Handling Static Configurability in Refactoring Engines

Jörg Liebig[1], Sven Apel[2], Andreas Janker[2], Florian Garbe[2], and Sebastian Oster[1]

[1]Method Park Consulting GmbH, Germany
[2]University of Passau, Germany

Abstract

Which refactoring engine do you use in your C project? Although most systems written in C are statically configurable with the preprocessor (e.g., using the #ifdef directive), contemporary commercial, open-source, and research refactoring engines still do not support preprocessor directives well, if at all. As a result, even simple refactorings, such as RENAME IDENTIFIER, may introduce errors in some variants of the system to be refactored. In this article, we report on our experience with making refactoring practical (i.e., scalable, sound, and complete) in the presence of static configurability.

1 Refactoring Configurable Systems

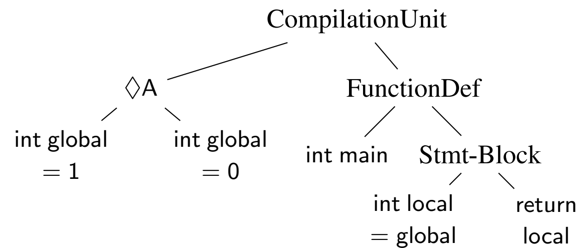
Refactoring is an important activity in software engineering. The goal is to improve the internal structure of source code, while preserving its external behavior. Refactoring engines belong to the standard equipment of software developers. A refactoring engine automates the code transformations involved in refactoring (e.g., moving code) and thus simplifies the development process for developers as well as improves their productivity. Classic examples of refactorings are RENAME IDENTIFIER (renaming an identifier consistently), EXTRACT FUNCTION (encapsulating code in a new function and replacing the old code with a function call), and INLINE FUNCTION (replacing a function call by introducing the function's code at the call side). These three refactorings are among the most frequent used refactorings in software engineering.

Although refactoring engines are useful, there are problems regarding their applicability in practice [11]. One important problem that we address here is refactoring in the presence of static configurability (e.g., implemented with the C preprocessor). The background is that most software systems are configurable and provide configuration options to support different platforms and use cases. End users configure such systems by setting values for configuration options, which control the inclusion and the exclusion of optional and alternative code in the code base, giving rise to different system

variants. Many systems today are statically configurable, including the LINUX kernel, which provides more than 10 000 configuration options, supporting various hardware platforms, devices, and system features. Implementing a system to be configurable, developers often use simple mechanisms, such as preprocessor directives, as provided by the C preprocessor CPP. CPP provides directives for textual substitution (#define), file inclusion (#include), and conditional compilation (e.g., #ifdef, #if, #else, and #endif). Let us illustrate CPP's capabilities by the example of a configuration-dependent declaration of a program variable, as illustrated in Figure 1a. Depending on selecting or deselecting configuration option A, variable global is initialized with a different value, giving rise to the generation of two different system variants: one returning 1 and one returning 0. In what follows, we concentrate on static configurability with preprocessor directives, as they are widely used in practice [7].

```
1  #ifdef A
2  int global = 1;
3  #else
4  int global = 0;
5  #endif
6
7  int main() {
8    int local = global;
9    return local;
10 }
```

(a) Configuration-dependent declaration of variable global



(b) Simplified abstract syntax tree (AST); ◊ represents variability induced by preprocessor directives in the code

Figure 1: Example of a configuration-dependent declaration in C using #ifdefs

Refactoring always entails the risk of introducing errors into working source code [11]. In the context of configurable systems, this risk is even more serious as the development of configurable systems relies on developing reusable code artifacts. The idea is that artifacts are shared across multiple system variants. This sharing implies that code transformations involved in a refactoring may not only affect a single, but multiple system variants. For example, renaming variable global on Line 8 in Figure 1a requires that both declarations of global, in the configurations A and $\neg$A, have to be renamed. Such configuration-dependent code is common in practice [6], and many legacy software systems written in C make extensive use of preprocessor directives [7]. Refactoring engines that do not take static configurability (properly) into account will likely produce erroneous code, which is not only a theoretical problem. For example, the popular development tool XCODE (`https://developer.apple.com/xcode/`) would miss to consistently rename both declarations of variable global, leading to a compile error in one system variant (option A is selected), and so do other tools (as we will explain in Section 2).

In general, refactorings for configurable systems have to ensure behavior preservation for *all* system variants [1]. That is, static configurability needs to be incorporated at all levels of the refactoring process: in the static analysis that determines required information to perform the refactoring, in precondition checks to ensure behavior preser-

vation, and in transformations to restructure the code properly.

This article builds and reflects on a previous conference paper [8], which provides technical descriptions of refactorings in the presence of static configurability. Here, we take a practical view and share our experience with making refactorings practical in the presence of static configurability.

Next, we discuss different strategies to handle static configurability proposed in the research literature or implemented in state-of-the-art tools.

2 The State of the Art

Refactoring of C code in the presence of preprocessor directives is not a new problem. There are various tools that exploit different strategies on the refactoring challenge. To assess the different strategies, we conducted an empirical study on the capabilities of contemporary refactoring engines. Our goal was to understand and classify the engines' operation principles and to compare refactoring capabilities and limitations, when using these engines on source code with C-preprocessor directives. For the purpose of our investigation, we selected 18 different refactoring engines (academic prototypes, industry-proven tools, and open-source developments) and analyzed their handling of static configurability. In a first step, we looked at technical descriptions, such as conference papers, articles, documentation, specifications, and online presentations. Based on the results of this step, we created code snippets containing preprocessor directives, such as the one in Figure 1, and applied standard refactorings, including RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION, using the tools under investigation. We collected the results of applying the individual refactorings and checked behavior preservation (before and after refactoring) by manual code inspections and tests. Based on these results, we classified all engines regarding their handling of static configurability, as summarized in Table 1.

2.1 Four Ways to Handle Static Configurability

As a result of our investigation, we identified four different types of refactoring engines. We skip engines that provide only textual find and replace functionalities via standard editor services here and only list them in the result table for completeness.

Single Variant Many refactoring engines do not support static configurability (e.g., ECLIPSE CDT). A common strategy is to derive a single system variant, apply the refactoring to it, and transfer the result to the original source code. For the derivation process, the engines usually use a default system configuration, provided by the developer. Conceptually, these engines simply ignore that there are, in fact, multiple system variants, as in Figure 1, and they will most likely introduce compile errors and behavior deviations.

Variant-based Some refactoring engines use a variant-based strategy (e.g., CSCOUT), following a similar operation principle as engines without static-configurability support. But, instead of a single, default configuration, these engines derive all system variants affected by the refactoring, apply the refactoring to each variant in isolation, and merge

the individual results back to the original source code. The identification of affected configurations is a semi-automatic process, in which developers need to specify affected system configurations. However, specifying system configurations requires extensive knowledge about the system in question (e.g., the build system) and is an error-prone task. Handling multiple system variants individually does not scale, as a single refactoring may affect myriads of configurations, for which the specification as well as the merge operation is problematic.

Limited Patterns In contrast to the engines discussed so far, which only operate on individual system variants, there are refactoring engines that incorporate static configurability directly (e.g., DMS). For this purpose, source code and preprocessor directives are directly represented in data structures and algorithms inside the refactoring engine. For example, such engines represent preprocessor directives in the source code with specific kinds of AST nodes (see Figure 1b), which are then considered by the refactoring engine during code analysis and transformation. Nevertheless, these engines are typically limited to common usage patterns of preprocessor directives (e.g., to directives around entire statements, expressions, etc.), for which they handle code transformations properly. Arbitrary preprocessor directives (e.g., annotating a function parameter or a `case` block) must be transformed (e.g., annotating the entire function or the complete `switch` statement) before refactoring. This requires significant amount of work, which has to be done manually by developers, since proper tool support is still not available [9].

Heuristics The last strategy of handling static configurability is an improvement of the third strategy. To overcome the limitation of supporting only a few specific preprocessor-usage patterns, some refactoring engines, such as CREFACTORY, employ heuristics to be able to reason about arbitrary preprocessor directives. Internally they try to automatically map arbitrary preprocessor directives to patterns they support. In fact, engines pursuing this strategy trade soundness in favor of completeness. As a result, heuristics-based engines will fail in corner cases that the developers of the refactoring engines did not foresee, and it is not unlikely that errors will occur.

In Table 1, we provide an overview of the results of our investigation. Note that the table is in itself a valuable foundation for developers and researchers to pick the right tool for their tasks at hand. The central observation of our investigation is that most refactoring engines used in practice (industry-proven tools and open-source developments) lack any support for static configurability. Particularly, engines that are used widely provide usually only facilities for simple textual find and replace or operate on single, default system variants, for which they apply refactorings properly. This is an irony as especially in practical systems static configurability is pervasively used. Academic tools are more sophisticated, but they are still not sound (i.e., they employ heuristics to reason about static configurability), not complete (i.e., they do not support all usages of preprocessor directives), or not scalable (i.e., they do not scale to practical systems with possibly billions of system variants). Nevertheless, these three properties are basic requirements for refactoring engines in practice.

Table 1: Classification of refactoring engines with regard to the handling of static configurability

Refactoring Engine	**Version Reference**	**Find/Replace** No Proper Refactoring Support	**Single Variant** Neglecting Multiple System Variants	**Variant-based** Not Scalable	**Limited Patterns** Limited Applicability	**Heuristics** Unsound
COCCINELLE	[12]					✓
CODE::BLOCKS	10.05	✓				
CODELITE	2.8.0	✓				
CREFACTORY	[4]					✓
CSCOUT	[14]			✓		
DMS	[2]				✓	
ECLIPSE CDT	8.2.1		✓			
GEANY	0.21	✓				
GNAT GPS	5.0-6	✓				
JETBRAINS CLION	1.0.2		✓			
KDEVELOP	4.3.1	✓				
MONODEVELOP	2.8.6.3	✓				
NETBEANS IDE	7.4		✓			
PROTEUS	[16]			✓		
PTT	[13]				✓	
VISUAL STUDIO	2013 Prof.	✓[1]				
XCODE	5		✓			
XREFACTORY	[15]			✓		

[1] by default, support only with one of several, proprietary extensions

3 Taming Static Configurability with Variability-Aware Refactoring

To overcome the limitations of existing refactoring engines, we developed our own strategy to the refactoring challenge: *variability-aware refactoring*. We implemented a corresponding tool named MORPHEUS, which is available from `http://fosd.net/morpheus/`. In the following, we describe the architecture of MORPHEUS, share our experience with its development, and the obstacles we had to clear. We aimed at addressing all issues of existing refactoring engines that we identified in Section 2:

- handling arbitrary preprocessor directives,
- treating variability explicitly, and
- scaling refactoring to real-world configurable systems.

Although MORPHEUS is still under development, we were able to make significant progress on the way to make refactoring practical in the presence of static configurability.

3.1 Handling Arbitrary Preprocessor Directives

Refactoring engines typically operate on abstract program representations, such as ASTs, which they use during the preparation and realization of refactorings. The major challenge of handling static configurability is to create a correct and efficient representation of source code to support analysis and transformation. The two most promising strategies revealed by our empirical investigation (strategy 3 and 4) incorporate static configurability directly in the AST. That is, they employ static-configurability information in nodes of the AST representation, on which they work. Figure 1b shows such a representation; the alternative declaration of global is represented with a node (◇) denoting a static choice. Unfortunately, not all preprocessor directives respect the syntax of the host language and can be represented easily as AST nodes. Applying manual transformations (as in strategy 3) and applying heuristics (as in strategy 4) to handle these *undisciplined* directives are both insufficient solutions. Since arbitrary preprocessor directives are common [9], a manual approach is infeasible in practice. Heuristics may be feasible, but they do not guarantee behavior preservation.

Parsing C code with arbitrary preprocessor directives is known to be hard, and there are only two parsers that are able to create sound and complete AST representations: TYPECHEF [6] and SUPERC [5]. Both parsers expand arbitrary preprocessor directives using code duplication, to align them with the abstract syntax of AST [9]. At the same time, #include and #define directives are resolved, so the remaining code includes only C language constructs and preprocessor directives controlling conditional compilation. The reason for this resolution step is that #include and #define directives may manipulate the input source code, which makes it practically impossible to create a sound and complete AST without resolving them. We have been building MORPHEUS on top of TYPECHEF, because it offers more features than SUPERC, including type checking.

3.2 Representing and Reasoning about Configuration Knowledge

Static configurability is not limited to preprocessor directives in the source code of a system. While preprocessor directives enable fine-grained control over the inclusion of source code in a system's code base, developers use further mechanisms for the specification of static configurability, often closely tied to preprocessor directives. For example, the developers of the LINUX kernel built a whole framework, called KBUILD, around the MAKE utility for this purpose. By using configuration options in makefiles, developers make the build process itself configurable. That is, configuration options in build scripts control the inclusion and exclusion of source-code files (by adding and removing build targets and changing build dependencies).

Beyond configuration options in makefiles and preprocessor directives, variability models are commonly used to specify the *valid* configurations of a configurable system, in terms of dependencies and constraints among the configuration options. The LINUX kernel developers invented a domain-specific language for the specification of valid kernel configurations (KCONFIG). By using KCONFIG, they specify configuration options, their dependencies using configuration logic, help messages, and default values. KCONFIG is linked to the kernel build system and source-code files. That is, configuration options defined in KCONFIG may occur also in makefiles and in preprocessor directives. The combination of KCONFIG, KBUILD, and preprocessor directives forms the kernel's configuration knowledge, which ensures that only valid kernel variants can be derived (Figure 2).

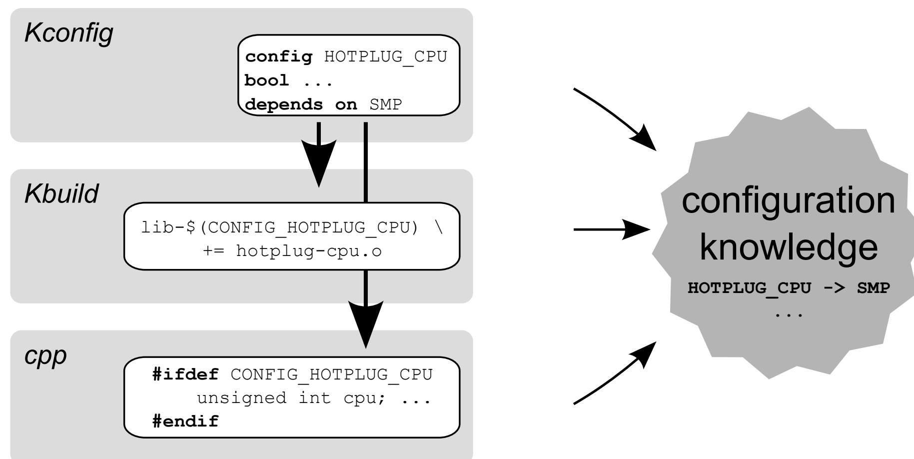


Figure 2: Excerpt of configuration knowledge in LINUX by the example of support for hot-plugable CPUs: specification of configuration option HOTPLUG_CPU in KCONFIG (top), configuration-dependent build rule in KBUILD (middle), and CPP-preprocessor directives in source code (bottom)

Developers of other projects, such as BUSYBOX, adopted KBUILD and KCONFIG in their project to support static configurability. Other projects rest on home-grown solutions to support static configurability. One example is OPENSSL, in which configuration support is encoded in a configuration script. Based on selected configuration options

passed as command-line parameters, the script deduces values for the remaining options.

All these different sources of configuration knowledge make use of different formalisms and representations to encode static configurability. To unify them and to make them usable for our purposes, we encode configuration knowledge in MORPHEUS with propositional formulas and a SAT solver [1]: Boolean variables represent configuration options, and logical operators encode dependencies between options. Let us assume a configurable system provides a set of configuration options $o_1, \ldots, o_n$. Then, the set of valid system configurations can be given in terms of a Boolean formula Φ over the system's configuration options. For example, the Boolean formula characterizing the constraint that configuration option SMP of Figure 2 must be activated when configuration option HOTPLUG_CPU is activated is HOTPLUG_CPU $\Rightarrow$ SMP.

During a refactoring's code analysis and transformation, many different configuration-related questions that are concerned with certain program elements need to be answered: In which system configurations is file lzop.c part of the compilation process? We answer this and other questions by reasoning about presence conditions. A presence condition $\mathcal{PC}$ maps program elements to a Boolean formula that defines all configurations in which the program elements are present. Using the declaration of variable cpu of Figure 2 as an example, we get $\mathcal{PC}(\mathsf{cpu}) = \mathsf{HOTPLUG_CPU}$. To check whether declaration cpu is part of, at least, one valid system variant, we make a satisfiability check using a SAT solver: SAT($\Phi \wedge \mathcal{PC}(\mathsf{cpu})$) . The SAT solver determines if there is any configuration c to satisfy the given Boolean formula. If so, the resulting configuration c contains a valid assignment of the configuration options $o_1, \ldots, o_n$ to $\mathbf{true}$ and $\mathbf{false}$ values, making up again a Boolean formula ϕ, for example, $\phi_c = o_1 \mapsto \mathbf{true} \wedge o_2 \mapsto \mathbf{false} \wedge \ldots$

Using presence conditions, we can answer a wide array of configuration-related questions including whether the target of a function call is correct in any valid system configuration, SAT($\Phi \Rightarrow (\mathcal{PC}(caller) \Rightarrow \mathcal{PC}(callee))$), or whether two program elements, e_1 and e_2, are in no configuration present together, SAT($\Phi \Rightarrow \neg(\mathcal{PC}(e_1) \wedge \mathcal{PC}(e_2))$).

Although answering these questions can be computationally hard (NP-complete), we can reason about these questions efficiently by using standard SAT solvers. In our experiments, we found that many configuration-related questions reoccur during code analysis and transformation. To increase performance, MORPHEUS and TYPECHEF store answers to configuration-related questions in a cache to serve them faster [10, 8]. For example, if there are many calls from one function to another, we do not need to solve the including SAT problem more than once.

3.3 Scaling Variability-Aware Refactoring

Based on variability-aware ASTs (Section 3.1) and configuration knowledge (Section 3.2), we next describe our variability-aware refactoring engine MORPHEUS.

For illustration, we discuss a realistic example of renaming a C-function identifier in the presence of static configurability. We show the standard workflow of the refactoring, present excerpts of internal data structures as a basis for refactoring, and outline differences of variability-aware refactoring to standard refactoring. A formal specification of this and other refactorings is outside the scope of this article and provided elsewhere [8].

Consider the example of a RENAME-IDENTIFIER refactoring taken from BUSYBOX in Figure 3. BUSYBOX is a tool suite of standard UNIX tools (e.g., LS), which is mainly used on embedded systems. Let us assume a developer wants to rename function bbunpack on Line 35 in file bbunzip.c into genunpack.

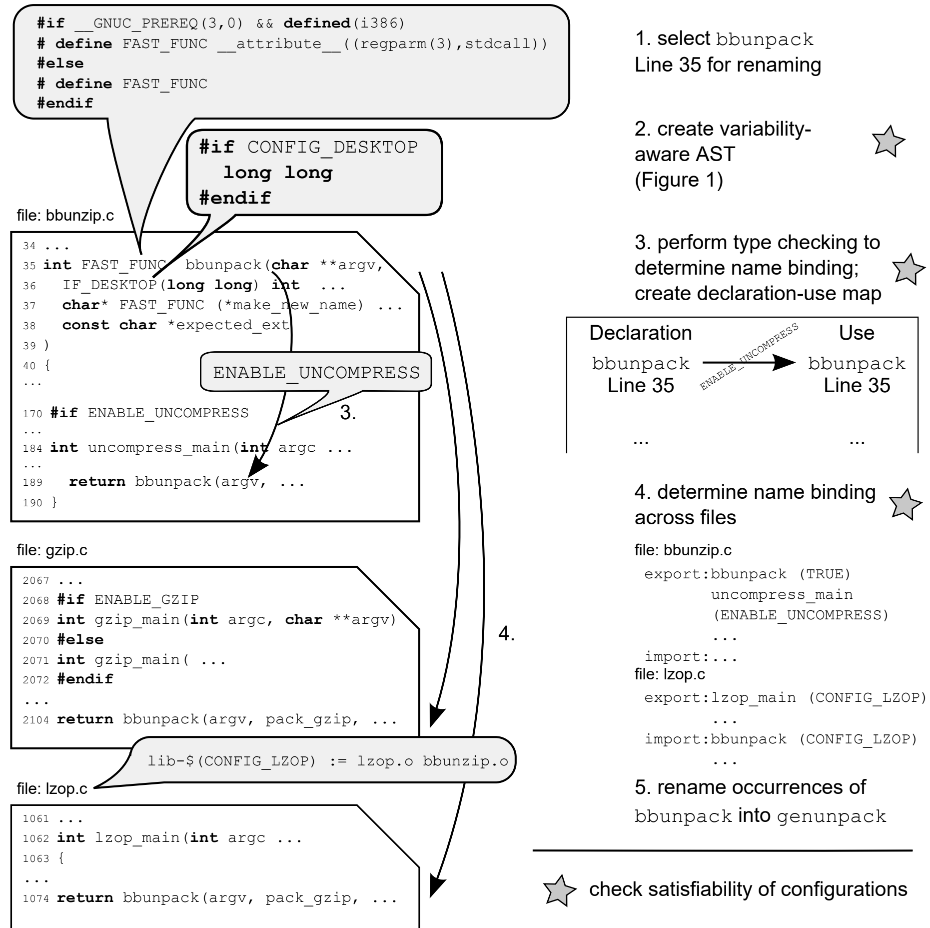


Figure 3: Applying a RENAME-IDENTIFIER refactoring step by step

Step 1. After selecting function bbunpack as the refactoring target and providing the new identifier name genunpack, the engine checks the identifier's conformity with the C standard of identifiers. This is a standard precondition of this refactoring and independent of static configurability. It ensures that the refactored code will compile after the refactoring's application.

Step 2. As the code transformations applied by the refactoring are not performed on the source code directly, MORPHEUS creates a variability-aware AST with TYPECHEF's parser. All configuration-dependent data structures that we use during refactoring are designed to keep the effect of static configurability as local as possible [17]. In particular,

program elements that occur in multiple configurations should occur only once in the data structure. This sharing among configurations leads to a compact representation of the configurable system, avoids the blowup of handling system variants individually (strategy 2), and is one of the main reasons for MORPHEUS' scalability [8]. However, creating data structures with static-configurability information is not for free. It requires solving configuration-related questions using a SAT solver (cf. Section 3.2).

Step 3. On top of the variability-aware AST, the engine creates a map to represent reference information (references between identifier declarations and identifier uses). Since, declarations and usages are configuration-dependent, this map incorporates static-configurability information, too. For our renaming, the use of function bbunpack on Line 189 is configuration-dependent, as the surrounding function definition uncompress_main (Line 184) depends on configuration option ENABLE_UNCOMPRESS. TYPECHEF's internal type checker creates a variability-aware declaration–use map for each C module affected by a particular refactoring. To rule out incompatible renamings (i.e., changing the name of the identifier to an identifier already available, which causes a type error), we check the precondition that genunpack is not in conflict with existing identifiers (variable names, function names, and so forth) in any valid system variant. This precondition check involves SAT solving (cf. Section 3.2).

Step 4. Since bbunpack is used also in other files of BUSYBOX (e.g., gzip.c and lzop.c), we must consider name binding across C modules as well. Neglecting inter-module name binding would lead to linker errors in the final stage of the build process. For this purpose, we extended TYPECHEF with facilities to determine exported symbols (functions available to other modules) and imported symbols (functions required in a module), and we use both pieces of information to compute a configuration-dependent linking interface. Module lzop.c (Line 1074) imports, with respect to configuration option CONFIG_LZOP, function bbunpack, which is exported in module bbunzip.c. Again, the computation of the interface requires a SAT solver. The interface also serves as an input for checking the precondition regarding name binding across files; genunpack should not be in conflict with any existing identifier in other compilation units. This step requires parsing (Step 2) and type checking (Step 3) of other modules, for which name binding was determined using the interface.

Step 5. MORPHEUS renames all identified occurrences of bbunpack in the variability-aware AST consistently for all configurations that can possibly be derived from BUSYBOX, checked again with a SAT solver (cf. Section 3.2). Finally, MORPHEUS applies pretty printing to write the refactored code back to source-code files. The final step omits the recreation of preprocessor directives #define and #include, which were resolved before parsing the code with TypeChef (Section 3.1). The problem of recreating #define and #include preprocessor directives from an AST has been solved before by other refactoring approaches [4, 15], and is applicable also to MORPHEUS (see Section 3.5).

Other refactorings supported by MORPHEUS (EXTRACT FUNCTION and INLINE FUNCTION) follow a similar procedure: variability-aware analysis (including the generation of data structures incooperating variability [17]), precondition checking, and code transformation. Special to EXTRACT FUNCTION and INLINE FUNCTION are variability-aware control-flow graphs (CFGs) [10]. Such CFGs include all possible execution paths of all system variants, incorporating static configurability. They are required for precondition checks, such as, checking the compatibility of the caller and

the callee function for INLINE FUNCTION. Due to space reasons, we do not discuss EXTRACT FUNCTION and INLINE FUNCTION in more detail.

3.4 Large-Scale Case Studies

To demonstrate the feasibility and scalability of variability-aware refactoring in practice, we applied MORPHEUS to three non-trivial subject systems: BUSYBOX (`http://busybox.net`), OPENSSL (`http://openssl.org`), and SQLITE (`http://sqlite.org`).

BUSYBOX is a collection of standard UNIX tools deployed as a single binary. It is highly configurable, with 792 different configuration options and 1.26×10^{159} possible configurations. OPENSSL is a cryptographic library for secure Internet communication. The library provides 589 configuration options, resulting in 6.5×10^{175} different configurations. SQLITE is an embedded relational database-management system, which can be integrated into other software systems, such as Web servers. SQLITE provides 93 configuration options, which give rise to 1.02×10^{39} different configurations.

In a series of experiments, we applied the three different refactorings, RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION, to all three subject systems. We randomly selected program elements as targets for refactoring that are affected by static configurability. Similar to the example of Figure 3, we selected identifiers surrounded by preprocessor directives for the RENAME-IDENTIFIER refactorings applied to the subject systems. For EXTRACT FUNCTION and INLINE FUNCTION, we selected statements and function definitions surrounded by preprocessor directives, respectively. In total, we applied 11 479 refactorings (11 068 RENAME-IDENTIFIER refactorings, 224 EXTRACT-FUNCTION refactorings, and 177 INLINE-FUNCTION refactorings). We ran all experiments on common desktop hardware.

As a key result, we observed that MORPHEUS performs reasonably well. Applying refactorings on real C code with preprocessor directives is in the order of milliseconds, so variability-aware refactoring is feasible in practice and is able provide an instantaneous user experience. Our results suggest that the critical operation (SAT solving) is not a bottleneck, as the configuration issues we have to address can be solved efficiently with existing standard SAT solvers and caching. For a comprehensive discussion of our experiment setup and all measurement results, we refer the interested reader elsewhere [8].

3.5 Limitations and Perspectives

Although we made a major leap toward scalable, sound, and complete refactoring in the presence of static configurability, some limitations remain to be addressed.

First, while MORPHEUS scales to configurable systems with billions of system variants, it does not fully support the entire diversity of the C ecosystem (e.g., it does not support the full set of GNU extensions). Second, capturing a system's configuration knowledge may be a difficult task for complex legacy systems. Often there is an impervious network of mutually dependent build scripts and configuration files that hinders automated analysis [3], so developers may still be required for setting up the project initially. Third, MORPHEUS does not support full round tripping yet (cf. Section 3.1).

The reason is that it uses TYPECHEF's variability-aware parser [6], which applies partial preprocessing (i.e., the resolution of #define and #include) as well as automatically transforms preprocessor directives that do not align with the syntactic structure of the underlying C code. Both transformations need to be tracked and reversed during the pretty-printing process in order to return the refactored, unpreprocessed source code to the developer. We could apply the reverse operation, as implemented in CREFACTORY [4]. The basic idea is to store additional information in AST nodes, such as the original location of tokens that form the AST node as well as information on macro expansion and file inclusion. Such information can be employed by the pretty printer to recreate #define and #include directives. Alternatively, we could use MORPHEUS to determine required changes to the source code in form of patches, as implemented in XREFACTORY [15]. Instead of applying the refactoring's transformation on the AST, the engine would infer textual patches (similar to diffs in version control systems) based on variability-aware analysis. A renaming, as in Figure 3, will consist of a patch, changing the identifier of the function definition and all function calls in affected code files. Applying the patch to the original source code yields the refactored code, thus avoiding a pretty-printing step during the refactoring. Using one of the two approaches, the full round-trip for variability-aware refactoring is possible.

While addressing these limitations requires additional effort, there are no principle obstacles left. For the first time, we are able to handle static configurability in refactoring in a scalable, sound, and complete manner in practical settings. Beyond this, our approach can be easily extended to other artifacts, languages, and tools that support static configurability as well as to more complex refactorings and other kinds of program transformations.

Acknowledgment

This work has been supported by the German Research Foundation (AP 206/4 and AP 206/6).

About the Authors

Jörg Liebig is a consultant for variant management at Method Park, Germany. He received his Ph.D. in Computer Science in 2015 from the University of Passau, Germany. His research interests include code analysis and transformation, code-complexity metrics, implementation techniques for software product lines, and software architecture.

Sven Apel is a full professor and holds the Chair of Software Engineering at the University of Passau, Germany. The chair is funded by the esteemed Emmy-Noether and Heisenberg Programs of the German Research Foundation (DFG). His research interests include software product lines, software analysis, optimization, and evolution, as well as empirical software engineering.

Andreas Janker is a M.Sc. student in Computer Science and research assistant at the Chair of Software Engineering at the University of Passau, Germany. In his Bachelor's thesis, he laid important foundations for implementing variability-aware refactoring. In his current research, he is interested in variability-aware code analysis and transformation on top of the TYPECHEF framework.

Florian Garbe is a M.Sc. student in Computer Science and research assistant at the Chair of Software Product Lines at the University Passau, Germany. He has been working for several years on developing variability-aware code transformations on top of the TYPECHEF framework.

Sebastian Oster leads the variant-management team at Method Park, Germany. He received his Ph.D. in 2011 from the Technische Universität Darmstadt, Germany. His research interests include variant management in software and systems engineering, complexity management, and software testing.

References

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[2] I. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 625–634. IEEE, 2004.

[3] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 21–30. ACM, 2012.

[4] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois, 2005.

[5] P. Gazillo and R. Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 323–334. ACM, 2012.

[6] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[7] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[8] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-Aware Refactoring in the Wild. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 380–391. ACM, 2015.

[9] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011.

[10] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013.

[11] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering (TSE)*, 30(2):126–139, 2004.

[12] Y. Padioleau, J. Lawall, R. Hansen, and G. Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the EuroSys Conference*, pages 247–260. ACM, 2008.

[13] M. Platoff, M. Wagner, and J. Camaratta. An Integrated Program Representation and Toolkit for the Maintenance of C Programs. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 129–137. IEEE, 1991.

[14] D. Spinellis. CScout: A Refactoring Browser for C. *Science of Computer Programming (SCP)*, 75(4):216–231, 2010.

[15] M. Vittek. Refactoring Browser with Preprocessor. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE, 2003.

[16] D. Waddington and B. Yao. High-Fidelity C/C++ Code Transformation. *Science of Computer Programming (SCP)*, 68(2):64–78, 2007.

[17] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226. ACM, 2014.