# Test2Feature: Feature-based Test Traceability Tool for Highly Configurable Software

Willian D. F. Mendonça
Silvia R. Vergilio
DInf, Federal University of Paraná
Curitiba, Paraná, Brazil

Gabriela K. Michelon
Alexander Egyed
ISSE, Johannes Kepler University Linz
Linz, Austria

Wesley K. G. Assunção
ISSE, Johannes Kepler University Linz
Linz, Austria
Opus, Pontifical Catholic University
of Rio de Janeiro
Rio de Janeiro, Brazil

## ABSTRACT

To ensure the quality of Highly Configurable Software (HCS) in an evolution and maintenance scenario is a challenging task. As HCSs evolve, new features are added, changed, or removed, which hampers the selection and evolution of test cases. The use of test traceability reports can help in this task, but there is a lack of studies addressing HCS test-to-feature traceability. Existing work usually are based on the variability model, which is not always available or updated. Some tools only link test cases to code lines. Considering this gap, this paper introduces `Test2Feature`, a tool that traces test cases to features using the source code of annotated HCSs, written in C/C++. The tool produces the following outputs: the code lines that correspond to each feature, the lines that correspond to each test case, and the test cases that are linked to each feature. `Test2Feature` is based only on the static analysis of the code. The traceability report produced can be used to ease different tasks related, for instance, to regression testing, feature management, and HCS evolution and maintenance.

## CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software testing and debugging**; **Traceability**; **Software maintenance tools**.

## KEYWORDS

Highly Configurable Software, Regression Test, Feature-based, Test Traceability

**ACM Reference Format:**
Willian D. F. Mendonça, Silvia R. Vergilio, Gabriela K. Michelon, Alexander Egyed, and Wesley K. G. Assunção. 2022. Test2Feature: Feature-based Test Traceability Tool for Highly Configurable Software. In *26th ACM International Systems and Software Product Line Conference - Volume B (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3503229.3547031


*SPLC '22, September 12–16, 2022, Graz, Austria*

ACM ISBN 978-1-4503-9206-8/22/09…$15.00
https://doi.org/10.1145/3503229.3547031

## 1 INTRODUCTION

*Highly Configurable Software (HCS)* provides adaptable and flexible solutions to complex and real-world problems. HCSs are usually implemented by using different configuration options to create custom system products, also known as variants [20]. Quality assurance of HCS is a fundamental issue, but this involves some challenges. Ideally, all possible configurations should be tested, but this is often unfeasible because the number of possible configurations grows exponentially with the number of features. This makes HCS testing more complex and expensive, as many features often need to be tested and several test cases overlap [16]. For instance, the test case selection in a regression testing scenario requires specific approaches to consider the HCS evolution in space and time, where features are added, modified, or removed [15].

For easing that and other tasks, approaches and tools based on test traceability could be used. Tufail et al. [19] developed a tool for test traceability, which links test cases to a set of requirements without requiring the program execution or test coverage, which would be expensive; but these tools do not consider HCSs. A recent mapping on testing tools for HCSs [5] reports few tools addressing test case traceability for features [1, 7, 11]. These tools are based on the *Feature Model (FM)*. This can be a limitation because, for many HCSs, the FM is unavailable or outdated. Existing tools for HCS that are based on source code [8, 9, 12] do not generate traceability for features, but only link test cases to the lines of code. Another limitation is that the great majority of these tools can be used only for Java code.

Motivated by these facts, this paper introduces `Test2Feature`, a tool that, given the source code of an annotated HCS and a test case set, produces test traceability reports linking test cases to the HCS features. The tool encompasses three modules that generate three main outputs: traceability from features to code lines, traceability from test cases to code lines, and traceability from features to tests. In this sense, the main contribution of this work is to describe a tool that traces test cases to features from the source code of HCSs. The tool is available online[1] and allows developers and testers to know the lines of the source code and features that correspond to a test case. Differently from existing tools, `Test2Feature` works with annotated HCSs written in C/C++ and is based only on the static analysis of the code. The traceability report produced can ease different tasks, as for example, regression testing, feature management, and HCS evolution and maintenance.

The paper is structured as follows. Section 2 overviews related work. Section 3 describes the tool architecture. Section 4 illustrates

[1] https://github.com/willianferrari/Test2Feature.git

the outputs of Test2Feature through a case study. Section 5 concludes the paper.

## 2 RELATED WORK

As mentioned before, there are many approaches and tools in the literature to generate test traceability to requirements using static analysis [19]. Different kinds of artefacts are used to specify these requirements, such as use cases and user stories. But they usually do not trace test cases for HCS code. We have found a few work that link test cases to the HCS code lines [8, 9, 12], but they do not generate traceability to features. Another limitation is that they generate traceability only for Java code.

We observed in the literature a lack of testing tools for HCSs written in C/C++. Out of 34 tools reported by a recent mapping on testing tools for HCS/SPL [5] only four can be applied to C/C++ [2, 4, 6, 18], and no one of these tools addresses test case traceability. The mapping also reports some pieces of work addressing test-to-feature traceability in the context of test case selection and prioritization [1, 7, 11]. But the problem of these work is that the approaches they implement are based on an FM.

Differently from the related work mentioned above, our work generates the test-to-feature traceability from the annotated code of HCS. This brings some advantages for an HCS evolution scenario because FMs are not always available and updated. We observe a lack of tools for C/C++ language that link features to the source code in the context of HCS. In addition to this, we have not found a tool to link features to test cases, which is the main goal of Feature2Test, described in the next section.

## 3 TEST2FEATURE MODULES

Test2Feature[2] encompasses three modules as depicted in Figure 1. The input provided by the software engineer is a configuration file containing the paths for the system repository and test folder, as well as the current commit to be analyzed. This information is passed to the module *MineFeaturesLines*, responsible for localizing features by code lines, and to the module *MineTestLines*, responsible for identifying the code lines that correspond to each test case of the test folder. *MineTestLines* uses the tool Doxygen[3] to generate XML files representing the dependency graph for the functions of the C/C++ code. The information of these both modules is used by the module *MergeTestFeaturesLines* to generate a CSV file that contains the traceability of features to the test cases.

**Configuration File**: in the configuration file the software engineer sets the names of the test and system folders. In addition, s(he) can choose any previous commit by using Commit Sha. This is illustrated in Figure 2. The engineer needs to set four variables: 1) REPGIT: which contains the system URL in GitHub; 2) SYSFOLDER with the name of the system folder; 3) TESTFOLDER with the name of the test folder of the system; and 4) NCOMMIT with the Sha Hash.

***MineFeaturesLines***: this module uses the implementation of the approach for mining features in space and time, described in our previous work [13, 14]. The approach abstracts the source code of

[2]A video demonstrating the core features of the TEST2FEATURE tool is available at: https://www.youtube.com/watch?v=6RN6F1TNYr8
[3]https://doxygen.nl/index.html

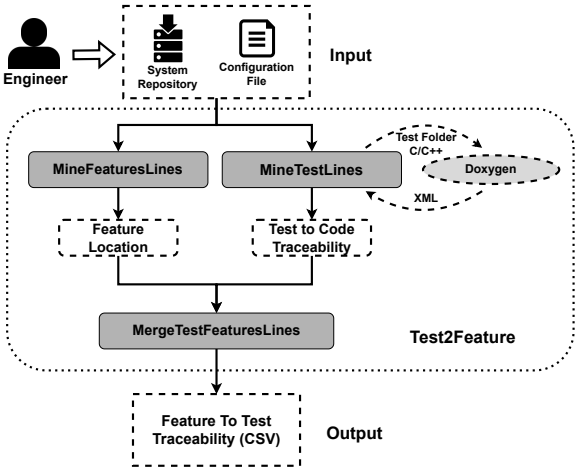


**Figure 1:** Test2Feature **architecture**

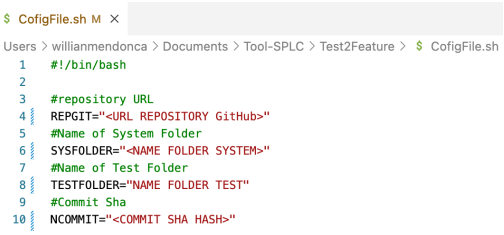


**Figure 2: Configuration File Content**

a commit snapshot at the level of preprocessor directives, which are distinguished in conditional blocks, i.e., #if, #ifdef, #elif, #else, and #ifndef; definition lines, i.e., #define and #undef directives; or import file lines, containing #include directives. This abstraction is less computationally expensive and time-consuming, as we do not need to analyze the abstract syntax tree to obtain, for each file, all the lines of code containing preprocessor directives. The approach uses a Constraint Satisfaction Problem (CSP) [17] to reliably identify interacting features or features depending on the execution of other features [3], and thus which features belong to which conditional blocks to obtain the features lines.

Figure 3 illustrates how the approach for mining features lines works. The figure contains four variation points/conditional blocks wrapped by conditional directives. For each conditional block, the approach creates constraints related to each particular conditional block of code. For instance, lines 1-3 belong to feature A. This is the simplest case, where there are no interactions or nested features. But the block of code of lines 6-8 belongs to a feature internally defined, inside feature A. This block of code of lines 6-8 is activated when feature A is selected, and thus when B is greater than 10. However, this is not the only constraint to take into account to determine which features belong to the block of code of lines 6-8 because there is an outermost block wrapping lines 6-8 with the conditional expression #if C. In this case, feature C also has to be selected so that this block of code can be executed. Therefore, in such cases, where multiple features imply executing a block of code, a heuristic is adopted to consider the lines as part of the closest

```
1   #ifdef A
2       #define B 15
3   #endif
4
5   #if C
6       #if B > 10
7           <code>
8       #endif
9   #endif
10
11  #ifndef D
12          <code>
13  #endif
14
15  <code>
```

**Figure 3: Conditional blocks of feature implementations.**

feature (not defined internally via `#define` directive) to the block of code. In this way, the block of code of lines 6-8 is assigned to the feature `C`. Therefore, the lines of code of feature `C` begins at line 5 and ends at line 9.

We also have a corner case example [10] at lines 11-13, where there is a negated conditional expression, i.e., the block of code is executed when feature `D` is not selected. In this case, there are no nested features or feature interactions, and this block of code is thus considered part of the system core (`BASE` feature), as there is no feature responsible for executing this block. After getting the features responsible to execute each block of code, we obtain the line numbers of each file that belongs to a feature. Therefore, lines 1-3 belong to feature `A`; lines 5-9 to feature `D`; lines 11-13 belong to `BASE`, as well as line 15, which is outside of any variation point.

***MineTestLines***: this module generates as result the traceability of test cases to system functions by using the output of `Doxygen`. This tool has the GNU General Public License, initially developed with a view to keeping the source code of systems with annotated C++ code documented, but also has support for other languages like C, C#, Python, Java, and so on. Doxygen supports visualization of the relationships between various elements through dependency graphs, inheritance, and collaboration diagrams, generated automatically, in different formats. *MineTestLines* uses the function dependency graph in XML format. We defined the minimal set of parameters in order to generate all possible dependencies available in Doxygen and to make the tool execution faster.

***MergeTestFeatureslines***: this module produces a merge between the *MineTestLines* and *MineFeaturesLines* outputs, e.g., the location of the features along with the location of the test cases. In this way, it is possible to know exactly the location of the tests and features per line of the files. Initially, a merge is performed considering the localization files; then, a filter is applied considering the location of the code lines.

***Implementation Aspects***: For the development of `Test2Feature` we used the Python programming language, in version 3.9.10. To deal with the CSV files the PANDAS data science library is used. The module *MineFeaturesLines* was developed in Java and uses as input a system folder already cloned and a Commit Sha. As output, it delivers a CSV file containing the location of the features per line.

## 4 ILLUSTRATIVE EXAMPLE

This section describes an example of our tool and the results generated by their modules. For this end, we use the `SQLite`[4] database system, available on GitHub. `SQLite` is an industrial case study, used on a large scale, and daily updated. After setting the variables in the configuration file, the modules *MineFeaturesLines* and *MineTestLines* can be run independently. Figure 4 shows an excerpt from the source code of the main.c file that belongs to `SQLite`, and Figure 5 presents the outputs produced by the modules *MineFeaturesLines* and *MineTestLines*. The `SQLITE_OMIT_WSD` feature of the code snippet in Figure 4 is responsible to execute the conditional block of lines 376-381.

```
375  int sqlite3_shutdown(void){
376  #ifdef SQLITE_OMIT_WSD
377    int rc = sqlite3_wsd_init(4096, 24);
378    if( rc!=SQLITE_OK ){
379      return rc;
380    }
381  #endif
```

**Figure 4: Features `SQLite`**

*MineFeaturesLines* and *MineTestLines* are modules for feature location (traceability of features to code) and traceability of the test cases to the code, respectively. Figure 5 shows an example of test-to-function traceability generated by the module *MineTestLines*. Test case traceability has the granularity of test cases to function considering the start line (column LineFrom) and end line (column LineTo); the test case `registerOomSimilator` (column TestCase) belonging to the file `fuzzcheck.c` (column TestFile) uses the function `sqllite3_shutdown` (column TargetFunction), located in the file main.c (column TargetFile) starting in line 375 and ending in line 418. The module *MineFeaturesLines* also generates a result that can be used by engineers for decision-making. Such a result can also be seen in Figure 5. We can observe the traceability between the output files, in this case, the feature `SQLITE_OMIT_WSD` (FeatureName) is located in the file `main.c` (TargetFile) between lines 376 and 381.

| TestFile | TestCase | TargetFile | TargetFunction | LineFrom | LineTo |
|---|---|---|---|---|---|
| fuzzcheck.c | registerOomSimulator | main.c | sqlite3_shutdown | 375 | 418 |
| fuzzcheck.c | disableOom | fuzzcheck.c | oomCounter | 627 | |
| fuzzcheck.c | disableOom | fuzzcheck.c | oomRepeat | 628 | |
| fuzzcheck.c | isOffset | optfuzz-db01.txt | c | 22 | |
| fuzzcheck.c | isOffset | fuzzcheck.c | hexToInt | 683 | 690 |
| fuzzcheck.c | decodeDatabase | optfuzz-db01.txt | a | 22 | |

| TargetFile | FeatureName | FeatFrom | FeatTo |
|---|---|---|---|
| src/main.c | SQLITE_OMIT_WSD | 376 | 381 |
| src/sqliteInt.h | SQLITE_OMIT_WSD | 1129 | 1135 |
| src/ctime.c | SQLITE_OMIT_WSD | 680 | 682 |
| src/test_multiplex.c | SQLITE_OMIT_WSD | 529 | 531 |
| src/tclsqlite.c | SQLITE_TCL_DEFAULT_FULLMUTEX | 3751 | 3753 |
| src/test_config.c | SQLITE_OMIT_TCL_VARIABLE | 646 | 648 |
| src/ctime.c | SQLITE_OMIT_TCL_VARIABLE | 645 | 647 |

**Figure 5: Outputs from *MineTestLines* and *MineFeaturesLines*.**

[4]https://github.com/sqlite/sqlite

Figures 6 and 7 show the outputs of the module *MergeTestFeatureslines*, which are, respectively, traceability from features to test cases and traceability from test cases to features. Figure 6 shows a file presenting traceability considering that the range corresponding to the feature belongs to the range that corresponds to the function on which the test case depends on. For example `registerOomSimilator` (TestCase) depends on function `sqllite3_shutdown` that corresponds to the lines 375 (LineFrom) to 418 (LineTo), and `SQLITE_OMIT_WSD` (FeatureName) corresponds to the lines 376 (FeatFrom) to 381 (FeatTo). We observe that lines 376 to 381 are within the range 375 to 418. Then, `SQLITE_OMIT_WSD` is within the range of the function on which the test case depends.

Figure 7 presents traceability considering that the function on which the test depends is inside the range that corresponds to the feature. For example, `vacuum1` (TestCase), which is located in the file `tt3_vacuum.c` (TestFile), depends on the function `sqlite3_enable_shared_cache` (TargetFunction), located in the file `btreee.c` (TargetFile), between lines 89 and 92. In this case, the feature `BASE` (FeatureName) is between lines 78 and 93. Thus, the test case is within the range that corresponds to the feature.

| TestFile | TestCase | TargetFile | TargetFunction | LineFrom | LineTo | FeatureName | FeatFrom | FeatTo |
|---|---|---|---|---|---|---|---|---|
| fuzzcheck.c | registerOomSimulator | main.c | sqlite3_shutdown | 375 | 418 | SQLITE_OMIT_WSD | 376 | 381 |
| fuzzcheck.c | runCombinedDbSqlInput | main.c | sqlite3_file_control | 3895 | 3940 | BASE | 3936 | 3938 |
| fuzzcheck.c | runCombinedDbSqlInput | main.c | sqlite3_file_control | 3895 | 3940 | SQLITE_UNTESTABLE | 3936 | 3938 |
| fuzzcheck.c | runCombinedDbSqlInput | main.c | sqlite3_initialize | 200 | 365 | BASE | 205 | 212 |
| fuzzcheck.c | runCombinedDbSqlInput | main.c | sqlite3_initialize | 200 | 365 | BASE | 212 | 295 |
| fuzzcheck.c | runCombinedDbSqlInput | main.c | sqlite3_initialize | 200 | 365 | BASE | 295 | 309 |
| fuzzcheck.c | runCombinedDbSqlInput | main.c | sqlite3_initialize | 200 | 365 | BASE | 309 | 317 |

**Figure 6: Traceability of Features to Tests.**

| TestFile | TestCase | TargetFile | TargetFunction | LineFrom | LineTo | FeatureName | FeatFrom | FeatTo |
|---|---|---|---|---|---|---|---|---|
| tt3_vacuum.c | vacuum1 | btree.c | sqlite3_enable_shared_cache | 89 | 92 | BASE | 78 | 93 |
| tt3_vacuum.c | vacuum1 | btree.c | sqlite3_enable_shared_cache | 89 | 92 | BASE | 81 | 93 |
| tt3_shared.c | shared1 | btree.c | sqlite3_enable_shared_cache | 89 | 92 | BASE | 78 | 93 |
| tt3_shared.c | shared1 | btree.c | sqlite3_enable_shared_cache | 89 | 92 | BASE | 81 | 93 |
| tt3_index.c | create_drop_index_1 | btree.c | sqlite3_enable_shared_cache | 89 | 92 | BASE | 78 | 93 |
| tt3_index.c | create_drop_index_1 | btree.c | sqlite3_enable_shared_cache | 89 | 92 | BASE | 81 | 93 |
| threadtest3.c | dynamic_triggers | btree.c | sqlite3_enable_shared_cache | 89 | 92 | BASE | 78 | 93 |

**Figure 7: Traceability of Tests to Features.**

## 5 CONCLUSIONS

This paper introduces `Test2Feature`, a tool to generate test-to-feature traceability from the code of an annotated HCS to the test cases of its features. The tool has three modules that produce the following outputs: the code lines that correspond to each feature, the lines that correspond to each test case, and the test cases that are linked to each feature. In this way, `Test2Feature` allows developers and testers to know which lines of the source code correspond to a test case, and which features correspond to the lines of a test case.

Furthermore, the tool was developed to work at the script level. This makes it possible to insert the tool as part of larger processes for automatic decision-making. For instance, the results can be used by other approaches/tools for a detailed analysis of the test cases' behavior, and/or mapping the co-change of features and test cases in space in time, hence easing maintenance and evolution tasks.

As future work, we intend to develop a friendly interface and also organize the results in different ways. We intend to implement other mechanisms and modules to ease regression testing activities such as test case prioritization in continuous integration of HCSs, selection of test cases linked to some particular features, and generation of test cases based on the code modifications.

## ACKNOWLEDGMENTS

This work is supported by CAPES (grant: 88887.464736/2019-00), CNPq (grant: 305968/2018-1), the LIT Secure and Correct Systems Lab, and FAPERJ, under the PDR-10 program (grant 202073/2020).

## REFERENCES

[1] M. Al-Hajjaji, J. Meinicke, S. Krieter, R. Schröter, T. Thüm, T. Leich, and G. Saake. 2016. Tool demo: testing configurable systems with featureIDE. In *GPCE*. 173–177. https://doi.org/10.1145/2993236.2993254

[2] S. Apel, H. Speidel, P. Wendler, A. Von Rhein, and D. Beyer. 2011. Detection of feature interactions using feature-aware verification. In *ASE*. IEEE, 372–375. https://doi.org/10.1109/ASE.2011.6100075

[3] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. 2006. *Using Java CSP Solvers in the Automated Analyses of Feature Models*. Springer, 399–408. https://doi.org/10.1007/11877028_16

[4] G. Cavarlé, A. Plantec, S. Costiou, and V. Ribaud. 2018. A feature-oriented model-driven engineering approach for the early validation of feature-based applications. *Science of Computer Programming* 161 (2018), 18–33.

[5] F. Ferreira, J. P. Diniz, C. Silva, and E. Figueiredo. 2019. Testing tools for configurable software systems: A review-based empirical study. In *VAMOS*. 1–10.

[6] C. Henard, M. Papadakis, and Y. L. Traon. 2014. Mutation-based generation of software product line test configurations. In *SSBSE*. Springer, 92–106.

[7] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Model Driven Engineering Languages and Systems*. Springer, 638–652.

[8] C. H. P. Kim, D. S. Batory, and S. Khurshid. 2011. Reducing combinatorics in testing product lines. In *AOSD*. 57–68. https://doi.org/10.1145/1960275.1960284

[9] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *23rd SSRE*. 221–230. https://doi.org/10.1109/ISSRE.2012.23

[10] K. Ludwig, J. Krüger, and T. Leich. 2019. Covert and phantom features in annotations: do they impact variability analysis?. In *SPLC*. ACM, 31:1–31:13. https://doi.org/10.1145/3336294.3336296

[11] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. 2013. Practical pairwise testing for software product lines. In *17th SPLC*. 227–235. https://doi.org/10.1145/2491627.2491646

[12] J. Meinicke, C. Wong, C. Kästner, T. Thüm, and G. Saake. 2016. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *ASE*. 483–494. https://doi.org/10.1145/2970276.2970322

[13] G. K. Michelon, W. K. G. Assunção, D. Obermann, L. Linsbauer, P. Grünbacher, and A. Egyed. 2021. The life cycle of features in highly-configurable software systems evolving in space and time. In *GPCE*. ACM, 2–15. https://doi.org/10.1145/3486609.3487195

[14] G. K. Michelon, D Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, and A. Egyed. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *SPLC*. ACM, 74–78. https://doi.org/10.1145/3382026.3425776

[15] G. K. Michelon, D. Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed. 2022. Evolving software system families in space and time with feature revisions. *Empirical Software Engineering* 27, 5 (May 2022). https://doi.org/10.1007/s10664-021-10108-z

[16] M. Mukelabai, D. Nešiundefined, S. Maro, T. Berger, and J. Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *ASE*. ACM, 155–166. https://doi.org/10.1145/3238147.3238201

[17] T. Schiex and S. de Givry (Eds.). 2019. *Principles and Practice of Constraint Programming*. LNCS '19, Vol. 11802. Springer. https://doi.org/10.1007/978-3-030-30048-7

[18] S. Souto, M. d'Amorim, and R. Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *ICSE*. IEEE, 632–642. https://doi.org/10.1109/ICSE.2017.64

[19] H. Tufail, M. F. Masood, B. Zeb, F. Azam, and M. W. Anwar. 2017. A systematic review of requirement traceability techniques and tools. In *2nd ICSRS*. 450–454. https://doi.org/10.1109/ICSRS.2017.8272863

[20] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *ICSE*. IEEE, 178–188. https://doi.org/10.1109/ICSE.2015.39