Proc. DSMM*, 2009, pp. 13—-18.

[12] C. Artho, "Iterative delta debugging," *LNCS*, vol. 5394, pp. 99–113, March 2009.

[13] A. Endres, "An analysis of errors and their causes in system programs," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 327–336, June 1975.

[14] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *CACM*, vol. 27, no. 1, pp. 42–52, January 1984.

[15] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *FTCS-22. Digest of Papers.*, 1992, pp. 475–484.

[16] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proc. SOSP*, vol. 35, no. 5, October 2001, pp. 73—-88.

[17] T. Zimmermann and A. Zeller, "HATARI: Raising Risk Awareness," in *Proc. ESEC/FSE*, 2005, pp. 107–110.

[18] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *Proc. ICSE*, May 2007, pp. 489–498.