PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations

Elias Kuiter
Otto-von-Guericke-University
kuiter@ovgu.de

Sebastian Krieter
Harz University of Applied Sciences
Otto-von-Guericke-University
skrieter@hs-harz.de

Jacob Krüger
Otto-von-Guericke-University
jkrueger@ovgu.de

Kai Ludwig
Harz University of Applied Sciences
kludwig@hs-harz.de

Thomas Leich
Harz University of Applied Sciences
METOP GmbH
tleich@hs-harz.de

Gunter Saake
Otto-von-Guericke-University
saake@ovgu.de

ABSTRACT

The source code of highly-configurable software is challenging to comprehend, analyze, and test. In particular, it is hard to identify all configurations that comprise a certain code location. We contribute PCLocator, a tool suite that solves this problem by utilizing static analysis tools for compile-time variability. Using BusyBox and the Variability Bugs Database (VBDb), we evaluate the correctness and performance of PCLocator. The results show that we are able to analyze files in a matter of seconds and derive correct configurations in 95% of all cases.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software configuration management and version control systems**; *Software testing and debugging*;

KEYWORDS

Software Product Line, Configuration, Static Source Code Analysis, Preprocessor, Build System

ACM Reference Format:
Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. 2018. PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *22nd International Systems and Software Product Line Conference - Volume A (SPLC '18), September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3233027.3236399

1 INTRODUCTION

Software product lines are a systematic approach to manage and reuse software artifacts [2]. For this purpose, software artifacts are implemented with *features* that can be either present or absent in a concrete product. Each product corresponds to a *configuration* that specifies options for the presence or absence of each feature. If a configuration satisfies all feature dependencies (e.g., requires, alternatives), it is *valid* and a product can be derived.

A high number of configuration options, which may be scattered across different variability mechanisms, hampers the comprehension of source code, its analysis, and especially testing it. For example, Linux comprises over 10,000 configuration options that allow for millions of products [17]. Also, Linux' configuration options and their dependencies are implemented in a combination of the C preprocessor and Kconfig files, which alone comprise more than 110,000 lines of code. In particular, it is important in such a context to know which configurations comprise which code, resulting in the challenge defined by Gazzillo et al. [8]:

> "Given a specific program location in the source code, can you apply automatic analysis techniques to find concrete configurations that include the program location in question?"

Due to the complexity and large number of options in real-world systems, the naive approach to check each single configuration grows infeasible quickly. With this paper, we contribute the *Presence Condition Locator (PCLocator)*[1] tool suite, which solves the proposed challenge with appropriate effort. We use static analysis based on existing tools to analyze the program and its build system to find a presence condition for the specified code location. Using a variability model and a satisfiability problem solver, we can then derive valid configurations.

2 APPROACH

The challenge is focused on three systems, *axTLS*, *BusyBox*, and *Linux*. Each of these programs is implemented with C and is highly configurable, comprising between 94 and 14,000 configuration options. To implement this variability, two common compile-time mechanisms are used: The C preprocessor and build systems.

Analyzing Source Code Annotations. The C preprocessor is even used for fine-grained variability, such as single statements or literals [13]. Conditional directives (i.e., `#ifdef`) annotate the source code that will be removed if the corresponding configuration option is disabled. For boolean options, we can express this as a propositional formula, the *presence condition*. A crucial step in our approach is to extract presence conditions automatically for a given program location by parsing the code.

[1] Sources, results, and information available at: https://github.com/ekuiter/PCLocator

SPLC '18, September 10–14, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3236399

Conditional directives of the C preprocessor do not always align with the underlying abstract syntax tree. Such usage of the C preprocessor is referred to as *undisciplined annotations* [13] and complicates the task of parsing variability. We use a combination of existing tools that deal with this problem and enable parsing of undisciplined annotations—namely, TypeChef [9, 10], SuperC [7], and FeatureCoPP [11]—in order to receive an optimal result.

Analyzing the Build System. A build system addresses coarse-grained variability by determining files that are included in a product. Often, entire files are omitted if the corresponding configuration option is disabled. Thus, similar to source code, files have presence conditions as well, describing which configurations include them. We also extract the presence condition for each given file from the build system to ensure that the variability of each code location is completely represented.

For C/C++ programs, *make* and its corresponding *makefiles* are widely used as build system. However, analyzing makefiles is not trivial [4, 6]. In contrast, the given programs rely on *Kbuild* as build system, which has been analyzed with Kmax [6]. Thus, we integrate Kmax in our approach to extract a file's presence condition.

Deriving Configurations. Using the described tools, PCLocator is able to compute fairly accurate presence conditions for a code location and its file. It then joins these extracted presence conditions to create a single one for the code location. Next, it derives configurations that satisfy this condition.

Not all possible configurations of a system that include the given code location have to be valid. To address this issue, we require a variability model [3] that defines the valid configuration space. All example programs of the challenge rely on *Kconfig*, for which we may utilize tools used in previous work (e.g., by She et al. [16]). However, as this is not in the scope of the challenge, we assume that a variability model is already available for the system at hand.

Based on a variability model and the extracted presence condition, PCLocator employs a satisfiability problem (SAT) solver to compute a satisfying configuration. If there is at least one valid solution for the variability model in conjunction with the presence condition, the solver provides a corresponding configuration. Thus, our tool can derive multiple satisfying configurations by incrementally calling the solver and excluding previously found configurations, until it is stopped or has found all satisfying configurations.

3 TECHNIQUE AND IMPLEMENTATION

Our approach is to compose the results of the described static analysis tools for compile-time variability. This improves the reliability of the results (i.e., if one tool fails, we fall back to another one).

3.1 Supported Programs

In general, PCLocator can be used on any system that fulfills the following requirements, which includes the ones of the challenge:

(1) It is implemented in C.
(2) It uses the C preprocessor or Kbuild to implement variability.
(3) It uses the Kconfig language to model variability.

For evaluation, we focus on the provided C programs from the variability bugs database [1] (easy) and BusyBox toolkit (hard), according to the challenge. Although the axTLS web server uses Kbuild, it is not supported by Kmax, our used analysis tool. Thus, we dismiss it in favor of BusyBox. Similarly, we do not explicitly support Linux, because analyzing such a large system poses new implementation challenges regarding performance optimizations and handling large variability models. Nevertheless, BusyBox is a complex program with more than one thousand configuration options that is able to demonstrate the capabilities of PCLocator.

3.2 Input and Output

Our tool may take the following input artifacts:

- A code location (i.e., a source file and a line number).
- A variability model as DIMACS file[2].
- A Kmax presence condition file if analyzing Kbuild.
- Any additional options required for parsing a C file (e.g., include paths and platform headers).

Except for the program location, all other artifacts can be omitted if they are not available. However, when analyzing code that uses Kbuild, supplying a Kmax file increases the accuracy of the results. In addition, a variability model is required if configurations (not only presence conditions) shall be derived.

By default, PCLocator returns the presence condition for a given line in a source file. Optionally, it can also provide the presence conditions for all lines in the file. If a variability model is available, our tool derives a concrete configuration in the form of flags that can be passed to the compiler or as a configuration file (*.config*) that can be used to build Kconfig-based projects. As mentioned, incremental usage is possible to identify all valid configurations.

The accuracy of the results depends on the provided input. In most cases, SuperC and TypeChef should yield more reliable results than FeatureCoPP, as they consider included files and special cases, such as `#define` and `#undef` annotations. However, in order to work properly, SuperC and TypeChef require additional information (e.g., headers). Similarly, the resulting presence condition is more reliable if a Kmax presence condition file is provided and, thus, the variability of the build system can be considered.

3.3 Processing a Source File

The execution of PCLocator can be divided into five major steps, which we illustrate in Figure 1. First, the source file is parsed to compute the preprocessor presence conditions associated with each line. Second, the found presence conditions are refined to improve the accuracy. Third, all presence conditions of each parser are merged into a single presence condition. Fourth, the build system presence condition is computed and combined with the merged one. Finally, a configuration space is derived from the combined presence condition and the variability model.

Parsing the File. Processing a program location starts with parsing the file and locating presence conditions for each line. To this end, we integrate different parsers: TypeChef [9, 10] and SuperC [7] are variability-aware parsers that include nodes for conditional directives (*choice nodes*) in the file's abstract syntax tree (AST). FeatureCoPP [11] is intended to physically separate annotated features, but can also be used to analyze conditional directives and presence conditions. For each of these parsers we implemented a locator,

[2]DIMACS files can be obtained for example with Kconfig Reader or LVAT [5].

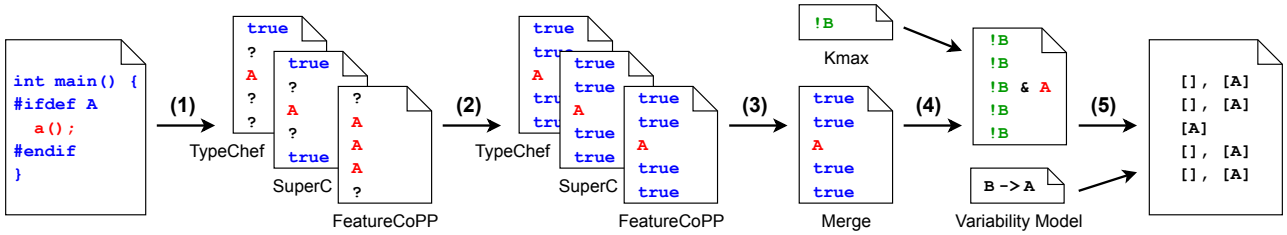

Figure 1: Processing a source file: (1) Parsing, (2) refining, (3) merging, (4) build system analysis, (5) configuration space analysis.

wherefore they can be used individually to return presence conditions for a file. Though, as we display in Figure 1, we recommend to combine them in the merging step.

TypeChef and SuperC create an AST containing line information for each node. PCLocator then builds a presence condition by traversing up through the AST and processing all encountered choice nodes, which represent a nested `#ifdef`. FeatureCoPP works differently: It does not build an AST of the source code, but analyzes only the structure of conditional blocks. Furthermore, `#include`, `#define`, and `#undef` directives are excluded when using FeatureCoPP. Both, TypeChef and SuperC provide a preprocessor-only mode that yields a conditional token stream without building a full AST. Using this mode, many parsing issues can be avoided, resulting in improved correctness and performance.

Refining the Results. The yielded presence conditions are refined based on the following steps:

(1) Header files, such as *stdio.h*, are mocked out, because, in general, they do not contribute to software variability. This speeds up the parsing step and decouples presence conditions from system-specific header files.
(2) For conditional directives, such as `#ifdef A`, we consider its outer scope as presence condition (i.e., `true` if `#ifdef A` is at top-level). However, the parsers return inconsistent results for directives (i.e., some parsers would consider `A` to be the directive's presence condition). Thus, we ignore presence conditions of conditional directives.
(3) Currently, the parser step does not locate presence conditions for all lines in a file, because some lines (e.g., empty ones) do not contain any tokens that appear in an AST or token stream. There is a simple way to deduce presence conditions for such lines though: In a block of non-conditional lines enclosed by conditional directives, there is only one distinct presence condition. Assuming we already know the presence condition for one line from the parser, we can deduce all other presence conditions in the block. Thus, we can also deduce presence conditions for conditional directives (which were ignored before) from surrounding lines. Also, every line not nested in any conditional block has the presence condition `true`. We can extend this deduction algorithm to nested conditionals by using a stack to track the presence conditions for nested blocks.

After this step, we still have three sets of presence conditions.

Merging the Results. The question arises, whether these results can be combined to produce a presence condition that has higher accuracy. A simple way to merge presence conditions, implemented in our tool suite, is to choose the "most appropriate" presence condition for each line. We choose an *appropriate presence condition* based on the following rules:

- Results from TypeChef and SuperC are preferable to results from FeatureCoPP, because the former consider included files and all preprocessor directives. FeatureCoPP only does a lexical analysis to determine the result and does not parse the actual C code, which makes it a very reliable fall-back.
- If two parsers locate presence conditions `A` and `B` for the same line and `A` implies `B` (i.e., `A` is a specialization of `B`), we argue that `A` is preferable to `B` whenever we are interested in deriving a configuration that includes the line with high confidence. Note that this way we trade accuracy of the presence condition for a more reliable configuration.
- If no presence condition implies another one, there is no easy way of determining a good result and chances are high that the results of TypeChef and SuperC are not accurate. Thus, we fall-back to FeatureCoPP, if possible.

Overall, if TypeChef and SuperC compute equivalent presence conditions, the result is likely to be correct.

Analyzing the Build System. In this step, PCLocator analyzes the Kbuild-system variability using the Kmax tool to compute the presence condition for the entire file. If no build system is used or only one file is considered (e.g., in VBDb), this step is skipped. Afterwards, our tool combines the results of Kmax with the previously obtained preprocessor presence conditions. Thus, both types of variability are captured and the resulting presence condition represents all configurations that include the particular code location, assuming the chosen parser and Kmax produced correct results.

Deriving a Configuration Space. Finally, PCLocator derives concrete configurations from the combined presence conditions if a variability model is provided. For TypeChef and SuperC, we employ the SAT solver Sat4j [12], which is integrated in TypeChef. As FeatureCoPP can derive values for non-boolean configuration options, we use the constraint satisfaction problem solver Choco [15]. Both solvers return a set of configuration options that satisfies the presence condition and variability model. By repeatedly calling a solver we can obtain more configurations.

4 EVALUATION

In this section, we present a first evaluation of PCLocator. To this end, we rely on two data sets: First, we use the 56 code locations provided by the challenge [8] for the variability bugs database (VBDb) [1]. Second, we use BusyBox, for which we sample 120

Table 1: Performance and correctness for using PCLocator.

	Merge	TypeChef	SuperC	FeatureCoPP
VBDb: 56 given locations				
Time (s)	40.01	42.53	44.62	36.52
Median	0.71	0.76	0.80	0.66
Min	0.62	0.63	0.25	0.10
Max	0.83	0.87	0.90	0.77
Correctness	96%	96%	96%	98%
Precision	96%	98%	96%	98%
Recall	100%	98%	100%	100%
BusyBox: 120 random locations				
Time (s)	277.95	244.60	173.12	110.46
Median	2.35	2.07	1.43	0.90
Min	1.28	1.26	1.00	0.78
Max	3.37	2.81	2.11	1.25
Correctness	95%	89%	95%	94%
Precision	95%	96%	95%	94%
Recall	100%	93%	100%	100%

random code locations. For BusyBox, we use an existing variability model from prior work [14]. We run PCLocator for each parser individually and in merge mode, which we refer to as *Merge* (cf. Figure 1), to get a single configuration for each location. Regarding correctness, we analyze all code locations in VBDb and all locations in our BusyBox sample that do not have the trivial presence condition `true` (i.e., 100 of 120). We manually check whether a given program location is included in the derived configuration to evaluate the correctness. We measure the computation time of our tool using a quad-core 2.3 GHz computer with 12 GB of RAM.

Time. The given program locations in the VBDb can be analyzed in a matter of seconds using the merge parser. Similarly, analyzing a random BusyBox code location with the merge parser takes about 2 seconds (median and mean). Analysis time differs somewhat for the different parsers: As TypeChef's implementation is not optimized for performance, it is 0.6 seconds slower than SuperC. Also, FeatureCoPP has the shortest analysis time because it only considers conditional directives on a lexical level. The merge parser executes all three tools in parallel; thus its analysis time roughly equals the time of the slowest individual parser.

Correctness. For correctness, we calculated precision and recall as defined in the challenge: Precision refers to how many of all successfully identified configurations include the given code location, while recall denotes how many of all files could be successfully parsed. For VBDb, we find that all measures have similar values (96%–98%), which—given the small sample size—is hard to compare. Most importantly, we find that none of the parsers can identify configurations flawlessly, which seems to justify our merging approach. For BusyBox, we find that our merge parser and SuperC performed best (95%), while TypeChef failed to parse multiple files, resulting in lower recall (93%) and therefore correctness (89%). In particular, FeatureCoPP performed reasonably well—considering that it ignores `#include`, `#define`, and `#undef` directives.

Mode of Operation. As mentioned in Section 3.3, TypeChef and SuperC have two major modes of operation: Initially, PCLocator was running these parsers using their default mode (building a variability-aware AST of the source file). For many static code analyses, an AST is desirable, but for the given challenge a token stream (annotated with presence conditions and code locations) suffices. Using this second mode of operation should have a positive impact on performance and correctness, because building a variability-aware AST for C code is time-consuming and error-prone. We updated PCLocator accordingly, and the results we show in Table 1 have already been obtained using the conditional token stream. For comparison, we also ran SuperC and TypeChef in their default mode using the same experimental setup: As expected, we found that, for BusyBox, analysis time rises to a median of 7 seconds (10 seconds mean, 87 seconds maximum) using the merge parser. In addition, TypeChef and SuperC regularly fail to build an AST at all in this mode due to unrecognized syntax and unresolved types (79% recall). These results confirm that, for this challenge, the token stream approach is superior to building ASTs.

Summary. The results show that all parsers perform similarly on the given data sets when run in token stream mode, with minor differences in time and correctness. No parser identifies configurations perfectly, which follows from the mutual strengths and weaknesses of the parsers: TypeChef and SuperC can handle more conditional directives than FeatureCoPP, while FeatureCoPP can derive configurations including non-boolean configuration options. In particular, our merge approach does not perform much better or worse than the comprised parsers. Nonetheless, it is useful to derive a configuration in the presence of parser failures and does not add much overhead to the required execution time. In addition, it can be used to assess the confidence of a presence condition: For BusyBox, we find that TypeChef and SuperC located equivalent presence conditions in 88% and disagreed only in 3% of all cases. We therefore regard the merge parser as the safest choice in terms of presence condition accuracy.

5 CONCLUSION

In this paper, we introduced PCLocator, a tool suite to identify configurations that comprise a given code location. To this end, we use three different parsers and refine their results with build system information and a variability model. Our evaluation shows that the different parsers perform similar despite their differing analysis approaches. In particular, our merge approach returns quite correct results within a reasonable time. Considering the correctness and performance of PCLocator, we are optimistic that it can solve the defined challenge.

We see the current version of PCLocator as a first step to provide a correct analysis of presence conditions. In future work, we aim to improve and extend it by adding possibilities for further input types. Especially, we want to further increase its correctness and performance. To this end, we may rely on additional tooling and implement analysis strategies ourselves.

Acknowledgments. Supported by DFG grants LE 3382/2-1, LE 3382/3-1, SA 465/49-1, and Volkswagen Financial Services AG.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *International Conference on Automated Software Engineering*. ACM, 421–432.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[3] Lianping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology* 53, 4 (2011), 344–362.

[4] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *International Software Product Line Conference*. ACM, 21–30.

[5] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig Semantics and Its Analysis Tools. In *International Conference on Generative Programming: Concepts and Experiences*. ACM, 45–54.

[6] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Joint Meeting on Foundations of Software Engineering*. ACM, 279–290.

[7] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 323–334.

[8] Paul Gazzillo, Ugur Koc, ThanhVu Nguyen, and Shiyi Wei. 2018. Localizing Configurations in Highly-Configurable Systems. In *International Systems and Software Product Line Conference*.

[9] 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), 805–824.

[10] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *International Workshop on Feature-Oriented Software Development*. ACM, 25–32.

[11] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development*. ACM, 74–84.

[12] Daniel Le Berre and Anne Parrain. 2010. The Sat4j Library, Release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), 59–64.

[13] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *International Conference on Aspect-Oriented Software Development*. ACM, 191–202.

[14] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Joint Meeting on Foundations of Software Engineering*. ACM, 81–91.

[15] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2017. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S.

[16] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. In *International Workshop on Variability Modelling of Software-Intensive Systems*. Universität Duisburg-Essen, 45–51.

[17] 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 *Conference on Computer Systems*. ACM, 47–60.