Understanding Conditional Compilation Through Integrated Representation of Variability and Source Code

David Baum
Christina Sixtus
Lisa Vogelsberg
Ulrich Eisenecker
david.baum@uni-leipzig.de
wir13dvx@studserv.uni-leipzig.de
ges11eso@studserv.uni-leipzig.de
eisenecker@wifa.uni-leipzig.de
Leipzig University
Leipzig, Germany

arXiv:1908.08375v1 [cs.SE] 20 Aug 2019

ABSTRACT

The C preprocessor (CPP) is a standard tool for introducing variability into source programs and is often applied either implicitly or explicitly for implementing a Software Product Line (SPL). Despite its practical relevance, CPP has many drawbacks. Because of that it is very difficult to understand the variability implemented using CPP. To facilitate this task we provide an innovative analytics tool which bridges the gap between feature models as more abstract representations of variability and its concrete implementation with the means of CPP. It allows to interactively explore the entities of a source program with respect to the variability realized by conditional compilation. Thus, it simplifies tracing and understanding the effect of enabling or disabling feature flags.

CCS CONCEPTS

•**Human-centered computing** → **Visual analytics;** *Information visualization;* •**Software and its engineering** → *Maintaining software;*

KEYWORDS

conditional compilation, variablity, software visualization, visual analytics, Getaviz, preprocessor, software prodect line

ACM Reference format:
David Baum, Christina Sixtus, Lisa Vogelsberg, and Ulrich Eisenecker. 2019. Understanding Conditional Compilation Through Integrated Representation of Variability and Source Code. In *Proceedings of 23rd International Systems and Software Product Line Conference, Paris, France, 9–13 September, 2019 (SPLC'19),* 4 pages.
DOI: 10.1145/3307630.3342387

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'19, Paris, France
© 2019 ACM. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/3307630.3342387

1 INTRODUCTION

Conditional compilation is a way of introducing variability to C source code immediately before compile time. The CPP can be used to include or exclude source code components, which change the structure and behavior of the resulting program. Often Boolean feature flags are used to design complete SPLs. The complexity created by the numerous variants is challenging. Although feature models help to describe the variability, they are of limited use when working with the source code directly, e.g., during bug fixing. In general, bugs that lead to unwanted runtime behavior are often more difficult to detect and to fix than compile time errors. This applies even more if a bug only occurs under certain feature configurations. For this reason, the developer needs support for answering the following questions, that appear regularly during development: **Q1:** What effect does the activation of a feature have on the structure of a program? **Q2:** Which elements are contained in the source code given a certain feature configuration? With these questions in mind we developed an interactive analytics tool that provides the following functionality:

(1) It provides an overview over the structure of the system, i.e., all functions, global variables, and complex types that can be part of any variant.
(2) The user can define a set of flags and explore the structure of the resulting variant. This includes method calls, read and write operations, as well as the original C code.
(3) The analysis runs fully automated without any manual preparation steps.

A demo is available online[1]. Additionally, the usage of the tool is demonstrated in a screencast[2]. We first examine how existing tools support the presented use case, followed by a presentation of our tool. We will address several design choices, including variability extraction, and the visualizations the user interface is based on. A small application scenario based on the online demo is presented in chapter 5. Finally, we will discuss our previous experiences with the tool and future development.

[1] http://softvis.wifa.uni-leipzig.de/splc2019
[2] http://softvis.wifa.uni-leipzig.de/splc2019screencast

2 RELATED WORK

Most work on SPLs and variability either focuses on automatic checks at compile time or provides abstract models without a direct connection to the source code. In the area of C code refactoring numerous works can be found, that take preprocessor statements into account [4, 7, 15, 22, 25]. Feature models are often used to prepare the extracted information. Badros and Notkin have written a tool that analyzes unpreprocessed C source code with simple scripts [1]. The SPL community offers a number of tools for visualization and for better understanding variability points and variants. For example, two Eclipse plugins visualize feature models and perform type checking of preprocessor code [17, 23]. With the help of Meta Programming System (MPS) different views for editing and understanding SPL source code can be provided to developers [5]. Other tools generally support the development of SPLs without the need for specific focus on C source code. Feigenspan et al. have developed an Eclipse plugin that enables highlighting of feature code [6]. Nestor et al. have created visualizations for the configuration of SPLs [20], but they do not provide a direct connection to the source code. The Feature Relation Graph presents possible feature combinations depending on a selected feature [16]. Illescas et al. as well as Urli et al. show different visualization models for feature combinations but without a connection to the source code [9, 24]. Many works are based on the same extraction tools such as Feature-CoPP [12], SuperC [8], TypeChef [10], and Yacfe [21]. We are not aware of any tool that supports the presented use case satisfactorily. In the area of SPLs, the focus of research is on the representation of variability points and legal combinations of features. In most cases links to the underlying source code are not presented.

In contrast, some tools are aimed at improving the developer's understanding of the code. Livadas and Small have created an integrated development environment (IDE) extension that can be accessed by clicking a macro expansion. It shows where a macro has been defined and how the macro is expanded [14]. Also Kullbach and Riediger visualize macro expansion and conditional compilation with an IDE extension by so-called folding. When clicking on a preprocessor instruction, the corresponding precompiled source code is collapsed [13]. However, these tools are only useful for local contexts and do not address systemwide variability.

3 VARIABLITY EXTRACTION

Comprehensive preprocessing of the C code and the CPP statements is required to provide answers to the questions that have been raised. Our requirements on such a parser can be summarized as follows:

(1) The result of the parsing must contain all the linguistic means of the C standard. This includes translation units, functions, elementary types, complex types, information about function calls as well as reading and writing of global variables.
(2) The parser should consider the included files to handle declarations correctly.
(3) Macro expansions should be performed before parsing since the content of the macros may influence feature detection and location.
(4) The result of the parsing should contain information about the conditional compilation, including the CPP directives extracted from the source code. Even more useful would be an evaluation of nested conditions and an explicit representation of alternatives as distinct branches in the result.

There exist various tools with different scopes to analyze code with conditional compilation. We came to the conclusion that TypeChef meets our requirements best, although it is significantly slower than, e.g., SuperC. The goal of the developers of TypeChef was to create a complete and solid parser that can parse C code without manual preprocessing. It uses an LL parser to create an abstract syntax tree (AST) which contains all of the variability information we need. We modified TypeChef to serialize the complete AST to an XML file for further processing with jQAssistant. This is a program for analyzing and visualisizing software artifacts [18]. It is built on top of Neo4j, a graph database. We implemented a plugin for TypeChef to include C code and feature flags. The result is a graph containing all code entities, method calls, read and write accesses, features, and their dependencies.

4 USER INTERFACE

Getaviz [3] is an open source toolkit for visual software analytics [3]. It uses jQAssistant as information source and supports the automatic generation of visualizations for different use cases [2]. Getaviz comes with a highly configurable browser-based user interface for viewing and interacting with a visualization. Getaviz can be easily expanded to support new visualization types and interaction components. Hence, we used Getaviz as starting point and customized it to fit our requirements.

Figure 1 shows the default view containing a visualization of the structure (I), a search bar (II), the *FeatureExplorer* (III), and the *Code-Viewer* (IV). To understand the structure and the included variablity it is useful to get an overview of the complete system first. Therefore, we visualize the structure in such a way that it can be fully grasped at a glance. This view contains all code entities that could be potentially compiled. Our prototype is based on the Recursive Disk (RD) metaphor [19]. It is designed to visualize the structure of imperative programming languages, with an emphasis on object-oriented languages, especially Java. As the name indicates, an RD visualization consists of nested disks, where each disk represents a package or a class in Java. In order to apply the visualization to C code, we had to make several changes. We chose translation units as top level elements replacing packages. They are depicted as gray disks as shown in Figure 2. A translation unit can contain multiple structs, unions, enums, global variables, and functions. Functions are depicted as blue segments. The area of a blue segment is proportional to the lines of code of the corresponding function. Variables are depicted as yellow segments that have a fixed size. Structs, enums, and unions are depicted as purple disks. They can contain further elements according to the content of the C entities. We have retained the original layout algorithm. All disks are ordered by size and then placed spiral-shaped clockwise around the largest disk. Although at first glance it seems chaotic, the emerging visual patterns and empty spaces give each disk a unique appearance and help the user to recognize specific disks.

The visualization is interactive, so the developer can easily explore it. The *FeatureExplorer* contains all extracted feature flags of

[3] https://github.com/softvis-research/Getaviz

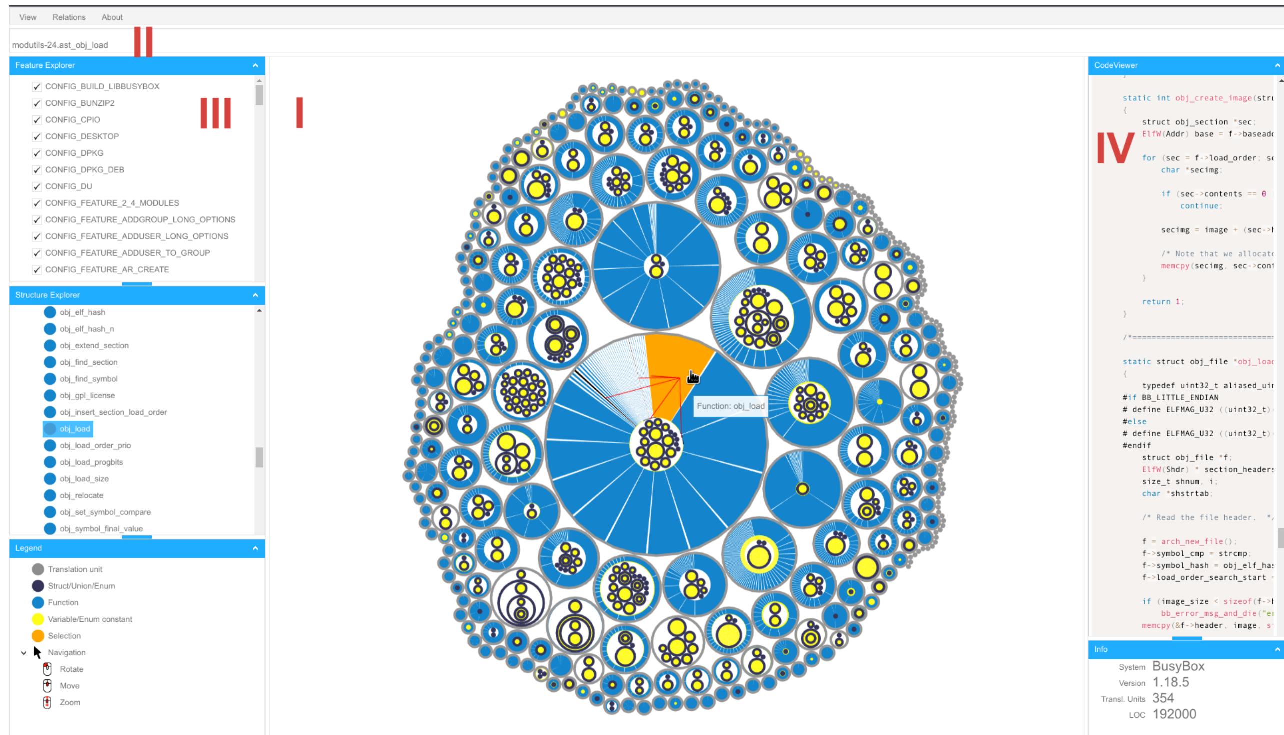


Figure 1: Screenshot of Getaviz visualisizing the structure of BusyBox

the system. The developer can select or deselect individual flags and the visualization gets updated accordingly. If the code entity is to be excluded by the CPP, then the graphical representation will be displayed transparently. In this way, the user can explore and understand the impact of the different flags to answer Q2 without having to jump from source file to source file and manually evaluate macros.

Detail information is provided as tooltip. In Figure 1, the method `obj_load` is selected and therefore highlighted orange. The red lines represent method calls and variable accesses of this method. Nevertheless, the source code is still of great interest for the developer since it is the main artifact to work with. To provide more context it is possible to view the source code directly in Getaviz. The *CodeViewer* on the right side displays the source code of the selected entity.

5 APPLICATION

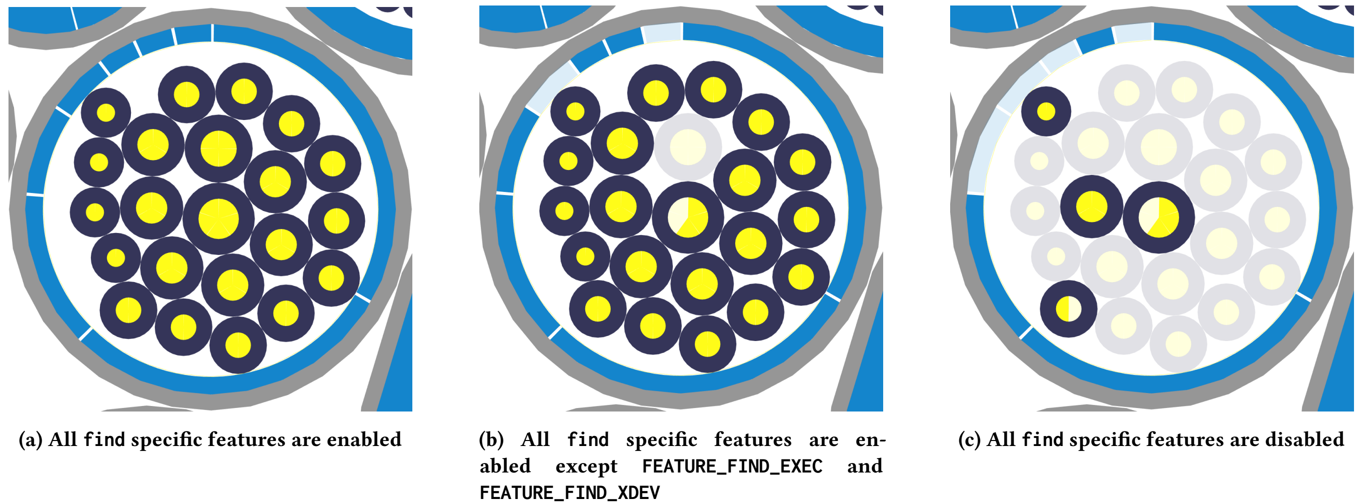
To demonstrate the usefulness of our tool we chose BusyBox 1.18.5 since it is a highly customizable system. It contains several hundreds of explicitly declared Boolean compile-time configuration options with complex dependencies [11]. One of these feature flags is `CONFIG_DESKTOP` which affects the macros `ENABLE_DESKTOP`, `IF_DESKTOP`, and `IF_NOT_DESKTOP`. They are used in 75 out of 354 translation units. Therefore, enabling this feature flag potentially changes behavior of more than 20% of the system in a variety of locations that can not be easily traced by the developer. Our tool takes the affected macros into account automatically and visualizes the influence of the feature flag on the structure with just one click.

Figure 2 shows the structure of the translation unit `find.c` with three different configurations. Our tool improves the developers understanding of the resulting structur and behavior by making the commonalities and differences explicit.

6 DISCUSSION

With our prototype we focus on the visualization design to support developers when exploring software systems which are making use of conditional compilation. We have therefore placed more emphasis on usability than on feature completeness. The application scenario demonstrates how the visualization supports the developer's usual workflow and eliminates time-consuming steps. Q2 can already be answered completely. Q1 can only be answered partially since not all feature locations are visually detectable. The necessary information is already available, but is not yet accessible in the user interface. For example, when selecting a feature, it would be possible to highlight all methods affected by the feature. It would also be helpful to support saving and loading configurations for subsequent analyses as well as comparing complete configurations visually. As soon as these features are implemented, we will compare our tool with existing solutions.

As for many software visualizations scalability is a critical point and necessary to use the tool in practice. The generation process took one day on a conventional notebook. We can visualize systems with up to four million lines of code without any problems. If the visualization becomes too complex, performance may decrease. However, the visualization framework still offers a lot of potential to improve performance to visualize larger systems.



(a) All `find` specific features are enabled

(b) All `find` specific features are enabled except `FEATURE_FIND_EXEC` and `FEATURE_FIND_XDEV`

(c) All `find` specific features are disabled

Figure 2: Visualizing the structure of BusyBox's "find.c" with three different configurations

7 CONCLUSION

Our tool supports the developer to explore variability implemented with CPP, especially in the context of SPLs. It simplifices tracing and understanding the effect of enabling or disabling these flags with respect to the code compiled subsequently. Thus, it bridges the gap between feature models and diagrams as more abstract representations of variability and its concrete implementation with the means of CPP. However, some features are still missing for use in practice that need to be addressed in future work.

REFERENCES

[1] Greg J. Badros and David Notkin. 2000. A framework for preprocessor-aware C source code analyses. *Software: Practice and Experience* 30, 8 (2000), 907–924.

[2] David Baum, Jens Dietrich, Craig Anslow, and Richard Müller. 2018. Visualising Design Erosion : How Big Balls of Mud are Made. In *IEEE VISSOFT 2018*.

[3] David Baum, Jan Schilbach, Pascal Kovacs, Ulrich Eisenecker, and Richard Müller. 2017. GETAVIZ : Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation. In *IEEE VISSOFT 2017*.

[4] Ira D. Baxter and Michael Mehlich. 2001. Preprocessor conditional removal by simple partial evaluation. In *Proceedings Eighth Working Conference on Reverse Engineering*. 281–290.

[5] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: projectional editing of product lines. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 563–574.

[6] Janet Feigenspan, Christian Kästner, Mathias Frisch, Raimund Dachselt, and Sven Apel. 2010. Visual support for understanding product lines. In *2010 IEEE 18th International Conference on Program Comprehension*. 34–35.

[7] Alejandra Garrido and Ralph Johnson. 2005. Analyzing multiple configurations of a C program. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 379–388.

[8] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12* (2012), 323. https://doi.org/10.1145/2254064.2254103

[9] Sheny Illescas, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Towards visualization of feature interactions in software product lines. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. 46–50.

[10] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. *ACM SIGPLAN Notices* 46, 10 (2011), 805. https://doi.org/10.1145/2076021.2048128

[11] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. *ACM SIGPLAN Notices* 47, 10 (2012), 773. https://doi.org/10.1145/2398857.2384673

[12] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: compositional annotations. (2016), 74–84. https://doi.org/10.1145/3001867.3001876

[13] Bernt Kullbach and Volker Riediger. 2001. Folding: An approach to enable program understanding of preprocessed languages. In *Proceedings Eighth Working Conference on Reverse Engineering*. 3–12.

[14] Panos E. Livadas and David T. Small. 1994. Understanding code containing preprocessor constructs. In *Proceedings 1994 IEEE 3rd Workshop on Program Comprehension-WPC'94*. 89–97.

[15] M. Vittek. 2003. Refactoring browser with preprocessor. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings*. 101–110. https://doi.org/10.1109/CSMR.2003.1192417

[16] Jabier Martinez, Tewfik Ziadi, Raul Mazo, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2014. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In *2014 Second IEEE Working Conference on Software Visualization*. 50–59.

[17] Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and B. ANDFONSECA. 2013. Colligens: A Tool to Support the Development of Preprocessor-Based Software Product Lines in C. In *Proc. áBrazilian Conf. áSoftware: Theory and Practice (CBSoft)*.

[18] Richard Müller, Dirk Mahler, Michael Hunger, Jens Nerche, and Markus Harrer. 2018. Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization. In *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018*. 107–111. https://doi.org/10.1109/VISSOFT.2018.00019

[19] Richard Müller and Dirk Zeckzer. 2015. The Recursive Disk Metaphor – A Glyph-based Approach for Software Visualization. In *Proceedings of the 6th International Conference on Information Visualization Theory and Applications (IVAPP '15)*. SciTePress, Setúbal, 171–176. https://doi.org/10.5220/0005342701710176

[20] Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. 2008. Applying visualisation techniques in software product lines. In *Proceedings of the 4th ACM symposium on Software visualization*. 175–184.

[21] Yoann Padioleau. 2009. Parsing C/C++ Code without Pre-processing. In *Compiler Construction*, Oege de Moor and Michael I Schwartzbach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 109–125.

[22] Diomidis Spinellis. 2010. CScout: A refactoring browser for C. *Science of Computer Programming* 75, 4 (2010), 216–231.

[23] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[24] Simon Urli, Alexandre Bergel, Mireille Blay-Fornarino, Philippe Collet, and Sébastien Mosser. 2015. A visual support for decomposing complex feature models. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. 76–85.

[25] Daniel G. Waddington and Bin Yao. 2005. High-fidelity C/C++ code transformation. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 35–56.