Towards an automated approach for bug fix pattern detection

Fernanda Madeiral[1], **Thomas Durieux**[2], **Victor Sobreira**[1], **Marcelo Maia**[1]

[1] Federal University of Uberlândia, Brazil

[2]INRIA & University of Lille, France

fernanda.madeiral@ufu.br, thomas.durieux@inria.fr, {victor, marcelo.maia}@ufu.br

Abstract. *The characterization of bug datasets is essential to support the evaluation of automatic program repair tools. In a previous work, we manually studied almost 400 human-written patches (bug fixes) from the Defects4J dataset and annotated them with properties, such as repair patterns. However, manually finding these patterns in different datasets is tedious and time-consuming. To address this activity, we designed and implemented PPD, a detector of repair patterns in patches, which performs source code change analysis at abstract-syntax tree level. In this paper, we report on PPD and its evaluation on Defects4J, where we compare the results from the automated detection with the results from the previous manual analysis. We found that PPD has overall precision of 91% and overall recall of 92%, and we conclude that PPD has the potential to detect as many repair patterns as human manual analysis.*

1. Introduction

Automatic program repair is a recent research field where approaches have been proposed to fix software bugs automatically, without human intervention [Monperrus 2018]. In automatic program repair, empirical evaluation is conducted by running repair tools on *known bugs* to measure the repairability potential. These known bugs are available on bug datasets, e.g., Defects4J [Just et al. 2014].

Building datasets of bugs is a challenging task. Despite the effort made by authors of bug datasets, such datasets generally do not include detailed information on the bugs and their patches (bug fixes), so a fair and advanced evaluation of repair tools becomes harder. We highlight two tasks that make the evaluation of repair tools more robust:

- *Selection of bugs* is used to filter out bugs that do not belong to the bug class which the repair tool under evaluation targets. For instance, NPEFix [Durieux et al. 2017] is a tool specialized in null pointer exception fixes, and it will probably fail on bugs that were not fixed by a human with a null pointer checking. So, a coherent and fair analysis would include only bugs within the target bug class(es) of the respective tools.
- *Correlation analysis* is an advanced analysis of the results produced by a repair tool, making possible to derive conclusions such as "the repair tool performs well on bugs having the property X". This kind of analysis requires a characterization of all bugs available in the used dataset.

To support these two tasks, in a previous work [Sobreira et al. 2018], we *manually* analyzed the patches of the Defects4J dataset [Just et al. 2014], which is a widely used dataset in automatic program repair field. As a result, we delivered a taxonomy of properties and the annotation of Defects4J patches according to such taxonomy. Despite the value of manual work, analyzing patches to calculate or to find properties when characterizing patches from different datasets is tedious and time-consuming. Nevertheless, the already built taxonomy is a useful resource to guide the automation of patch analysis.

In this paper, we present PPD (Patch Pattern Detector), a detector of *repair patterns* in patches, one of the existing types of property in our previous taxonomy. Repair patterns are recurring abstract structures in patches. For instance, a patch that affects only one line in the buggy code is an instance of the pattern *Single Line*, while a patch that adds an entire conditional block is an instance of the pattern *Conditional Block Addition*.

PPD analyzes a given patch by first retrieving edit scripts at Abstract-Syntax Tree (AST) level from the *diff* between the buggy and patched code using GumTree [Falleri et al. 2014]. Then, PPD searches for instances of patterns by analyzing the AST nodes of the *diff* using Spoon [Pawlak et al. 2015], a peer-reviewed library to analyze Java source code. We evaluated PPD and found that it has the potential of detecting the nine repair pattern groups from Sobreira et al. (2018). PPD can support automatic repair researchers on selecting bugs from bug datasets and performing correlation analysis between repaired bugs and their properties. Moreover, PPD can be useful in comparisons between different datasets of bugs. By discovering the constitution of bug datasets, it is possible to study the balance between them and also the flaws.

To sum up, the main contributions of this paper are 1) a tool to detect repair patterns from patches written in Java, which is publicly available, and 2) the detection of new 90 instances of the patterns on Defects4J.

2. Taxonomy of Repair Patterns

Previously, we delivered a taxonomy of repair patterns containing nine groups and 25 patterns in total [Sobreira et al. 2018]. PPD implements the detection of all these patterns. In this section, we briefly define the nine pattern groups. Additionally, we refer in this paper to patches from Defects4J (using a simple notation with project name followed by bug id) to provide examples with external links for patch visualization, e.g., Chart-1[1].

Conditional Block involves the addition or removal of conditional blocks (e.g., Lang-45).

Expression Fix involves actions on logic (Chart-1) or arithmetic (Math-80) expressions.

Wraps/Unwraps consists of (un)wrapping existing code with/from high-level structures such as try-catch blocks (Closure-83) and low-level ones such as method calls (Chart-10).

Single Line is dedicated to patches affecting one single line or statement (Closure-55).

Wrong Reference occurs when the code references a wrong variable (e.g., Chart-11) or method call (e.g., Closure-10) instead of another one.

Missing Null-Check is related to the addition of a conditional expression or the expansion of an existing one with a null-check that was missing in the code (e.g., Chart-15).

Copy/Paste is the application of the same change to different points in the code (Chart-19).

Constant Change involves changes in literals or constant variables (e.g., Closure-65).

Code Moving involves moving code statements or statement blocks around, without extra changes to these statements (e.g., Closure-117).

[1]For paper printed version: all links on Defects4J patches can be built by inserting two parameters in `http://program-repair.org/defects4j-dissection/#!/bug/⟨project_name⟩/⟨bug_id⟩`. Example: Chart-1 contains the link `http://program-repair.org/defects4j-dissection/#!/bug/Chart/1`

3. PPD: a Detector of Bug Fix Patterns

The detection of repair patterns in patches falls in source code change analysis task. Analyzing source code changes can be performed at different levels of granularity such as file level, line level, and AST level. Our approach is at the AST level, and it consists of two main tasks to detect repair patterns in a given patch:

Retrieval of the AST diff: Given as input the buggy version of the program and the patch file (*diff* file), PPD retrieves the AST *diff* between the buggy and patched code (also known as *edit scripts*) using the GumTree algorithm [Falleri et al. 2014]. There are different implementations of the GumTree algorithm: we use GumTree Spoon[2] since such tool delivers the AST *diff* nodes in the representation of the Spoon library [Pawlak et al. 2015], based on a well-designed meta-model for representing Java programs. Therefore, we can analyze the edit scripts returned by GumTree with Spoon.

Analysis of the AST diff: With the AST *diff* retrieved, the AST nodes are analyzed to detect the repair patterns. PPD contains a set of detectors, one for each pattern group, because each pattern group has its own definition, which lead us to define a specific strategy for the detection of each of them[3]; the strategies are mainly based on searching and checking code elements/structures in the AST *diff*. However, all detectors follow the same general process: it analyzes edit scripts using Spoon based on the defined strategy to detect the pattern it was designed for. Thus, we choose one pattern, *Missing Null-Check*, to be used as an example to describe in details how PPD performs automatic detection. Given the edit scripts from a patch, the strategy of the *Missing Null-Check* detector is the following:

1. It searches for the addition of a binary operator where one of the two elements is `null`, i.e., a null-check;
2. It extracts from the null-check the variable being checked (`variable <operator> null`) or the variable being used to call a method where its return is being checked (`variable.methodCall() <operator> null`);
3. It verifies if the extracted variable is new, i.e., was added in the patch: a) if the variable is not new, a missing null-check was found; b) if the variable is new, it verifies if the new null-check wraps existing code: if it does, a missing null-check was found.

Consider the *diff* in Listing 1. In the buggy version of this code, in the old line 2166, a null pointer exception had been thrown when the variable `markers` was null and accessed for a method call. In the fixed version, a conditional block was added to check whether `markers` is null, and in such case, the method returns, so the program execution does not reach the point of the exception. The added null-check is an instance of the pattern *Missing Null-Check*. Note that the null-check was added in a new conditional block, so this patch also contains an instance of the pattern *Conditional Block Addition with Return Statement*. Additionally, this conditional block was added in four different locations on the code (see Chart-14), which consists in the *Copy/Paste* pattern.

```
2166       + if (markers == null) {
2167       +     return false;
2168       + }
2166  2169   boolean removed = markers.remove(marker);
```

Listing 1. Patch for bug Chart-14.

[2]`https://github.com/SpoonLabs/gumtree-spoon-ast-diff`

[3]PPD was designed in a modularized way that makes possible the addition of a new pattern detection by extending an existing class and implementing a strategy for the detection of the new pattern.

A missing null-check can appear in different variants beyond the addition of an entire conditional block. In Listing 2, for instance, the missing null-check was added in a new conditional by wrapping an existing block of code. This type of change consists of the pattern *Wraps-with if*. Different from *Conditional Block Addition*, the body of the conditional contains existing code in *Wraps-with if*.

```
  ChartRenderingInfo owner = plotState.getOwner();
+ if (owner != null) {
     EntityCollection entities = owner.getEntityCollection();
     if (entities != null) {
        entities.add(new AxisLabelEntity(this, hotspot,
                this.labelToolTip, this.labelURL));
     }
+ }
```

Listing 2. Patch for bug Chart-26.

Since our detector searches for binary operators involving null-check, it also detects missing null-checks in other structures beyond `if` conditionals. In Listing 3, for instance, there is an example of a conditional using the ternary operator. When the ternary operator is used, and an existing expression is placed in the `then` or `else` expression, we have the pattern *Wraps-with if-else*. Note that this patch is also an instance of the pattern *Single Line* since only one line was affected by the patch.

```
- description.appendText(wanted.toString());
+ description.appendText(wanted == null ? "null" : wanted.toString());
```

Listing 3. Patch for bug Mockito-29.

For these three example patches, PPD was able to detect all the existing patterns in them, according to the taxonomy presented in Section 2.

4. Evaluation

Method. Our evaluation consists of running PPD on real patches to measure its ability at detecting the 25 repair patterns.

Subject Dataset. The patches used as input to PPD are from Defects4J [Just et al. 2014], which consists of 395 patches from six real-world projects (e.g. Apache Commons Lang and Mockito Testing Framework). We chose this dataset since it contains real bugs and all its patches have been annotated with repair patterns [Sobreira et al. 2018], allowing the direct comparison between results generated by PPD and the previous manual detection.

Result analysis. We analyzed the results in two steps. First, we calculated the precision and recall of the PPD for each pattern, using the available manual detection [Sobreira et al. 2018] as an oracle. We refer to such manual detection as *human detection*, while we refer to the detection produced by PPD as *automatic detection*. Second, we performed manual analysis on the disagreements between the automatic and human detection. For each pattern, two different authors of this paper analyzed all patches where there were disagreements and determined whether PPD actually missed or wrongly detected such pattern. We annotated the disagreements with one of the five diagnostics presented in the first column of Table 2. Then, we calculated the actual precision and recall for each pattern, using the following formulas: $TP = A+B+DC$, $precision = \frac{TP}{TP+DW}$, $recall = \frac{TP}{TP+HC}$, where A is the number of agreements between PPD and human detection, B is the disagreements when both PPD and human detection may be accepted, DC is the disagreements when PPD detection is correct and DW when PPD detection is wrong, and HC is the disagreements when the human detection is correct.

Table 1. PPD performance.

Pattern	Variant	Prior Precision (%)	Prior Recall (%)	Post Precision (%)	Post Recall (%)
Conditional Block	Addition	74.75	93.67	99.00	98.02
	″ with Return Statement	90.12	94.81	100.00	96.47
	″ with Exception Throwing	93.75	90.91	96.88	91.18
	Removal	60.71	77.27	86.67	89.66
Expression Fix	Logic Modification	82.22	75.51	91.11	83.67
	″ Expansion	90.20	95.83	92.16	97.92
	″ Reduction	76.92	83.33	76.92	100.00
	Arithmetic Fix	69.57	50.00	91.67	64.71
Wraps/Unwraps	Wraps-with if	74.19	95.83	83.87	96.30
	″ if-else	81.25	84.78	92.00	90.20
	″ else	16.67	100.00	33.33	100.00
	″ try-catch	100.00	100.00	100.00	100.00
	″ method	78.57	78.57	85.71	85.71
	″ loop	40.00	100.00	60.00	100.00
	Unwraps-from if-else	42.11	61.54	57.89	68.75
	″ try-catch	100.00	100.00	100.00	100.00
	″ method	45.45	83.33	54.55	85.71
Single Line	–	100.00	97.96	100.00	100.00
Wrong Reference	Variable	66.67	76.19	82.35	89.36
	Method	68.42	83.87	86.84	89.19
Missing Null-Check	Positive	95.45	84.00	100.00	100.00
	Negative	96.67	90.63	100.00	96.77
Copy/Paste	–	56.16	85.42	91.78	90.54
Constant Change	–	77.27	89.47	90.91	90.91
Code Moving	–	60.00	85.71	81.82	100.00
Overall		78.26	86.95	91.53	92.39

Results. The evaluation results are presented in Table 1: for each pattern, this table shows the precision and recall before (column "prior") and after (column "post") the disagreement analysis.

We observed that PPD has a high overall precision and recall, even when just comparing it directly with the human detection (see the last line in the table). For the most recurring pattern group, *Conditional Block*, both detections agreed on 194 instances of such pattern (prior). After the disagreement analysis, we found that PPD detected 39 new instances of such pattern, which increased the precision and recall of the PPD (post), for at least 86% and 89%, respectively.

For some less recurring patterns, *Single Line* and *Missing Null-Check*, PPD performed well by detecting 96 and 50 instances of these patterns in agreement with the human detection, respectively. In fact, for *Single Line*, the only two instances missing by PPD were not truly instances of such pattern.

However, we identified some particular patterns that PPD did not perform well. PPD found 95 instances of the patterns from the group *Wraps/Unwraps* in agreement with the human detection. On the disagreement analysis, we identified that PPD detected 7 new instances of this group, but that also generated 30 false positives. The major responsible for these false positives are the pattern variants involving `if`, `else` and `method`.

Table 2. Overall absolute results on the disagreement analysis and reasons for automatic detection differing from the manual detection.

Diagnostic	# Occurrences	Related Reason
DW (PPD false positive)	73	#1, #7
DC (PPD true positive)	90	#1, #4
HW (human detection false positive)	24	#5, #6
HC (human detection true positive)	65	#2, #7
B (both could be accepted)	33	#1, #3
A (agreements)	666	
TP (correct detection = A + B + DC)	789	

Discussion. During the disagreement analysis, we also investigated why PPD failed or differed from the human analysis. Table 2 relates the diagnostics with the reasons for the disagreements, which we discuss as follows.

Reason #1: Global human vision versus AST-based analysis. The GumTree algorithm identifies implicit structures that are not visible by humans. For instance, in Mockito-18, both automatic and manual detections found the pattern *Conditional Block Addition with Return Statement*. However, the automatic detection also found the pattern *Wraps-with if-else*. In this patch, the human sees the structure as in Listing 4, while the structure considered by PPD is like in Listing 5. In other words, the new conditional block wraps a part of the code, but with an implicit block. On these occurrences, we considered that both automatic and manual detection could be accepted.

```
+ } else if (type == Iterable.class) {
+     return new ArrayList<Object>(0);
  } else if (type == Collection.class)
      {
  [...]
```

Listing 4. Human vision.

```
+ } else {
+     if (type == Iterable.class) {
+         return new ArrayList<Object>(0);
      } else {
          if (type == Collection.class) {
  [...]
```

Listing 5. AST-based analysis.

Still on the global human vision versus AST-based analysis discussion, due to fine-grained changes, PPD takes into account small changes that do not make sense as the composition of a pattern in some cases. For instance, PPD detected the *Copy/Paste* pattern in Chart-3. Even though the two additions have a high similarity, these changes are not enough to be considered as an instance of the pattern *Copy/Paste*, so we determined this as a false positive generated by PPD.

In the same direction, PPD takes into account *relevant* small changes that humans may not identify in big patches. In these big patches, humans may intuitively consider only the global vision of the patch and miss smaller changes. For instance, Math-64 has several changes: one of them is the addition of a block with three lines of code in two different locations (i.e., *Copy/Paste*), which was missed by human detection.

Reason #2: The automatic detection relies on rules defined by humans (i.e., the authors of this paper), and it is difficult to identify all cases where an instance of a pattern may exist, thus PPD missed some pattern instances. The *Expression Fix* detector is the primary responsible for these missing detections. For instance, it missed the detection of an arithmetic expression fix in Math-77, where an arithmetic operation occurs with the assignment operator +=, which was replaced by a non-arithmetic assignment operator.

Reason #3: There are some borderline cases where a given pattern may fit or not. For instance, in line 1167 of Time-17, one could consider the removed method call as a part of the arithmetic expression used as argument for such method call, and another one could not. Only the manual detection detects such statement as an instance of the *Arithmetic Expression Fix* pattern, but we considered that detecting it or not can be both accepted.

Reason #4: The automatic detection applies the same rules for all the patches while it is a difficult task to be done by humans. Therefore, some pattern instances were missed by the manual detection due inconsistencies between patches.

Reason #5: In the manual analysis, humans may consider the semantic of the changes (even without noticing) and make assumptions on how the developer could write a patch that matches one of the patterns. For instance, Mockito-28 had been considered as an instance of the *Single Line* pattern. Semantically, it could be correct, but such pattern should be limited to changes affecting a single line or a single statement in a given patch.

Reason #6: A misconception of the patch can impact the human analysis. For instance, in Lang-50, the manual detection found an instance of the pattern *Logic Expression Modification* in line 285 (and also in the new line 463, which is the same case, i.e. *Copy/Paste*). However, the existing conditional block in line 285 was actually completely changed: the statement inside it was unwrapped in the patch, and the conditional was deleted. Then, an existing conditional block in the code took the place of the conditional considered as having its logic expression modified, i.e. a moving happened, not characterizing a genuine logic expression modification.

Reason #7: GumTree is a sophisticated algorithm that may return imprecise results for some patches. For example, it can consider the change over some elements when it is not the case. As a consequence, PPD incorrectly detects or misses some pattern instances.

5. Threats to Validity

Internal validity. The ultimate precision and recall calculated when evaluating the performance of the PPD are based on a manual disagreement analysis. This analysis can be subject to small errors and misconception, typical of any manual work. To mitigate this, such analysis was performed to each pattern group by two authors of this paper, in live discussion sessions.

External validity. We have evaluated PPD on patches from Defects4J. However, since Defects4J may not be representative on all different cases on fixing bugs using one of the 25 patterns, it is possible that PPD still cannot generalize for systems including patches that differ a lot from those in Defects4J. Moreover, detected repair patterns are Java-based, therefore our detector is limited to systems written in this language.

6. Related Work

Martinez et al. [Martinez et al. 2013] also reported on the automatic detection of bug fix patterns at the AST level. The main differences between their work and our work are the following. First, they focused on 18 bug fix patterns from [Pan et al. 2009] while we focused on 25 patterns from [Sobreira et al. 2018]. Second, they used the ChangeDistiller AST differencing algorithm [Fluri et al. 2007] while we use GumTree [Falleri et al. 2014]. The latter outperforms the former by maximizing the number of AST node mappings, minimizing the edit script size, and detecting better move actions [Falleri et al. 2014]. Moreover, they pointed out that ChangeDistiller works at the statement level, preventing the detection of certain fine-grain patterns. Third, they formalized a representation for change patterns and used this representation to specify patterns. Then, to detect a pattern, a match of its specification must happen in a given edit script. However, such representation is based on change type (e.g. addition) over code elements (e.g. `if`), which does not support the specification of patterns such as *Single Line*.

7. Final Remarks

In this paper, we report on PPD, a detector of repair patterns in bug fixes. Through an evaluation on Defects4J, we found that PPD has a good performance in general, and for some patterns (e.g., *Missing Null-Check*) it can even perform better than human detection. Moreover, a fruit of the disagreement analysis, we found that human detection made fewer mistakes (24) than PPD (73), but also detected less exclusive occurrences (65) than PPD (90). As future work, we intend to conduct experiments over other bug datasets to evaluate the scalability of PPD and also to compare bug datasets, which may guide researchers on automatic program repair at choosing datasets when evaluating their tools. Finally, we intend to create a visualization for patches where the repair patterns are highlighted, to support the human patch comprehension task.

Tool Availability. PPD is part of the project ADD, which is publicly available at:

`https://github.com/lascam-UFU/automatic-diff-dissection`

One can find instructions in such repository on how to use PPD and also to reproduce the results on Defects4J presented in our evaluation (Section 4).

References

Durieux, T., Cornu, B., Seinturier, L., and Monperrus, M. (2017). Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming. In *SANER '17*.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and Accurate Source Code Differencing. In *ASE '14*, pages 313–324.

Fluri, B., Wuersch, M., PInzger, M., and Gall, H. (2007). Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *TSE*, 33(11):725–743.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *ISSTA '14*, pages 437–440.

Martinez, M., Duchien, L., and Monperrus, M. (2013). Automatically Extracting Instances of Code Change Patterns with AST Analysis. In *ICSM '13*, pages 388–391.

Monperrus, M. (2018). Automatic Software Repair: a Bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24.

Pan, K., Kim, S., and Whitehead, Jr., E. J. (2009). Toward an understanding of bug fix patterns. *EmSE*, 14(3):286–315.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. (2015). Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179.

Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., and Maia, M. A. (2018). Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *SANER '18*.