Coming: A Tool for Mining Change Pattern Instances from Git Commits

Matias Martinez, Martin Monperrus

Coming: a Tool for Mining Change Pattern Instances from Git Commits

Matias Martinez
University of Valenciennes, France

Martin Monperrus
KTH, Sweden

***Abstract*—Software repositories such as Git have become a relevant source of information for software engineer researcher. For instance, the detection of Commits that fulfill a given criterion (e.g., bugfixing commits) is one of the most frequent tasks done to understand the software evolution. However, to our knowledge, there is not open-source tools that, given a Git repository, returns all the instances of a given change pattern.**

In this paper we present Coming, a tool that takes an input a Git repository and mines instances of change patterns on each commit. For that, Coming computes fine-grained changes between two consecutive revisions, analyzes those changes to detect if they correspond to an instance of a change pattern (specified by the user using XML), and finally, after analyzing all the commits, it presents *a)* the frequency of code changes and *b)* the instances found on each commit.

We evaluate Coming on a set of 28 pairs of revisions from Defects4J, finding instances of change patterns that involve *If* conditions on 26 of them.

I. Introduction

During recent years software engineering researchers have been inspecting software code repositories such as Git or SVN to gain knowledge about the evolution of applications. For example, there are a considerable number of studies ([3], [12], [18]) that have focused on the bug fixing activity by studying bug fix commits. Other researcher have also presented approaches that aim at repairing automatically buggy programs. Some of those repair approaches [7], [6] consume information extracted from software repositories such as the most frequent bug fix patterns [5] or frequency of code changes [16].

To carry out such kind of analysis on repositories, a researcher needs a tool that: *a)* visits a set of revisions (i.e., commits); *b)* filters those revisions that are interesting (e.g., bug fix commits) according to, for example, the commit message; *c)* computes changes between a revision and its precedent; *d)* summarizes the results (e.g., to compute the probability of each change type); *e)* capture the commits that introduces a set of particular changes; among other activities. However, to our knowledge, there is not an open-source that carries out all of these tasks.

In this paper we present *Coming*, a tool that inspects Git repositories with two main goals: *1)* to compute fine-grained changes between revisions, and *2)* to detect instances of change patter.

In a nutshell, Coming takes as inputs a list of revisions (e.g., Commits from git). For each pair of consecutive revisions, i.e., r and $r+1$, Coming first computes the changes between them at a fine-grained level using an AST-diff algorithm. Then, it analyzes the changes to detect if they correspond to a change pattern given as input. A change pattern specifies a set of changes (e.g., insert, remove) done over code entities (e.g., invocations, assignments). A pair of revisions has an instance of a pattern if: *1)* all the changes that a pattern specifies exist on the diff between those revisions; *2)* the entities affected by the changes from the diff match with those that the pattern specifies. Finally, after analyzing all the commits, Coming post-processes the results from each pair of revisions and exports the final results (i.e., pattern instances found and frequency of code changes) to a JSON format. Moreover, Coming provides extension points for overriding the default behaviour or to add new functionality.

Coming can be used by researchers that aim at filtering commits to automatically create, for instance, datasets of bugs. Moreover, it can be used by researchers that aim at post-processing a distilled set of changes found by the tool, and then apply, for instance, an algorithm of change clustering.

To evaluate Coming, we first collect 28 pairs of revisions from Defects4J [4] which diffs affected to, at least, one *if* condition. Then, we write change patterns that specify different changes over *if*. Finally, we execute Coming to detect instances of such patterns over the 28 pairs of revisions. Coming could successfully find the correct instance of a change pattern on 26 pairs.

Coming is publicly available at https://github.com/Spirals-Team/coming. The video that shows a demonstration of Coming is available at https://youtu.be/dR6B9qRpjic

II. Approach

A. Goals of Coming

The main goals of Coming are: *a)* to compute the fine-grain changes between two revisions; *b)* to detect instances of change pattern from those fine-grain changes; *c)* to count the frequency of changes along all the revisions of a Git repository; and *d)* to count the occurrence of change pattern instances.

In the rest of this section, we present the series of steps that Coming carries out to accomplish those goals.

B. Coming Inputs

Coming analyzes commits from a Git repository, whose path is given as parameter. The implementation of Coming navigates the Git history using the library Eclipse GIT.[1] Coming navigates each commit starting from the oldest one. For each commit c, Coming takes each file f that the commit c modifies, and creates a revision pair with: *a)* a file f modified by C, and *b)* the previous version of f, introduced or modified by a commit older than C.

C. AST-based Analysis of Revisions

Coming carries out a fine-grained comparisons based on the AST (Abstract Syntax Tree) of each revision pair. This step has two main steps:

1) AST Representation of Files: Coming creates a AST from the code of a revision. In a AST, tree node corresponds to a code element (e.g., an invocation, a parameter). Previous works have been working on different granularities of tree nodes: from coarse-grained from ChangeDistiller [2], where the finest-grained code element represented by a node is the statement, to fine-grained such as Eclipse JDT [1].

In this paper, we present a new level of granularity of AST, named GTSpoon, which is based on the Spoon meta-model.[2] Spoon [13] is a library to analyze, transform, rewrite, transpile Java source code.

The granularity-level of the Spoon meta-model is between the previously two mentioned: *a)* Spoon nodes are finer (e.g., parameter, field write and read) that ChangeDistiller (statements); and *b)* those nodes contain more information than the Eclipse JDT nodes, resulting more compact trees. [3]

Coming uses, by default, the GTSpoon granularity, which means that each node of the AST of a revision r is corresponds to an element from the Spoon model of r. We have written an open-source library named GTSpoon[4] that returns a AST with that granularity from a source code file.

2) Tree-Diff Comparison: To obtain the different between two models m_s and m_t (by default GTSpoon's ASTs) retrieved from two revisions r_s and r_s, resp., Coming applies a tree-difference algorithm. By default, Coming uses GumTree [1], a state-of-the-art AST diff algorithm. The output of the diff between m_s and m_t is a list of *Operations*, where each of them contains *a)* an action type (Insert, Remove, Update, Move); *b)* a reference to a node from m_s; and/or *c)* a reference to a node from m_t.

D. Analysis of Diffs: Finding Instances of Change Pattern

Coming executes a set of *Analyzers* which take as input the results of previous diffs and carry out some task. In this paper we present an analyzer that mines instances of change patterns. The analyzer uses (and improves) the specification of change pattern that we have previously defined [9] and implements the algorithm, also defined tehre, to match patterns and changes.

A *Change Pattern* defines a set of changes between two revisions and the elements affected by those changes.

A pattern has a list called *Pattern actions* pa where each of them specifies a particular change between two revisions. A *Pattern action* has two fields. First, the type of action, with four predefined values: insert, mode, remove, and update. Secondly, it contains a reference to a $PatternEntity$ which has also three fields: *1) type*, which indicates the type of code element of the entity (e.g., if, invocation, return); *2) value*, which indicates the value of the element (e.g, $callMethod1(), return\quad null$); *3) parent*, a recursive relation to a $PatternEntity$ which indicates the parent of an entity. This relation has an argument, *distance*, which indicates the max distance between e and ep in the AST. For example, a value of 1 indicates that the ep is the immediate parent of e, whereas 2 means a grand-parent relation. The use of wildcard character "*" in any of those mentioned fields produces a matching with any kind of type or value.

Coming accepts Change Patterns specified in a XML files. Listing 1 shows as example a pattern in XML that specifies: *a)* two entities (id 1 and 2), one representing a $Return$, the second one an If; *b)* a parent relation between the if and the $Return$ entities (with a max distance of 2 nodes); and *c)* two actions of type INS (insert), one affecting the entity id 1 ($Return$), the other one the entity id 2 (the if).

Listing 1. Change Pattern *Add If-Return*

```
<pattern>
        <entity id=''1'' type=''Return''>
                <parent parentId=''2'' distance=''2'' />
        </entity>
        <entity id=''2'' type=''If'' />
        <action entityId=''1'' type=''INS'' />
        <action entityId=''2'' type=''INS'' />
</pattern>
```

E. Summarization of Results

Finally, Coming processes all results obtained from the analyzers over all the commits and then it exports the results to a JSON file.

F. Extending Coming

Beyond the functionality that Coming already includes, such as AST differencing and mining of change pattern instances, it provides *extension points* to override the default behaviour and to define new tools for evolution analysis. The main extension points that Coming provides are the following, with, in parentheses the implementation already provided. *a) Input* (Git, Files System); *b) Revision filter* (presence of keywords in revisions messages, size of the revisions in terms of # of hunks and in terms of # files); *c) Analyzers* (Computation of syntactical (line-based) diff, AST-based diff, pattern instance detection); *d) Output processor* (Standard output, JSON containing the instances found and change frequency).

In the documentation hosted in the Coming Github site,[5] we explain how to create an new implementation for each extension point.

[1] https://projects.eclipse.org/projects/technology.egit

[2] Spoon meta-model: http://spoon.gforge.inria.fr/code_elements.html and http://spoon.gforge.inria.fr/structural_elements.html

[3] Discussion about the Spoon and JDT granularity: https://github.com/INRIA/spoon/issues/1303

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

[5] https://github.com/Spirals-Team/coming/blob/master/docs/extension_points.md

TABLE I

MINING CHANGE PATTERN INSTANCES OVER BUGGY AND PATCHED VERSION OF DEFECTS4J DEFECTS. THE COLUMNS TP, FN AND TN SHOWS THE TRUE POSITIVES, FALSE NEGATIVES AND TRUE NEGATIVES. A DEFECTS4J'S REVISION COULD HAVE +1 PATTERN INSTANCES.

Change Pattern	TP (instances found)	FN	TN
Add If-return	M3, M38, M53,M55, M84, M92, M93	M60	-
Add If-return null	M4	-	-
Add If-assig	M29, M51, M54, M102	-	-
Add If-throw	M19, M25, M45, M48, M73, M99	-	M86
Upd If-cond	M21, M37	-	-
Add 2 nested Ifs	M39, M68, M78	M64	-
Mov If-return	M64	-	-
Add If-break	M1	-	-
Del If-return	-	-	M64
Add If Mov assig	M95	-	-

III. EVALUATION

This experiment aims at measuring the ability of Coming to detect change pattern instances. For this propose, we create a set of 10 patterns from the related work. Then we run Coming over 28 pairs of revisions to mine instances of those patterns.

A. Experiment setup

In this experiment, we aim at mining instances of change pattern detecting bug fixing. We consider Defects4J [4], a dataset of buggy programs from 6 Java open-source projects. It contains, for each buggy program, a patch that repairs the bug. Due to the scope of this paper, we focus on buggy programs: *1)* from the Apache Commons Math project; and *2)* whose patches affect *if* conditions. In total, with the help of the Defects4J dissection [15], the number of buggy programs satisfying those criteria is 28. For each of those bugs, we prepare the buggy and the patched version according to Coming's input format.

Then, we create a set of 10 change patterns to detect the changes that affect the *if* conditions. For instance, the first pattern *Add If-return* corresponds to that one presented in Listing 1. It is able to detect, for instance, the changes between the buggy and patched version of bug M3 from Defects4J, which patch is shown in Listing 2.

Listing 2. Bug fix changes corresponding to bug Math-3. It is an instance of Change Pattern *Add If-Return* presented in Listing 1.

```
@@ −818,10 +818,7 @@ public class MathArrays {
+ if (len == 1) {
+ return a[0] ∗ b[0];
+ }
```

Finally, we execute Coming over the 28 pairs of buggy and patched revisions from Commons Math projects. We then manually inspect the results i.e., the mined instances from the revisions, to assert whether they are: *a) true positive* i.e., the pattern instance exists between the revisions, *b) false negatives* i.e., Coming could not detect an instance of the pattern.

B. Experimental Results

Table I shows the results of our experiment. The first column shows the change pattern name.

The second column (TP) shows, for each pattern p, the Defects4J *identifier* for which Coming can successfully find a pattern instance of p between the buggy and the patched version. For example, Coming correctly identifies an instance of pattern "Add-If-Return" in revision Math-3. In total, Coming can find correctly instances for 26 out of 28 patches (93%), those are true positives. Moreover, Coming is capable of finding more than 1 instances of the same pattern inside a revision pair. For instance, the revisions Math-93 has two instances of pattern "If-Return".

Then, the column FN shows the false negatives, i.e., the revisions that actually have an pattern instance but Coming fails to detect it. We observe that the two false negatives are due the AST diff algorithm (Gumtree in vanilla mode) which did not create a correct minimal diff, i.e., it produces unnecessary INS and DELETE operations.

Finally, the column TN shows the cases considered as true negative, i.e., Coming does not return any instance (correctly). The line line-based diff shows that *if* conditions are added and removed, but Coming does not detect any instance of patterns *Add If-** or *Del if-** A true negative occurs in Math-64, which patch is partially presented in Listing 3.

Listing 3. Two hunks from the patch Math-64. The Tree diff algorithm detects that the if condition is moved

```
+ if (checker.converged(getIterations(), previous, current)) {
+   return current;
+ }
+ }
− } else {
− if (checker.converged(getIterations(), previous, current)) {
− return current;
− }
```

The listing shows two hunks, one that adds an *if*, another that removes the same *if* code. From that revision pair, the AST-diff algorithm Gumtree detects *move* operations (both the *If* and *return* elements are moved to another location). Thus, Coming is not able to find an instance of the pattern *Del If-return* giving those two AST changes (Moves). Consistently, when Coming mines instances of the pattern *Mov If-return*, it successfully finds one between the buggy and patched version of Math-64.

Lastly, Table I shows a pattern "Add If-return null" that specify the value of the entity, in addition to the entity type. Using this feature, Coming can identify, for instance in Math-4, an instance of an *if* that returns a $null$ value.

The code base of Coming includes the specifications of all the patterns presented in this experiment.[6]

IV. RELATED WORK

Coming uses the method to specify a change pattern that we presented in [9] and implements the instance mining algorithm presented in that work. Moreover, Coming provides several improvements including: *1)* a finer-grained level of ASTs, which allows to create more precise change pattern; *2)* matching of entity values; *3)* more descriptive parent

[6]https://github.com/Spirals-Team/coming/blob/master/docs/experiment_mining_instances_

relation (allowing a chain of parents); *4)* the use of a more reliable tree-diff algorithm [1].

There are other open-source tools that focus on the analysis of software repositories such as Gits. Some of them are: PyDriller[7] [17], Git-of-theseus[8], CVSAnalY[9] and Hercules[10]. However, to our knowledge, these tools do neither provide a fine-grained analysis of changes between revisions, nor the detection of change instances.

Different approaches have focused on the mining of bug fix pattern. For instance, Madeiral et al. [8] have presented an approach that detects repair patterns in patches, which performs source code change analysis at abstract-syntax tree level. Their approach, as it is also built over our technology stack (GTSpoon, Spoon and GumTree), could be easily included in Coming using the extension point *output processor*. Osman et al. [11] analyze code hunks from line-based diff to detect bug-fix patterns, Rolim et al. [14] propose a method for discovering quick fixes based on the identification of code edits (at the level of AST) from revisions, and then to cluster those edits. A similar work has been done by Molderez et al. [10] which use closed frequent itemset mining algorithm on sets of distilled code changes. Hanam et al. [3] present a tool for discovering the most prevalent and detectable bug patterns on JavaScript code, based on unsupervised machine learning. As difference of those works, Coming focuses on the detection of *instances* of existing change pattern, including bug fix pattern. Moreover, as Coming computes the fine-grained diffs and also provides extension points to analyze that information, any of those works can be implemented in our tool.

V. Future Work

The current version of Coming includes all features presented in this paper. Nevertheless, we continue working on new features and improving the tool usability. Some of the planned features are: *a)* Enrichment of the pattern specification to include cardinality of elements (numbers of children, siblings, etc), assert the absence of elements, different matching strategies of entity types and values, and accept changes that affect different files; *b)* parallelisation; *c)* post-processors to mine, for instance, change patterns; *d)* tuning arguments of tree-diff algorithm to avoid true negatives.

VI. Conclusion

In this paper we present the tool named Coming which given a Git repository, navigates every commit, calculates fine-grained changes between a revising of a commit and its precedent, detects change pattern instances from those changes, computes the frequency of code changes along the repository and finally exports the results in JSON format. Coming presents extension points allowing researchers to plug-in their own approaches that, for example, focus on the discovering of bug-fix patterns from the changes computed by Coming. Coming is publicly available at https://github.com/Spirals-Team/coming/. New features and extensions are welcome via Pull Request.

[7]https://github.com/ishepard/pydriller
[8]https://github.com/erikbern/git-of-theseus/
[9]https://github.com/MetricsGrimoire/CVSAnalY
[10]https://github.com/src-d/hercules

References

[1] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[2] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Softw.*, 26(1):26–33, January 2009.

[3] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 144–156, New York, NY, USA, 2016. ACM.

[4] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[5] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[6] X. B. D. Le, D. Lo, and C. L. Goues. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224, March 2016.

[7] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[8] Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. Towards an automated approach for bug fix pattern detection. In *VEM '18 - Proceedings of the VI Workshop on Software Visualization, Evolution and Maintenance*, So Carlos, Brazil, September 2018.

[9] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 388–391, Washington, DC, USA, 2013. IEEE Computer Society.

[10] T. Molderez, R. Stevens, and C. De Roover. Mining change histories for unknown systematic edits. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 248–256, May 2017.

[11] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 343–347, Feb 2014.

[12] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[13] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exper.*, 46(9):1155–1179, September 2016.

[14] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories, 2018.

[15] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j.

[16] Mauricio Soto and Claire Le Goues. Common statement kind changes to inform automatic program repair. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 102–105, New York, NY, USA, 2018. ACM.

[17] Davide Spadini, Maurcio Aniche, and Alberto Bacchelli. *PyDriller: Python Framework for Mining Software Repositories*. 2018.

[18] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015. IEEE Press.