A Tool for Analysing Higher-Order Feature Interactions in Preprocessor Annotations in C and C++ Projects

David Korsman
Radboud University, Nijmegen
The Netherlands
david.korsman@ru.nl

Carlos Diego N. Damasceno
Radboud University, Nijmegen
The Netherlands
d.damasceno@cs.ru.nl

Daniel Strüber
Radboud University, NL
University of Gothenburg, SE
danstru@chalmers.se

ABSTRACT

Feature interactions are an intricate phenomenon: they can add value to software systems, but also lead to subtle bugs and complex, emergent behavior. Having a clearer understanding of feature interactions in practice can help practitioners to select appropriate quality assurance techniques for their systems and researchers to guide further research efforts. In this paper, we present pdparser, a Python-based tool for analysing structural feature interactions in software systems developed with C and C++ preprocessor. Our tool relies on a lightweight methodology to quantify the frequency of pairwise and higher-order feature interactions and the percentage of code affected by them. We showcase the individual characteristics brought forward by the automated analysis of one toy example and two open-source text editors: Vim and Emacs. The source code and a demo video are available on GitHub at HTTPS://GITHUB.COM/DKORSMAN/PDPARSER.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Feature Interaction, Product Lines, Static Analysis, Preprocessors

ACM Reference Format:
David Korsman, Carlos Diego N. Damasceno, and Daniel Strüber. 2022. A Tool for Analysing Higher-Order Feature Interactions in Preprocessor Annotations in C and C++ Projects. In *Proceedings of 26th ACM International Systems and Software Product Lines Conference (SPLC'22).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In variability-intensive systems, end-users and engineers typically have configuration spaces with thousands of features to choose [3]. Being able to customize programs by removing functionalities and adding only what is really needed can have benefits, such as making programs less complicated, saving storage space, or having a smaller attack vector. However, this also leads to additional complexity and concerns.

Even when there is a small number of options, the number of valid configurations can be so large that it becomes practically impossible to exhaustively build and analyze all product variants. Moreover, certain configuration selections may also expose situations where features conflict with each other or are required to work together when enabled at once. These situations are known as *feature interactions* [17]. Having a bigger numbers of features interacting can lead to additional problems where unexpected behavior may emerge from sets of three or more features jointly selected, namely *higher-order feature interactions.* For instance, a database system with features for collecting statistics, lock contention, and user authentication that can interact when *logging statistics about locks removed by administrator users.*

As a source of subtle bugs and emergent behavior, feature interactions are still an understudied topic. In the Linux kernel, empirical evidence has raised the awareness of its community about the fact that vulnerable functions have higher variability than non-vulnerable ones [10], the amount of potentially harmful but still untreated configuration-dependent warnings [14], and the impact of preprocessor blocks nesting on code reasoning, understandability and readability [9]. Having systematic support for reasoning about feature interactions in existing projects could be useful for researchers and practitioners.

In this paper, we present a Python-based tool for automated reasoning of structural feature interactions in preprocessor directives of programs written in C and C++. Using one toy example and two open-source text editors – Vim and Emacs, we experiment with pdparser to showcase its functionalities and suitability to analyze realistic C/C++ source code and handling syntactic peculiarities known to exist in the wild.

Our tool can be useful for both researchers and practitioners, especially those interested in software product lines [3]. Researchers can use our tool for empirical research, quantifying the nature of feature interactions "in the wild". The results of such research can foster further software analysis and testing techniques that are tailored to real-life projects (e.g., focusing either on pairwise or higher-order interactions). Furthermore, our tool can be used to identify projects suitable for benchmarking [19], in the sense that they contain a suitable (i.e., typical or particularly large) number of feature interactions. Practitioners can use our tool to understand the characteristics of feature interactions in their projects, allowing them to steer their efforts to improve design and evolution [6]. In particular, can use it to identify code sections that are error-prone due to high numbers of interacting features and deep nesting of code blocks, which should be carefully reviewed. Moreover, they can choose appropriate analysis and testing techniques tailored to the type of feature interactions in their projects.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC'22, 12-16 September, 2022, Graz, Austria
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

The rest of this paper is structured as follows. Section 2 discuss some preliminary concepts in feature interactions and preprocessors. Section 3 presents works related to ours and how our tool differs from them. Section 4 describes the functionalities and design of our tool. Section 5 presents an experiment performed to showcase the functionalities of our tool. We conclude this paper in section 6 with final remarks and ideas for future work.

2 BACKGROUND

Feature interactions have a certain design space [3], arising from the different programming languages (e.g., C, C++, Java), their supported feature implementation techniques (e.g., build systems, preprocessors, aspects), the different possible types of interactions (e.g., structural, control-flow-based), and their interaction orders (e.g., feature-wise, pairwise, and higher-order) [4]. In this design space, we focus on (i.) higher-order structural feature interactions in software (ii.) written in the programming languages C and C++ (iii.) via preprocessor directives (such as `#ifdef`).

Preprocessor directives in C and C++ code follow a relatively simple syntax, where a line in a source file either is a directive or is not a directive. These are defined in the C and C++ standards [1, 2]. If the first non-whitespace character on a line is a hash symbol (#), then it is a directive. The hash is followed by the instruction given to the preprocessor, for example `include` or `ifdef`.

Nevertheless, preprocessor directives can get more complicated than `#ifdef` and `#ifndef`. In fact, `#ifdef X` itself is shorthand notation for `#if defined(X)`. The `#if` directives function on a constant expression that can consist of logic, comparison, and certain operators like `defined`. The `#elif` and `#else` directives can also be used, and work as one might expect. A complex usage of preprocessors is shown in Listing 1.

Listing 1: Example of complex preprocessor directives

```
#if defined(__linux__) || defined(__unix__)
    /* Linux/Unix code... */
#elif defined(_WIN32)
    /* Windows code... */
#else
    /* Other code... */
#endif
```

In this example, we have the second block of code included if the following constraint holds: `!(defined(__linux__) || defined(__unix__)) && defined(_WIN32)`. Similarly, the third block will be included when `!(defined(__linux__) || defined(__unix__)) && !defined(_WIN32)` holds.

When parsing C and C++ files, our tool can extract the feature identifiers trivially from `#ifdef` and `#ifndef` directives, and from features checked by `defined` in `#if` and `#elif` directives. Our tool reasons about `#elif`/`#else` blocks and combines them into conditional expressions with logical operators as in the original code. Moreover, we invested a considerable engineering effort to address peculiarities of C code (e.g., comments and white spaces).

As of the time of writing, the GitHub platform alone has more than one million C/C++ repositories available [11]. Preprocessor directives are a widespread method for implementing feature variability in C and C++ programs [3]. Structural feature interactions [4] provide means to reason about interactions from the source code itself, as for interactions arising from nested `#if` and `#ifdef` blocks. Hence, they address threats to validity in analysing interactions dependent on execution aspects, such as input data.

3 RELATED WORK

The most important related work direction is on extracting variability information from preprocessor-based software implementations [12, 13, 20] and feature interactions [9, 10, 18]. A state-of-the-art tool in that direction is KernelHaven [13], which integrates four different code extractors for addressing this task, namely Undertaker [20], Code Block [13], srcML [7] and TypeChef [12].

Code extractor name	Input files	Implem. Lang.	Preproc. Block	Annot. AST	Quant. Report
Undertaker [20]	*.c, *.h, *.s	C++	■		
Code Block [13]	*.c, *.h, *.s	Java	■		
srcML [7]	*.c, *.h	Java		■	
TypeChef [12]	*.c	Java		■	
pdparser [5]	*.c, *.h, *.cpp, *.hpp, *.cxx, *.m, *.cc	Python	■		■

Table 1: Code Extractors in KernelHeaven [13] vs. pdparser

As summarized in Table 1, these tools differ in their supported input formats, implementation language, granularity (e.g., preprocessors in code blocks, abstract syntax trees annotated with variability information, quantitative analysis). Black cells indicate the characteristics present in a given code extractor. While these tools are fundamentally different, they all work on top of variability information from source code elements, such as preprocessors [13, 20] and presence conditions in abstract syntax trees [7, 12]. This information is typically enough for some types of analyses, as in anomaly detection. However, so far, they have not been used to characterize quantitative aspects of feature interactions. Our tool fills in this gap by supporting the characterization of feature interactions in C and C++ source code.

Moreover, due to Python's consistent grown in popularity [21], we chose to implement pdparser in Python to target a wider audience of software engineers and potentially data scientists. Our work improves upon the state-of-the-practice by lifting the capacity of extracting preprocessor annotations, feature interactions, and include guards to the Python ecosystem. Thus, data scientists will be able to incorporate variability information in their pipelines and build machine learning-based variability analysis technologies.

In the context of feature interactions, combinatorial testing offers one effective alternative for sampling products [15]. Pairwise and model-based test criteria have been previously investigated and shown capacity of achieving a higher coverage of feature interactions at a fraction of cost when compared to testing all feature interactions [16]. Deep nesting levels of preprocessors have been known to suggest feature interactions and lead to higher difficulties in code reasoningy [9]. Additionally, they have been associated with higher number of vulnerabilities [10]. For an overview on feature interaction, we refer the reader to the systematic mapping by Soares et al. [17].

4 DESCRIPTION OF THE TOOL

On a high level, our methodology consists of parsing the C/C++ source files of a project, extracting preprocessor directives within these files, and reasoning about these directives. In this section, we discuss the functionalities provided by our tool, its design, and typical syntax.

4.1 Tool Functionalities

Since our methodology does not have to "understand" C or C++ code, we designed the pdparser tool [5] purely based on *parsing preprocessors directives* rather than writing or extending a full C or C++ parser or compiler. Our tool provides a CLI where users have access to the following functionalities:

4.1.1 ***Analyze all source code files in a C/C++ project.*** Our tool enumerates and parses all files in one project directory that has a recognized extension related to C or C++ (i.e., `.c`, `.cpp`, `.h`, `.hpp`, `.cxx`, `.m`, `.cc`). The tool applies regular expressions to recognize preprocessor directives, and track which lines of code are subject to which conditions.

4.1.2 ***Analyze projects in batch mode.*** By default, our tool processes one project at a time. Then, for processing multiple projects, we provide a shell script that iterates over a list of projects and reports their individual results.

4.1.3 ***Export the results in JSON format.*** For each analyzed project, the tool returns JSON files with a rich set of information, including: lists of identifiers for features, `#include` guards and other `#defines`, the number of blocks and lines with each feature interaction order and at each nesting level, the time taken to analyze the code, and the total number of files and lines of code.

4.1.4 ***Visualize the variability information.*** The JSON files with the results can be further processed to visualize the results. Users can see the exact locations in the code where certain feature interaction orders and nesting levels were found, and generate bar plots for the frequency of feature interaction orders and nesting levels.

4.1.5 ***Check occurrence of features in multiple projects.*** From a set of results from different projects, users can create a file that centralizes all information and check the number of projects each feature appears in.

4.1.6 ***Project showcase and help menu for getting started.*** The pdparser tool includes a `test` project to showcase its functionalities. This test project has features with obvious names and nesting levels to ease manual inspection. Moreover, there is a help menu that lists all parameters available for usage.

4.2 Tool design

We designed our tool using the Python programming language, and the native python libraries for regular expressions (`re`), and arguments parsing (`argparse`). Our regular expressions were handcrafted in a iterative way with the support of several C/C++ projects that we analyzed and then had their results manually inspected.

The architecture of our tool is organized as follows: (1) The `pdparser.py` file provides a command line interface (CLI) for setting the source projects to be analyzed and indicating the sets of information to be captured from the code. (2) The `source_parser.py` file includes (2.a) the analyzer class for maintaining the parsing state as a stack of nested preprocessors, and the regular expressions that (2.b) parse different preprocessing annotations and (2.c) extracts features involved in a given preprocessor annotation. (3) The `count-project-features.py` script that centralizes all features extracted from a set of projects and the number of projects where each feature appears in. (4) The `run-all-projects.sh` script enables the analysis of multiple projects. (5) The `test/` directory to showcase and test the functionalities of the tool.

5 SHOWCASE EXPERIMENT: ANALYSING THE TEXT EDITORS VIM AND EMACS

To illustrate the work of our tool in analysing C/C++ source code, we performed an experiment with the showcase `test` example and two open-source projects of text editors: `Vim` and `Emacs`. In Table 2, we show the statistics about each subject analyzed, including the number of lines of source code (SLOC) in the whole project and the SLOC per file. Additionally, we report the time required to analyze each project and the throughtput in terms of SLOC analyzed per second. These results show that our tool can handle thousands of files at a scale of seconds.

Project	Hash	SLOC	$\overline{\text{SLOC/file}}$	Time (s)	Throughput
emacs	#8785d70	361,890	686.7	4.8	74,781.3
vim	#accf4ed	391,635	1,445.1	6.5	59,993.4
test	#ec86559	133	26.6	0.0	12,916.9

$\overline{\text{SLOC/file}}$: Average number of SLOC per file **Throughput**: SLOC per seconds

Table 2: Performance of the parser

5.1 Number of features and code within each nesting level

In terms of number of features involved, both projects have similar frequencies of feature interaction orders. In particular, HOFIs tend to occur at a lower rate when order interactions are considered up to level 10. In constrast, in the Emacs, we noticed that feature interaction with orders above 15 tended to increase again.

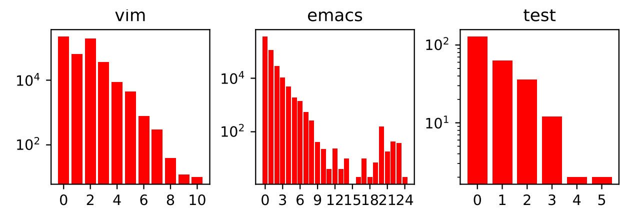

Figure 1: Number of features involved

When the amount of code surrounded by preprocessors is analyzed, we see that, for higher nesting levels, there is fewer conditional code. The frequency distribution is mostly the same in both text-editors.

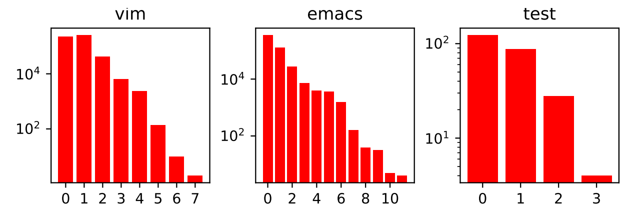

Figure 2: Number of lines at each nesting level

5.2 Syntax peculiarities found

Along this analysis process, we observed a few errors in the parsing process that helped to improve our tool. In most cases, these errors were humorous because we could see in which way the parser was misled. Sometimes, however, they were a legitimate issue in the source code of the studied projects. In particular, we discovered two syntactic peculiarities in the source code of the text-editors analyzed which we discuss in this section.

In the Vim project, we found the code fragment shown in Listing 2. Initially, this fragment deceived our parser by considering the string "/*" as the starting point of a block comment. After identifying this issue, we fixed our tool and successfully parsed the remainder of this project. This code fragment can be found in the Vim repository on GitHub [22].

Listing 2: Example with backslash character

```
static char *(except_tbl[][2]) = {
    {"*",           "star"},
    {"g*",          "gstar"},
    {"[*",          "[star"},
    {"]*",          "]star"},
    {":*",          ":star"},
    {"/*",          "/star"},
    {"/\\*",        "/\\\\star"},
    {"\"*",         "quotestar"},
    {"**",          "starstar"},
```

When parsing the source code of the Emacs editor, we did not find any parsing problems. However, we detected a code fragment using preprocessors in an inconsistent but harmless ways to the code behavior. In Figure 3, we show a snapshot for JSON file resulting from the Emacs project.


```
1  {
2      "unique_features_filtered": false,
3      "n_source_files": 527, "n_code_lines": 361890, "n_counted_lines": 521640, "n_
4      "ignored_exts": [".sl",".pif",".14",".de",".12",".gz",".bg",".eps",".tex",".
5      "n_possible_guards": 160,
6      "possible_guards": ["_Restrict_","_EmacsFrameP_h","NLIST_STRUCT","_GL_FTOAST
7      "n_other_defines": 135,
8      "other_defines": ["ADDRESS_SANITIZER_WORKAROUND","__BEGIN_DECLS","HELPER_LOC
9      "n_features": 1804,
10     "features": ["N_NAME_POINTER","DOUBLE_SLASH_IS_DISTINCT_ROOT","IEXTEN","HAVE_
11     "n_features_plus_guards": 1964,
12     "feature_interaction_blocks": {"0": 4303,"1": 5966,"2": 3673,"3": 2401,"4":
13     "nesting_level_blocks": {"1": 4243,"2": 3146,"3": 973,"4": 888,"5": 937,"6":
14     "feature_interaction_lines": {"0": 358788,"1": 114253,"2": 28792,"3": 10550,
15     "nesting_level_lines": {"0": 350619,"1": 126897,"2": 27739,"3": 7062,"4": 39
16     "time_taken": 4.839312424068339
17 }
```


Figure 3: Snapshot of the JSON file for the Emacs project

In the Emacs code, there is a file `test/manual/etags/c-src/h.h` which has an annotation `#else` without any corresponding `#if`. This file is also filled with meaningless fragments of code and test cases, and this is in a `test` folder which contains more source files in various programming languages that are also intended for testing. Emacs is a general-purpose text editor, so this occurrence can be easily explained. This code fragment can be found on the Emacs repository which is mirrored on GitHub [8].

Listing 3: Example of #else without corresponding #if

```
/* comment */ #define ANSIC
 #define ANSIC
 #else
typedef void (proc) ();
```

6 FINAL REMARKS

We present pdparse, a tool for analysing higher-order feature interactions in C and C++ source code. The source code and and a demo video or our tool are available on GitHub [5] so that others can reuse and repurpose it. As future work, we plan to assess the scalability of our tool in a larger set of open source projects.

REFERENCES

[1] ISO/IEC 14882:2020. 2020. *Programming languages – C++*. Standard.
[2] ISO/IEC 9899:2018. 2018. *Programming languages – C*. Standard.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer, Berlin, Heidelberg.
[4] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD '13)*.
[5] The authors. 2022. Online appendix for this tool paper submission at SPLC'22. https://github.com/dkorsman/pdparser.
[6] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. 2018. Analyzing a decade of Linux system calls. *Empirical Software Engineering* 23, 3 (2018).
[7] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *2013 IEEE International Conference on Software Maintenance* (2013-09). 516–519. ISSN: 1063-6773.
[8] Emacs. 2022. *Comment start deliminter in a string literal*. https://github.com/emacs-mirror/emacs/blob/98365c7b1e1e1d3d5f7185f2d4a2baa1c65b4540/test/manual/etags/c-src/h.h#L86
[9] Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. 2015. When code smells twice as much: Metric-based detection of variability-aware code smells. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*.
[10] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*.
[11] GitHub. 2022. *GitHub repositories implemented in C++*. https://github.com/search?q=language%3AC%2B%2B&type=Repositories&ref=advsearch&l=C%2B%2B&l=
[12] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: toward type checking #ifdef variability in C. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD '10)*.
[13] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: an open infrastructure for product line analysis. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 2 (SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 5–10.
[14] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wąsowski. 2016. A Quantitative Analysis of Variability Warnings in Linux. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-intensive Systems*.
[15] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). 196–210.
[16] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. 2011. Pairwise feature-interaction testing for SPLs: potentials and limitations. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*. Association for Computing Machinery, New York, NY, USA, 1–8.
[17] Larissa R. Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo S. de Almeida. 2018. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology* (2018).
[18] Stefan Strüder, Mukelabai Mukelabai, Daniel Strüber, and Thorsten Berger. 2020. Feature-Oriented Defect Prediction. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A (SPLC '20)*.
[19] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC '19)*.
[20] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems (EuroSys '11)*. 47–60.
[21] TIOBE. 2022. *TIOBE Index - The Python Programming Language*. https://www.tiobe.com/tiobe-index/python/
[22] Vim. 2022. *#else preprocessor without corresponding #if*. https://github.com/vim/vim/blob/9bd3ce22e36b5760a5e22e7d34d1bd6a3411258e/src/help.c#L329