Morpheus: Variability-Aware Refactoring in the Wild

Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer
University of Passau, Germany

***Abstract*—Today, many software systems are configurable with *conditional compilation*. Just like any software system, configurable systems need to be refactored in their evolution, but their inherent variability induces an additional dimension of complexity that is not addressed well by current academic and industrial refactoring engines. To improve the state of the art, we propose a *variability-aware refactoring* approach that relies on a canonical variability representation and recent work on variability-aware analysis. The goal is to preserve the behavior of *all* variants of a configurable system, without compromising general applicability and scalability. To demonstrate practicality, we developed MORPHEUS, a sound, variability-aware refactoring engine for C code with preprocessor directives. We applied MORPHEUS to three substantial real-world systems (BUSYBOX, OPENSSL, and SQLITE) showing that it scales reasonably well, despite of its heavy reliance on satisfiability solvers. By extending a standard approach of testing refactoring engines with support for variability, we provide evidence for the correctness of the refactorings implemented.**

I. INTRODUCTION

Today, many software systems are configurable to support a variety of requirements and hardware platforms. A common mechanism to implement configurable systems is *conditional compilation*, with the C preprocessor (CPP) as its most prominent and widely used tool. With CPP, programmers annotate source code with *preprocessor directives* (e.g., `#ifdef WIN32`) to include or exclude the annotated code conditionally according to the selection or deselection of a configuration option (e.g., `WIN32`) before compilation.

As most software systems, configurable systems evolve. *Refactoring* is an important technique to governing software evolution [41]. While refactoring has been studied thoroughly in academia and proved successful in practice [41], the variability of configurable systems adds a new dimension of complexity that has not been tamed so far. A key challenge is to ensure *behavior preservation* not only of a single system, but of *all* system variants that can possibly be derived from a configurable system. This turns out to be problematic because of the possibly huge configuration space.

Existing refactoring approaches and tools use heuristics to reason about variability (which does not guarantee behavior preservation) [21], [46], employ a brute-force strategy to process all variants individually (which does not scale to realistic systems) [60], [61], [56], or limit the use of variability (which makes many systems unrefactorable) [39], [7], [25], [47]. To make matters worse, state-of-the-art refactoring engines of widely used IDEs, such as ECLIPSE and XCODE, even produce erroneous code when applying standard refactorings in the presence of preprocessor directives (e.g., RENAME IDENTIFIER or EXTRACT FUNCTION).

We strive for a refactoring solution that is *sound* (i.e., preserves the behavior of all variants), that is *general* (i.e., does not rule out large sets of configurable systems), and that is *scalable* (i.e., scales to systems of substantial size—hundred thousands of lines of code). The key idea is to build the refactoring engine on a canonical representation of the configurable system that includes all variability [30], and to use variability-aware static analysis [13], [10], [31], [37] and transformation to implement refactorings.

To this end, we build on recent developments of variational data structures [18], [62] and variability-aware analysis techniques [37], which make *variability-aware refactoring* possible in the first place. While there have been some proposals for variability-aware refactoring [50], [51], [52]—all based on academic languages and tools—we go beyond that in supporting the full power of C and CPP. We developed the refactoring engine MORPHEUS that, exemplary, implements the three standard refactorings RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION. We applied MORPHEUS to the three substantial, real-world systems BUSYBOX, OPENSSL, and SQLITE to assess its correctness and scalability. Although the engine relies internally on solving many satisfiability problems, it scales far beyond state-of-the-art tools (that guarantee behavior preservation), which was not to be expected: The response time of a standard refactoring is less than a second, on average. For each subject system, we used a substantial test suite to provide evidence for the correctness of our refactoring engine. For this purpose, we extended a standard approach of testing refactoring engines [16], [24], [54] with support for variability.

In summary, we make the following contributions:

- We characterize the problem of refactoring configurable systems and discuss the shortcomings of existing refactoring approaches from academia and practice.
- We provide specifications of three variability-aware refactorings based on the three standard refactorings RENAMING IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION. The specifications rely on variational data structures and variability-aware analysis.
- We offer a tool, called MORPHEUS, for variability-aware refactoring of C code with preprocessor directives, supporting the three refactorings we specified. MORPHEUS has three desirable properties. First, MORPHEUS is *sound*: It preserves the behavior of all variants of a given

configurable system, as it relies not on heuristics but on a precise and efficient variability representation. Second, it is *general* and can principally be applied to any C program with preprocessor directives. Third, it is *scalable*, as it uses variability-aware analysis to take advantage of the similarities of variants.
- We apply MORPHEUS to three substantial, real-world systems: BUSYBOX, OPENSSL, and SQLITE. MORPHEUS performs reasonably well in all three case studies; the response time of a standard refactoring is less than a second, on average.
- We use an extensive test suite to provide evidence that MORPHEUS operates correctly. To this end, we extend a standard approach of testing refactoring engines [16], [24], [54] with support for variability.
- We discuss the merits and perspectives of variability-aware refactoring based on the experience we gained.

MORPHEUS, the subject systems, and all experimental data are available on the project's Web site: http://fosd.net/morpheus/ .

II. STATE OF THE ART

Before we introduce our approach of variability-aware refactoring, we review the refactoring capabilities of state-of-the-art development environments for C. In particular, we outline their operation principles and discuss shortcomings in handling preprocessor directives. However, we start with explaining the terminology that we use throughout the paper. Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [20]. Refactorings employed by developers usually follow a set of *refactoring pattern*s, such as RENAMING IDENTIFIER or EXTRACT FUNCTION [20]. A *refactoring engine* implements these patterns as (semi-)automatic code transformations. We call the application (including preparation and execution) of a particular pattern within a refactoring engine a *refactoring task*. If it is clear from the context, we simply use the term refactoring.

Next, we review a number of publicly available IDEs for C and their refactoring engines, including commercial tools, open-source tools, and research prototypes. Note that most development environments lack a refactoring engine and provide only a simple textual search-and-replace functionality. This kind of functionality is barely a compensation for a missing refactoring engine and only of limited usability, even for simple refactoring patterns such as RENAMING IDENTIFIER. Therefore, we omit them in our discussion. Table I summarizes the findings of our investigation of refactoring engines and their capabilities.

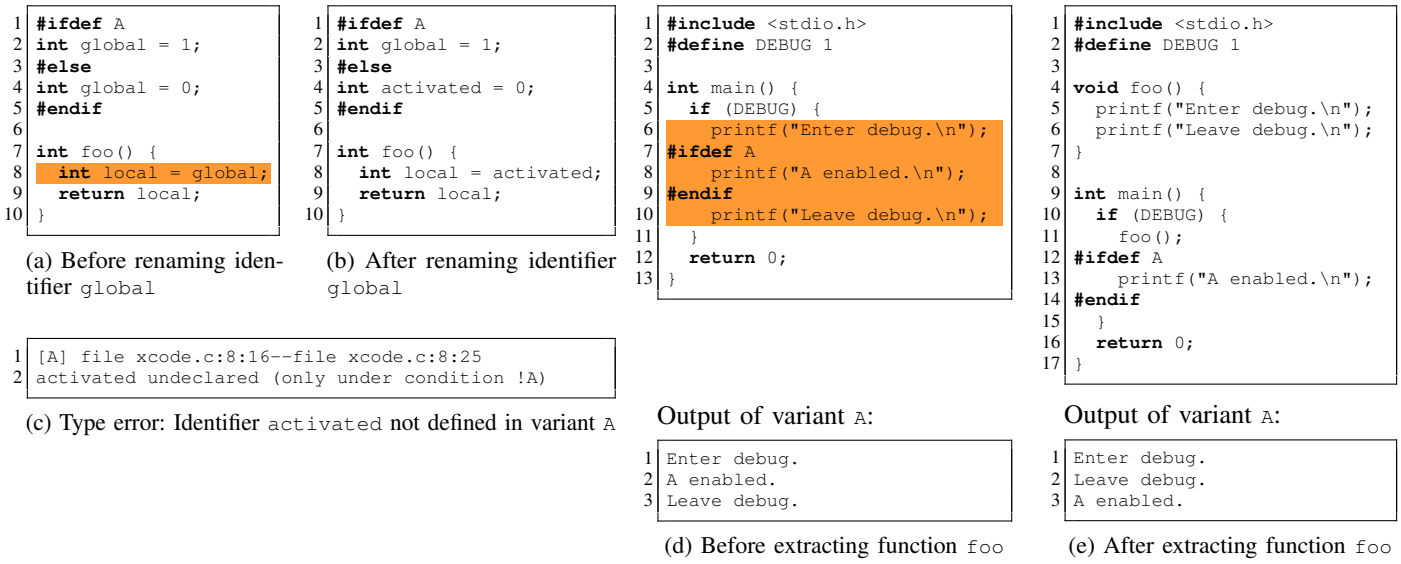
Existing refactoring engines follow one of four operation principles: No variability support, variant-based, disciplined subset, and heuristics.

No Variability Support: Commercial refactoring tools, such as ECLIPSE and XCODE, provide a set of basic refactorings, including RENAMING IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION, which are usually not variability-aware. To handle variability induced by preprocessor directives, these tools evaluate CPP directives implicitly by using default values of the configurable system's project setup. Both ECLIPSE and XCODE basically work on only a single default variant (one of possibly billions). In practice, this can easily lead to errors. For example, in Figure 1, we depict the application of two refactorings: RENAMING IDENTIFIER in XCODE and EXTRACT FUNCTION in ECLIPSE. We were able to apply both refactorings without any warning from the refactoring engines. Unfortunately, after the application, the transformed code contains errors for some system variants. As for RENAMING IDENTIFIER, not all depending identifiers are renamed, causing a compilation error if a particular variant is compiled (variant `A` in Figure 1c). As for EXTRACT FUNCTION, after the selection and extraction of a set of statements, the resulting code compiles but has an altered behavior due to a change in the statement order (Figure 1d vs Figure 1e).

Variant-based: Some refactoring engines do not handle variability directly [60], [55], [56], [61], but employ a variant-based approach. That is, they generate all system variants that are affected by a refactoring, apply a particular refactoring task to each variant independently, and propagate the result back to the variable code. To this end, a developer has to specify one or more configurations, which serve as an input for the generation of system variants. Even though the specification process is sometimes supported by a tool, a major drawback is that specifying system configurations remains a tedious and error prone task. Furthermore, errors in the specification process may easily lead to incorrect code.

The variant-based approach rests on two assumptions. First, a system's number of valid configurations is often low, thus configuration specification can be handled manually. Second, the complexity induced by `#ifdef` directives cannot be handled by refactoring algorithms in practice. In particular, checking the satisfiability (i.e., validity) of configurations, which is a frequent task when refactoring C code, is difficult. Both assumptions do not hold in practice. First, configurable systems usually have a huge number of configuration options giving rise to billions of valid configurations [35], [37]. Second, we and others observed that reasoning about configuration knowledge is tractable even for large software systems using BDDs or SAT solvers [15], [59], [37]. Finally, variant-based approaches face a severe limitation. Since refactoring tasks are applied solely to individual variants of a system, all transformed variants have to be merged again, a problem that is challenging in its own right [40]. Because of this reason, existing engines often support only RENAMING IDENTIFIER, for which merging is easy.

Disciplined Subset: Basically, CPP is a token-based text processor, enabling developers to annotate arbitrary code fragments with `#ifdef` directives. Since refactoring engines require structured input, usually in the form of an abstract syntax tree (AST), a common idea is to disallow arbitrary annotations of code [6], [47], and limit the developer to a subset of *disciplined annotations* [36]: Annotations on entire functions, type definitions, and statements. If developers use solely disciplined annotations, `#ifdef`-annotated source code can be parsed based on preprocessor-enriched grammars [6], [36], and an AST with variability information can be used for further pro-



(a) Before renaming identifier `global`

(b) After renaming identifier `global`

(c) Type error: Identifier `activated` not defined in variant `A`

(d) Before extracting function `foo`

(e) After extracting function `foo`

Fig. 1: Before (Figure 1a) and after (Figure 1b) applying RENAMING IDENTIFIER in XCODE; type error after renaming (Figure 1c); before (Figure 1d) and after (Figure 1e) applying EXTRACT FUNCTION in ECLIPSE, including program outputs

cessing. To apply such a disciplined-subset approach, arbitrary, undisciplined annotations have to be transformed (manually) to disciplined annotations. Although researchers experienced that such manual labor is feasible and scales to medium-sized software systems [6], [47], a recent study showed that undisciplined annotations occur frequently in practice [36]. Consequently, manually disciplining annotations for such systems is a tedious and error-prone task that will hardly be adopted in practice. Similarly, the substitution of CPP by a new language for source-code preprocessing also involves manual code transformation [11], [39]. Even after undisciplined annotations have been disciplined, the generation of a variability-aware AST is particularly challenging, because complex interconnections between **`#ifdef`** directives and **`#define`** macros have to be considered [30]. Existing approaches fail to handle such interconnections properly, as they do not employ a sound and complete parsing approach.

Heuristics: Several approaches use heuristics for automatically transforming undisciplined to disciplined annotations [23], [45]. Then, similar to the disciplined-subset approach, an engine uses an **`#ifdef`**-enriched grammar for the generation of ASTs with variability information. Heuristics either automatically rewrite **`#ifdef`** annotations that do not align with the grammar specification or report problematic code that cannot be parsed to the developer for manual rewriting [23], [45]. The key problem is that an AST generated using unsound heuristics introduces an additional source of error on top of the refactoring challenge. Nevertheless, there are some approaches applying the heuristics-based approach successfully. Padioleau et al. applied a generic patch-generation engine to the LINUX kernel [46]. Garrido and Johnson's refactoring engine CREFACTORY [22], [21] provides a set of simple refactoring patterns (e.g., RENAME VARIABLE, a subclass of RENAMING IDENTIFIER), neglecting complex patterns (e.g., EXTRACT FUNCTION). In addition to the unsound heuristics during parsing, CREFACTORY employs heuristics for reasoning about configuration options: The engine comes without a SAT solver or BDDs for answering configuration-related questions (e.g., whether a code fragment is still selectable after applying FUNCTION INLINE). To the best of our knowledge, CREFACTORY has been applied only in refactoring tasks of small, manageable software systems, so its scalability is an open issue [21].

TABLE I
CLASSIFICATION OF REFACTORING ENGINES

Ref. Engine	Version/Reference	Search & Replace	No Variability Support	Variant-based	Disciplined Subset	Heuristics
CODE::BLOCKS	10.05[1]	✓				
CODELITE	2.8.0[1]	✓				
CREFACTORY	[21]					✓
CSCOUT	[56], [55]			✓		
COCCINELLE	[46]					✓
DMS	[7]				✓	
ECLIPSE CDT	8.2.1[2]		✓			
GEANY	0.21[1]	✓				
GNAT GPS	5.0-6[1]	✓				
KDEVELOP	4.3.1[1]	✓				
MONODEVELOP	2.8.6.3[1]	✓				
NETBEANS IDE	7.4[1]		✓			
PROTEUS	[61]			✓		
PTT	[47]				✓	
XCODE	5[3]		✓			
XREFACTORY	[60]			✓		
VISUAL STUDIO	2013 Prof.[4]	✓[5]				

[1]http://freecode.com/; [2]http://eclipse.org/cdt/;
[3]http://developer.apple.com/xcode/; [4]http://visualstudio.com/;
[5]by default support only via one of several, proprietary extensions

III. Variability-aware Refactorings with Morpheus

The key to variability-aware refactoring is the introduction of variability into data structures and algorithms that are used by the refactoring engine [58], [19], [62]. In this section, we briefly introduce variational data structures (Sections III-A and III-B), and describe the development of a refactoring engine, which is a novel contribution (Section III-C). We denote variability in data structures and algorithms with the type $V[T]$, which represents a variable set of values of data type T; values of T are selected based on a given configuration. The notation and formalization of this representation is based on the *choice calculus* by Erwig et al. [18]. For example, $V[Int]$ can be used to store integer values, and `choice(A,1,2)` is an instance of that type, providing the choice between the values 1 and 2, depending on the selection of `A`. With ϵ we denote the empty selection in choice nodes to express optional elements in the input representation (e.g., `#ifdef A int a = 0; #endif`).

A. Variability-aware AST

To express variability induced by `#ifdef` directives, we extend a standard AST with variability information [30]. To this end, we annotate AST nodes with presence conditions. A *presence condition* is a propositional formula over configuration options that encodes dependencies among options using boolean operators. We extract presence conditions from different sources [17], including variability models [57], [9], build scripts [8], and source code [53]. A variable program element is represented with a `choice` node, which denotes a selection of alternative AST nodes controlled by a presence condition. For the generation of a variability-aware AST, we use the variability-aware parsing and analysis framework TypeChef [30], which can even handle undisciplined annotations: TypeChef has a sound and complete parser that is capable of parsing any C code with `#ifdef` directives and of representing it in the form of a typed AST with presence conditions on AST nodes (`V[AST]`). Figure 2 illustrates the alternative definition of variable `global` (`choice(A,global=0,global=1)`) in our Renaming Identifier example (Figure 1a). Typical AST nodes that we use are `Id` (identifier), `Expr` (expression), `FDef` (function definition), and `Stmt` (statement). For more information on the parsing process and variability-aware ASTs, we refer the interested reader to the literature [30].

B. Variability-aware Analysis

On top of variability-aware ASTs, we created additional data structures, in particular, variability-aware control-flow graphs (CFGs), as well as algorithms to compute static-analysis information that is required during the refactoring process. For example, most refactoring engines exploit type and reference information. Similar to a traditional type checker that maintains a map of identifiers to types (`Map[Id,Type]`) to infer the types of the expressions of an input program (getType : `Map[Id,Type]` $\times$ `Expr` $\rightarrow$ `Type`), a variability-aware type checker traverses a variability-aware input AST, collecting type declarations in a symbol table and checking whether all expressions are well typed [31]. During the type-checking process, the variability-aware type checker incorporates presence conditions in typing information so that type checking can be performed on all variants simultaneously (getType : `Map[Id,V[Type]]` $\times$ `Expr` $\rightarrow$ `V[Type]`) [37]. We use the results of type checking to obtain reference information (`RefInf` $=$ `Map[Id,List[V[Id]]]`). The key (`Id`) represents a variable usage or a variable declaration, while the values (`List[V[Id]]`) represent corresponding variable declarations and variable usages. For example, Figure 2 shows reference information for variable `global` of our Renaming Identifier example (Figure 1a). All references of variable `global` in Line 8 are linked to their original declarations including presence conditions (`List[V[Id]]`). For more details on variability-aware type checking, we refer the reader to the literature [3], [29], [31].

In a similar fashion, we created variability-aware CFGs [37]. In a nutshell, we augmented the CFG's successor relation for the representation of all execution paths in a program with variability (succ : `CFG` $\rightarrow$ `List[V[CFG]]`). That is, depending on variability in the code, the successor of a program element may vary, and again we used presence conditions to encode this variability. For example, the successor of statement `printf("Enter debug.\n")` in Line 6 in our Extract Function example (Figure 1d) is either `printf("A enabled.\n")` (Line 8) or `printf("Leave debug.\n")` (Line 10), depending on the selection of configuration option `A`. This kind of information is crucial for the definition of variability-aware refactoring patterns [49]. For more details on variability-aware control-flow analysis, we refer the reader to the literature [12], [37], [10].

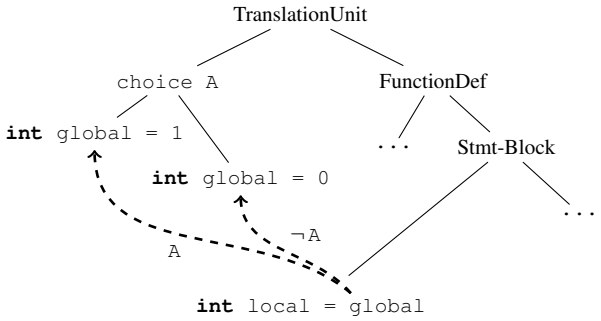


Fig. 2: AST representation enriched with reference information of the Renaming-Identifier example in Figure 1a; node `choice A` represents a variable AST node providing a selection of two different definitions of variable `global`

C. Specification of Variablity-Aware Refactorings

As representative and widely used refactoring patterns [42], we select Renaming Identifier, Extract Function, and Inline Function. For the definition and implementation of the refactorings, we abstract from the underlying variability-aware analysis framework and rely on an interface, as illustrated in Figure 3. Note that the signature of the interface incorporates variability (use of the `V` type constructor).

Renaming Identifier: The challenge of the specification of Renaming Identifier is that all identifiers in a configurable program (e.g., function names, function parameters, local

addFDef : V[AST] × V[FDef] → V[AST]
compatibleCFG : V[AST] × V[FDef] → Boolean
genFCall : V[AST] × List[V[Stmt]] × Id → V[Stmt]
genFDef : V[AST] × List[V[Stmt]] × Id → V[FDef]
genFPro : V[FDef] → V[Stmt]
getDefs : RefInf × List[V[Id]] → List[V[Id]]
getPC : List[V[Stmt]] → PC
getUses : RefInf × List[V[Id]] → List[V[Id]]
getModuleReferences : RefInf × Id → List[V[Id]]
getProgramReferences : CProgram × Id → List[(V[AST],List[V[Id]])]
insertBefore : V[AST] × V[FDef] × V[Stmt] → V[AST]
isFunctionCall : V[AST] × Id → Boolean
isRecursive : V[AST] × V[FDef] → Boolean
isValidId : Id → Boolean
isValidInModule : TypeEnv × List[V[Id] × Id → Boolean
isValidInProgram : CProgram × V[Id] × Id → Boolean
isValidSelection : V[AST] × List[V[Stmt]] → Boolean
isWritable : V[Id] → Boolean
replaceFCalls : V[AST] × List[Id] × List[V[FDef]] → V[AST]
replaceIds : V[AST] × List[V[Id]] × Id → V[AST]
replaceStmts : V[AST] × List[V[Stmt]] × V[Stmt] → V[AST]
typeCheck : V[AST] → (TypeEnv, RefInf)

Fig. 3: Auxiliary functions of MORPHEUS that provide the interface to the underlying variability-aware analysis and transformation engine

or global variables, and user-defined data types) may vary depending on configuration options. In our RENAMING-IDENTIFIER example (Figure 1a), variable `global` is defined twice, for `A` and ¬`A`. For a consistent renaming of such identifiers, which can possibly be scattered across multiple source files, we employ reference information (within a file using `RefInf` and across files using `CProgram`). If we select one identifier for renaming, we rename also all dependent references, even across several files.

The refactoring expects the following input: A selected identifier (oid), a variable AST ($tunit$), a global linking interface (li), and a name for the new identifier (nid); MORPHEUS applies the refactoring as follows: After checking that nid conforms to the C standard of identifiers (isValidId), the engine applies variability-aware type checking on the variable input AST (typeCheck). As a result, we get a type environment (te) including all identifiers and their (possibly variable) types (`Map[Id,V[Type]]`) and reference information (ri) of declarations and usages of variables (`Map[Id,List[V[Id]]]`). The representation of identifiers and their types in the former map ensures detecting conflicting identifiers, i.e, an identifier cannot be renamed to nid, if this identifier is already available with a corresponding type (with respect to the given presence condition). The latter map helps to preserve name binding—the crucial property of this refactoring. For example, to rename variable `global` in Line 8 (Figure 1a), the engine determines the transitive closure of identifier usages and their declarations (getModuleReferences). The resulting list contains all variable identifiers including their presence condition: $rid = [\texttt{choice}(\texttt{A}, global, global)]$ (Figure 2).

MORPHEUS performs three checks: First, MORPHEUS rules out impossible renamings of identifiers in system-header files (e.g., the renaming of function `printf` in file `stdio.h`)—files affected by the refactoring must be writable (isWritable). Although simple, this condition is not checked by ECLIPSE's refactoring engine for C.[1] Second, to preserve binding and the visibility of identifiers, the engine checks possible violations of C's scoping rules for each identifier (isValidInModule) that is going to be replaced with nid. To do so, MORPHEUS accesses the type environment te and determines whether there is a variable definition in any variant that is in conflict with the identifier nid. Third, RENAMING IDENTIFIER does not affect only the source file (module) on which the developer currently operates; renaming a function declaration or function call may require renamings of corresponding calls or declarations in other modules. To support refactorings with a 'global' effect, we rely on a data structure for module interfaces (`CProgram`), i.e., a map of all modules and their imported/exported symbols (function declarations),[2] including presence conditions [31]. Using li (`CProgram`), MORPHEUS determines defined symbols (limited to declarations determined with getDefs) in conflict to nid (isValidInProgram), and terminates if there are any. If all premises (isValidId, isWritable, isValidInModule, and isValidInProgram) apply, we replace all references of oid (using rid) in the variable AST $tunit$ with nid (using replaceIds). The RENAMING-IDENTIFIER specification in Figure 4 does not include renamings of files with linked identifiers. To support their renamings, MORPHEUS uses getProgramReferences to fetch dependent variable ASTs and linked identifiers. The engine uses both to apply renamings in dependent files in the same fashion as rename (Figure 4).

Extract Function: We have already seen in Figure 1a that EXTRACT FUNCTION can be problematic. A program's control flow has to be preserved while different code transformations are performed. To address this problem, we employ variability-aware CFGs and reference information. Given a variability-aware AST ($tunit$), a selection of statements ($lstmt$), a global linking interface (li), and a function name ($fname$), we apply EXTRACT FUNCTION as defined in Figure 4: First, MORPHEUS validates the conformance of the given function name with the C standard of identifiers (isValidId). Second, the engine determines whether $lstmt$ is a valid statement selection for extraction (isValidSelection). In particular, MORPHEUS computes a variability-aware CFG and determines whether the selection contains elements that will disrupt the control flow after extraction. To check this property, we traverse the CFG (using the variability-aware successor relation) and ensure that jump targets of problematic code constructs (e.g., **`break`**, **`continue`**, and **`goto`**) belong to the input selection in any variant. Third, as function identifiers can be variable too, MORPHEUS checks $fname$ for violations of C's typing rules within the same module (isValidInModule) and across modules (isValidInProgram) and for all system variants (using the auxiliary function getPC to obtain the common presence condition of the selected statements). This check enables us to

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=396361

[2] Externally visible global variables are currently unsupported.

turn down the extraction of the selected statements in Figure 1d and put them into a function named `main`, as the same symbol is already applied in Line 4. Both validation functions require SAT checks to determine conflicting declarations by querying the type environment (te) and the global linking interface (li). If both checks are successful, MORPHEUS replaces $lstmt$ with the appropriate function call ($fcall$ generated with genFCall), and introduces the new function declaration ($fdef$ generated with genFDef) and its prototype ($fpro$ generated with genFPro) with the auxiliary functions addFDef and insertBefore, respectively.

Inline Function: Similar to EXTRACT FUNCTION, we need to check control-flow properties and reference information. INLINE FUNCTION requires merging a caller's control flow with the callee's control flow, which may result in a disrupted control flow. The merge operation is particularly challenging, because each function call to be inlined may have a different context (e.g., available identifiers) and have a different presence condition. To apply INLINE FUNCTION properly, MORPHEUS must check a set of different premises, which requires solving SAT problems (Figure 4).

First, the engine uses the auxiliary function isFunctionCall to validate that the selected identifier ($fcall$) is a function call. Subsequently, MORPHEUS type checks the variability-aware input AST to infer the type environment (te) and reference information (ri). The former is necessary to determine conflicting identifiers of variables that need to be renamed before the function can be inlined. The latter is necessary to determine all function declarations available for this refactoring. Again, since the source code is annotated with presence conditions, a single function call may reference different function declarations that need to be inlined (if possible), including their context. Using reference information, MORPHEUS determines all dependent identifiers (getModuleReferences) and filters them regarding function calls (getUses) and function declarations (getDefs).

For each function definition in $fdefs$, the engine validates that the function is not recursive (recursive functions cannot be inlined) and that it has a compatible non-disruptive control flow (compatibleCFG). If both checks pass, MORPHEUS inlines each function call incrementally (replaceFCalls). During each inline operation, the engine repeatedly determines reference information for variables and renames them on demand, if they violate C's scoping rules. This occurs especially when multiple function calls that are located next to each other are inlined or when identifiers of the callee function are in conflict with identifiers of the caller function. To handle such cases, MORPHEUS accesses variational data structures, such as reference information (ri), type environment (te), and variability-aware CFG.

IV. EXPERIMENTS

To show that variability-aware refactoring is feasible in practice, we applied MORPHEUS to the three, real-world subject systems BUSYBOX, OPENSSL, and SQLITE. As MORPHEUS relies heavily on SAT solving, a crucial question is whether applying variability-aware refactoring scales in practice.

Next, we introduce the three subject systems and our experiment setup. Then, we present our measurement results and reflect on our experience with applying variability-aware refactoring in practice.

A. Subject Systems

For our experiments, we selected three software systems of substantial size, of which billions of variants can be generated. Beside variability, an important selection criterion was that each subject system is shipped with a test suite. The selected systems have been used in part in previous case studies [30], [31], [37]. The sole restriction to the three subject systems is accounted by the complex setup of variability-aware analysis, which has to be aligned with a system's setup (in particular, the build system).

- BUSYBOX[3] is a collection of standard UNIX tools (e.g., list files with LS) in a single binary, which is often deployed on embedded systems. It consists of 522 files and 191 615 lines of C code (version 1.18.5). BUSYBOX can be configured using 792 different configuration options, resulting in 1.26×10^{159} variants.
- OPENSSL[4] is an open-source implementation for secure Internet communication protocols. Its implementation serves as a foundation for many different software systems, such as Web servers. We used OPENSSL version 1.0.1c with 733 files and 233 450 lines of source code. OPENSSL has 589 configuration options, with which 6.5×10^{175} variants can be derived.
- SQLITE[5] is a library implementing a relational database-management system. The library gained much attention due to a number of desirable properties (e.g., zero-configuration, cross-platform, transactions, small footprint) and is considered the most widespread database management system worldwide, with installations as part of ANDROID, MOZILLA FIREFOX, and PHP. To ease embedding of SQLITE in other software systems, its code base consists only of two source-code files (amalgamation version). We use a recent version of SQLITE (3.8.1) with 143 614 lines of C code, which can be configured using 93 configuration options (1.02×10^{39} variants).

B. Experiment Setup

To create variability-aware ASTs of C code with `#ifdef` directives, we used TYPECHEF (Section III-A) [30]. Based on TYPECHEF's AST representation, we employed TYPECHEF's analysis facilities for type checking and control-flow analysis (Section III-B) [31], [37]. After parsing and analyzing the C code, we applied the three refactorings (RENAMING IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION) to each input file of all systems. As refactoring tasks, we selected code fragments (an identifier, a statement sequence, and a function call) randomly and applied the appropriate refactoring. Since we wanted to measure the effect of variability, we preferably selected code fragments that contained variability. Nevertheless, our refactorings do still work when the selected code fragments do not contain any

[3] http://busybox.net/
[4] http://openssl.org/
[5] http://sqlite.org/

$$\frac{\mathsf{isValidId}(nid) \quad (te, ri) = \mathsf{typeCheck}(tunit) \quad rid = \mathsf{getModuleReferences}(ri, oid) \quad \forall r : r \in rid : \mathsf{isWritable}(r) \quad \forall r : r \in rid : \mathsf{isValidInModule}(te, r, nid) \quad \forall d : d \in \mathsf{getDefs}(ri, rid) : \mathsf{isValidInProgram}(li, d, nid) \quad tunit' = \mathsf{replaceIds}(tunit, rid, nid)}{\mathsf{rename}(li, tunit, oid, nid) \rightarrow tunit'}$$

rename : CProgram × V[AST] × Id × Id → V[AST]

$$\frac{\mathsf{isValidId}(fname) \quad \mathsf{isValidSelection}(tunit, lstmt) \quad (te, ri) = \mathsf{typeCheck}(tunit) \quad pc = \mathsf{getPC}(lstmt) \quad \mathsf{isValidInModule}(te, tunit, fname) \quad \mathsf{isValidInProgram}(li, \mathtt{choice}(pc, fname, empty), fname) \quad fcall = \mathsf{genFCall}(tunit, lstmt, fname) \quad fdef = \mathsf{genFDef}(tunit, lstmt, fname) \quad tunit' = \mathsf{replaceStmt}(tunit, lstmt, fcall) \quad tunit'' = \mathsf{addFDef}(tunit', fdef) \quad fpro = \mathsf{genFPro}(fdef) \quad tunit''' = \mathsf{insertBefore}(tunit'', fdef, fpro)}{\mathsf{extract}(li, tunit, lstmt, fname) \rightarrow tunit'''}$$

extract : CProgram × V[AST] × List[V[Stmt]] × Id → V[AST]

$$\frac{\mathsf{isFunctionCall}(tunit, fcall) \quad (te, ri) = \mathsf{typeCheck}(tunit) \quad rid = \mathsf{getModuleReferences}(ri, fcall) \quad fcalls = \mathsf{getUses}(ri, rid) \quad fdefs = \mathsf{getDefs}(ri, rid) \quad \nexists fd : fd \in fdefs : \mathsf{isRecursive}(tunit, fd) \quad \forall fd : fd \in fdefs : \mathsf{compatibleCFG}(tunit, fd) \quad tunit' = \mathsf{replaceFCalls}(tunit, fcalls, fdefs)}{\mathsf{inline}(tunit, fcall) \rightarrow tunit'}$$

inline : V[AST] × Id → V[AST]

Fig. 4: Specification of RENAMING IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION

variability. In this case, the variational data structures remain invariable, and accesses to them can be handled without consulting BDDs or a SAT solver. Finally, to obtain a proper test suite, we parametrized each refactoring pattern based on a previous test setup for refactoring engines as follows [24]:

Renaming Identifier: For each file, we randomly selected up to 50 identifiers (e.g., function names, local or global identifiers, user-defined data types), and renamed them using a predefined name. Each selected identifier was annotated with at least 1 configuration option (up to 27 options) and its renaming affected multiple configurations (Table II). If the renaming had a global effect, we employed a global module interface for consistent renamings of dependent identifiers in other files (Section III-C). Overall, we renamed 5832 identifiers of BUSYBOX, 5186 of OPENSSL, and 50 of SQLITE.

Extract Function: For each file, we attempted to extract a statement sequence into one function with a predefined name. To this end, we randomly selected sequences of statements (up to 100 selections, similar to Gligoric et al. [24]) from a function's implementation that contain variability in form of `#ifdef` annotated statements. In our experiments, the selected statement sequences were partially annotated with at least 1 and up to 18 configuration options. Given a valid selection, MORPHEUS determined whether the statement sequence satisfied the refactoring's preconditions, as stated in Section III-C. Many files contained only a small number of functions with a small function body, for which we were not able to apply EXTRACT FUNCTION. Overall, we extracted 61 functions of BUSYBOX, 172 of OPENSSL, and 1 of SQLITE.

Inline Function: Similar to EXTRACT FUNCTION, INLINE FUNCTION is not always applicable. Hence, for each file, MORPHEUS scanned the source code for possible function calls with function definitions that could be inlined. Such definitions satisfied the refactorings preconditions, as described in Section III-C. Either the function call or the corresponding function definition contained variability that our refactoring engine had to take into account during the transformation process. At least 1 and up to 6 configuration options affected the refactorings in our experiments. Overall, we inlined 50 functions of BUSYBOX, 126 of OPENSSL, and 1 of SQLITE.

Applying a refactoring, MORPHEUS solves many satisfiability problems, in particular, when accessing the global module interface, reference information, and the control flow (Figure 4). To avoid expensive satisfiability checks, a common approach is to cache the outcome of satisfiability problems already solved [3], [4]. MORPHEUS and the underlying parsing and analysis infrastructure also make extensive use of caching, and we are interested in whether variability-aware refactorings can benefit from caching, too.

C. Performance Results

We ran the experiments for OPENSSL on a LINUX machine with AMD Opteron 8356 CPUs, 2.3 GHz, and 64 GB RAM. For BUSYBOX and SQLITE, we used a LINUX machine with a Intel Core2 Quad Q6600 CPU, 2.4 GHz, and 8 GB RAM. We configured the Java JVM with 2 GB RAM for memory allocation.

In Table II, we list the measurement results for each refactoring and subject system. We report refactoring times and affected variant configurations in terms of mean ± standard deviation as well as the maximum for a single refactoring task. Additionally, we use box plots to visualize the distribution of time measurements and of the number of affected variants

of the refactoring per subject system. The numbers do not include times for parsing and type checking the input C code.[6]

Overall, the results demonstrate that variability-aware refactoring is feasible in practice. For BUSYBOX and OPENSSL, the refactoring times are less than one second (RENAMING IDENTIFIER and EXTRACT FUNCTION), on average. Applying INLINE FUNCTION is a little more expensive since we need to update reference information each time we inline a function call.

For SQLITE, the results are different. Due to the nature of this system (in particular, the fact that the source code has been merged in a single file with more than 140 000 lines of code), a refactoring task takes a comparably large amount of time. But, this is *not* caused by variability, nor by shortcomings of our approach.

D. Testing the Refactoring Engine

To validate that our refactorings are behavior-preserving, we employed a standard testing approach for refactoring engines [16], [24]. We used two test oracles that checked automatically whether a refactoring's code transformation was correct. Our oracles were: (1) the source code of our subject systems still compiles and (2) the results of the system tests (post-refactoring and pre-refactoring) do not vary. In contrast to existing test suites, which do not incorporate variability, we determined which variant configurations were affected and tested them against both oracles automatically. This way, we ensured that MORPHEUS did not introduce any variability-related errors to a system's code base.

For our three subject systems, we used the following tests:

- BUSYBOX comes with a test suite of 410 single test cases in 74 files, of which 46 tests fail (which we ignored during our evaluation). Each test checks the correctness of a single component of BUSYBOX' tool collection.
- OPENSSL's test suite is delivered with its source and checks individual components of OPENSSL's implementation, including the correct implementation of hashing algorithms (e.g., MD5 and SHA-256), key-generation algorithms (e.g., NIST prime-curve P-192), and encryption/decryption of messages using cryptographic algorithms. The test suite runs as a whole and test success is indicated by the output "`ALL TESTS SUCCESSFUL`".
- For testing SQLITE, we used the proprietary test suite TH3.[7] TH3 is a test program generator and provides a full branch test coverage of the compiled object code.

The test setup was as follows: During the application of a refactoring task, MORPHEUS determined, based on the variability-aware transformations, all affected presence conditions. For example, renaming the variable `global` in Line 8 of Figure 1a, has an impact on the presence conditions `A` and ¬`A`. After collecting the affected presence conditions for all refactoring tasks (e.g., applying up to 50 times RENAMING REFACTORING), MORPHEUS computes valid system configurations for the configurable system. For each configuration, the engine compiles and applies all system tests before and after the refactoring and compares their results.

During our experiments, our selected test oracles did not reveal any alteration of a variant's behavior. So there is evidence that MORPHEUS did not introduce any variability-related bugs during refactorings.

E. Perspectives of Variability-aware Refactoring

Despite the encouraging results, there are two engineering issues that need to be solved before MORPHEUS can be applied in a practical setting. First, the parsing infrastructure TYPECHEF applies partial preprocessing of source code before the parser creates the variability-aware AST [30], [32]. That is, the preprocessor directives **`#define`** and **`#include`** are resolved using automatic macro expansion and file inclusion. Both resolutions are necessary to be able to parse C code at all, as both directives manipulate the token stream—even conditionally if annotated with **`#ifdef`** directives—that serves as the input to the parser. As a result, generating source code from the variable-aware ASTs (i.e., pretty printing) requires the additional effort of reversing macro expansion and file inclusion. Existing refactoring engines, such as CREFACTORY, preserve partial-preprocessing information in AST nodes, so that the pretty printer can use them when recreating source code. This approach can also be used in MORPHEUS to support the full round trip (parsing→refactoring→pretty-printing).

Second, setting up TYPECHEF and MORPHEUS for a new software system is a non-trivial task. The main burden is the creation of a proper setup for the parsing, type-checking, and linking infrastructure. As TYPECHEF solely works on the basis of C files, the possibly complex setup of a software system (e.g., configuration scripts, library dependencies, and the build-system setup) has to be made explicit (by hand). In principle, it is possible to use TYPECHEF as a compiler replacement, with the downside that a user has to specify additional information for variability (e.g., which configuration options should be considered variable and which dependencies exist between configuration options). So far, we have made use of a number of existing tools for the extraction of configuration knowledge from build systems [57], [8] and configuration models [9]. Further approaches, such as the automatic inference of configuration knowledge [43], will simplify the setup process further.

We have not yet tried to do everything possible to speed up the transformation process. There is one optimization possibility that is likely to improve the refactoring times substantially. With the idea of continuous development of a software system in mind, there is some potential for improving transformation times by reusing results of variability-aware analyses in subsequent transformations. We can facilitate reuse by storing analysis results persistently, including type-checking, linking, and control-flow information. Using such a cache, it is sufficient to recompute information that changed between consecutive transformation runs by inferring the delta between the versions in question. Along the same line, we can reuse the

[6]Both tasks are usually done as background tasks in development environments and their results (AST and reference information) are used for other tasks beside refactoring (e.g., syntax highlighting and static analysis for error checking).

[7]http://sqlite.org/th3/

TABLE II

MEASUREMENT RESULTS, INCLUDING MEAN ± STANDARD DEVIATION (SD) AND MAXIMUM (MAX) TIME FOR PERFORMING A REFACTORING TASK IN MILLISECONDS; SAME FOR THE NUMBER OF CONFIGURATIONS A REFACTORING TASK AFFECTS (# CONFIGS); BOX PLOTS SHOW THE CORRESPONDING DISTRIBUTIONS (EXCLUDING OUTLIERS)

System	RENAMING IDENTIFIER				EXTRACT FUNCTION				INLINE FUNCTION			
	time in ms		#configs		time in ms		#configs		time in ms		#configs	
	mean ± sd	max	mean ± sd	max	mean ± sd	max	mean ± sd	max	mean ± sd	max	mean ± sd	max
BUSYBOX	72 ± 77.8	988	15.9 ± 20.5	52	410 ± 99.6	778	10.6 ± 18.4	52	4049 ± 2976	19282	4.1 ± 8.2	52
	40 50 60 70		0 20 40		250 400 550		2.0 3.0 4.0		2000 5000		1.5 2.5	
OPENSSL	114 ± 159	2089	1.73 ± 1.36	11	1065 ± 316	2345	1.5 ± 0.72	7	3552 ± 2040	13140	1.11 ± 0.39	3
	0 50 150		1.0 2.0 3.0		600 1200		1.0 2.0 3.0		1000 4000		0.6 1.0 1.4	
SQLITE	476 ± 98.4	1164	56 ± 0	56	1350 ± 0	1350	4 ± 0	4	85378 ± 0	85378	8 ± 0	8
	460 470		40 60 80		800 1400		2.5 4.0 5.5		50000 90000		5 7 9 11	

results from SAT solving not only within a single refactoring task but across consecutive tasks.

F. Threats to Validity

First of all, our selection of only three subject systems threatens external validity, because refactoring tasks in these systems may be particularly easy, and a significant performance loss may only occur when using MORPHEUS with different software systems. We consider this as a minor threat, since we selected systems with a substantial code base (containing several thousand lines of source code), which have been developed over decades by many different developers, and which have been well-received in practice. Furthermore, in our experience, the crucial performance factor of variability-aware refactoring is the time for SAT solving. In general, the time to resolve a single SAT call depends on the number of configuration options and dependencies between them. While existing systems usually have many configuration options [35], the number of option dependencies is usually small, such that satisfiability problems can be solved efficiently. Additionally, recent advances in SAT-solver technology make it easier to solve large problems with thousands of configuration options quickly [59].

Second, for our experiments, we could not rely on existing refactoring tasks, which one could extract from the system's version history. Hence, our tasks may not be representative of practical refactoring tasks applied by developers in the wild. Still, we believe that our large random selection of code fragments for refactoring tasks largely compensates this threat, and that the results provide a reasonable picture of the refactoring performance in practice.

Third, the application of the selected refactoring patterns may be particularly easy and, as a result, not representative of refactorings in practice. We did not strive for incorporating more refactoring patterns, because existing studies show that RENAMING IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION are among the most important refactorings that developers use in practice [42]. Furthermore, other refactorings, such as DELETE UNREFERENCED VARIABLE or REORDER FUNCTION PARAMETERS, have a similar complexity and make use of preconditions and code transformations already covered by our three refactorings. Therefore, we expect a similar performance. Finally, our three refactorings already represent an improvement over the capabilities of state-of-the-art refactoring engines. Such engines typically lack an implementation for more complex refactorings, such as EXTRACT FUNCTION and INLINE FUNCTION, and mainly focus on simpler refactorings (CREFACTORY [22], [21] supports RENAME VARIABLE, DELETE UNREFERENCED VARIABLE, and MOVE VARIABLE INTO STRUCTURE; CSCOUT [55] supports RENAMING IDENTIFIER and REMOVE PARAMETER; XREFACTORY [60] supports RENAMING IDENTIFIER and EXTRACT FUNCTION).

Fourth, one technical problem is that the variability-aware parsing and analysis infrastructure TYPECHEF does not fully support the ISO/IEC standard for C. In particular, the infrastructure implements only a subset of the C standard and extensions of GNU C that are used in our and a set of other subject systems (including the LINUX kernel). During analysis, TYPECHEF ignores unsupported constructs, so we had to exclude 6 files from BUSYBOX and 11 from OPENSSL. Furthermore, we changed 9 files syntactically (e.g., changing invalidly formatted hexadecimal numbers from `\x8` to `\x08`). For OPENSSL, we had to exclude 127 identifiers from refactoring, as they use function pointers in a way not supported by TYPECHEF. SQLITE's source code remained untouched.[8] All our numbers and plots have been generated after excluding problematic files.

V. RELATED WORK

Beside refactoring, which we already discussed in Section II, there are three areas of related work: Variability-aware code transformations, testing refactoring engines, and testing configurable systems.

Variability-aware Transformations: As researchers often discourage the use of preprocessors for the development

[8] A list of all changed and excluded files is available on the project's Web site.

of configurable systems [27], they have proposed novel implementation mechanisms [2], such as aspects [33] and feature modules [5], for variability implementation. There has been some effort of transforming preprocessor-based implementations into aspects [48], [14], [1] and feature modules [28]. This transformation usually rests on the identification of typical patterns of preprocessor use [1], [35] and the definition of appropriate transformation rules for code extraction [28]. Prior to code extractions, developers often have to prepare the source code manually [48], which, however, can be automated in part using MORPHEUS. While the objectives of variability-aware transformations (e.g., separation of concerns) are different, the variant-preservation challenge is the same. To ensure behavior preservation, developers sometimes use run-time tests [38], [28], but in general they do not employ a systematic testing approach as we do.

Closest to our work is the project OPENREFACTORY/C [25], [26], which—similar to our approach—employs variational data structures and algorithms to cope with multiple system configurations. Up to now, the available prototype has not been ready for production use and suffers from several limitations (e.g., support for only one configuration [26] and missing support for preprocessor directives `#define` and `#include` in the source code).

Testing Refactoring Engines: Testing is a common approach of detecting errors in a refactoring engine's implementation [16], [54], [24]. Existing approaches usually generate input programs [54], [16] or use real software projects as test input [24]. Testing procedures usually involve an automatic check of one or more test oracles. In addition to previous work, our testing approach incorporates variability.

Testing Configurable Systems: Since the number of valid configurations of a system can grow exponentially with the number of configuration options, testing all variants individually is elusive [58]. We and others employed sampling (i.e., reducing the number of configurations to an interesting subset) to take variability into account [44], [37]. Sampling was successfully applied in different contexts, enabling the detection of errors in a reasonable amount of time. Fortunately, variability-aware refactoring has a mostly local effect on source code. That is, in our subject systems, RENAMING IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION usually affected only a few configuration options, which enabled exhaustive testing of all variants. Nevertheless, testing approaches such as SPLAT [34], which employ a dynamic analysis of execution traces to infer affected configurations, could reduce the number of configurations to be tested further.

VI. CONCLUSION

Refactoring C code with preprocessor directives is challenging, because a single refactoring may affect not only a single but a multitude of system variants that can be derived from a configurable system. We propose a sound, general, and scalable refactoring approach for C code with preprocessor directives, accompanied by a refactoring-engine implementation, called MORPHEUS. A comparative analysis of state-of-the-art refactoring engines for C revealed that most refactoring engines suffer from one of several shortcomings: They cannot handle variability at all, provide only limited support for variability-aware refactoring, or make use of unsound heuristics. Based on variational data structures and variability-aware static analysis, we specified and implemented sound variability-aware instances of common refactorings (RENAMING IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION). We demonstrated the feasibility of variability-aware refactoring with our variability-aware refactoring engine MORPHEUS by applying it to the three real-world systems BUSYBOX, OPENSSL, and SQLITE with a total number of 11 479 refactorings. Our experimental results show that MORPHEUS performs well, especially, compared to the state of the art: The average transformation time is in the order of milliseconds, performing sound refactorings on real, variable C code.

To verify that our refactorings are indeed behavior-preserving, we extended a standard testing approach with support for variability. We were able to show that all variants of our subject systems still compiled and conformed to the systems' test suites after applying the refactorings.

On top of our interface specification of variability-aware analysis and transformation, further refactorings, such as DELETE UNREFERENCED VARIABLE or REORDER FUNCTION PARAMETERS, are possible, making our refactoring engine MORPHEUS an ideal testbed for experiments of variability-aware refactoring for other researchers. Overall, we demonstrated that sound refactoring engines for C and its preprocessor are within reach, and that variability-aware refactorings, including analysis and transformation, scale to substantial, real-world software systems. MORPHEUS closes a gap in concrete tool support for the development of software systems written in C with CPP.

ACKNOWLEDGEMENTS

This work has been supported by the DFG grants: AP 206/4 and AP 206/6.

REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. Hassan. Can we Refactor Conditional Compilation into Aspects? In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 243–254. ACM, 2009.

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

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[4] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Language-Independent Reference Checking in Software Product Lines. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 65–71. ACM, 2010.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[6] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 281–290. IEEE, 2001.

[7] 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.

[8] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski. Feature-to-Code Mapping in Two Large Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 498–499. Springer, 2010.

[9] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability Modelling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010.

[10] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPL$^{\text{LIFT}}$ — Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM, 2013.

[11] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 333–336. ACM, 2010.

[12] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 13–24. ACM, 2012.

[13] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. *Transactions on Aspect-Oriented Software Development (TAOSD)*, 10:73–108, 2013.

[14] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple Crosscutting Concerns Are Not So Simple: Analysing Variability in Large-Scale Idioms-Based Implementations. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 199–211. ACM, 2007.

[15] L. Chen, M. Babar, and N. Ali. Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 81–90. SEI, CMU, 2009.

[16] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated Testing of Refactoring Engines. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 185–194. ACM, 2007.

[17] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. Understanding Linux Feature Distribution. In *Proceedings of the Workshop on Modularity in Systems Software (MISS)*, pages 15–20. ACM, 2012.

[18] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[19] M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Enginering (GTTSE)*, pages 55–100. Springer, 2011.

[20] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

[22] A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 323–326. IEEE, 2003.

[23] A. Garrido and R. Johnson. Analyzing Multiple Configurations of a C Program. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 379–388. IEEE, 2005.

[24] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic Testing of Refactoring Engines on Real Software Projects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 629–653. Springer, 2013.

[25] M. Hafiz and J. Overbey. OpenRefactory/C: An Infrastructure for Developing Program Transformations for C Programs. In *Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 27–28. ACM, 2012.

[26] M. Hafiz, J. Overbey, F. Behrang, and J. Hall. OpenRefactory/C: An Infrastructure for Building Correct and Complex C Transformations. In *Proceedings of the Workshop on Refactoring Tools (WRT)*, pages 1–4. ACM, 2013.

[27] C. Kästner. *Virtual Separation of Concerns: Preprocessor 2.0*. PhD thesis, University of Magdeburg, 2010.

[28] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM, 2009.

[29] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):1–39, 2012.

[30] 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.

[31] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 773–792. ACM, 2012.

[32] A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010.

[33] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

[34] C. Kim, D. Marinov, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 257–267. ACM, 2013.

[35] 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.

[36] 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.

[37] 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.

[38] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the EuroSys Conference*, pages 191–204. ACM, 2006.

[39] B. McCloskey and E. Brewer. ASTEC: A New Approach to Refactoring C. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 21–30. ACM, 2005.

[40] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering (TSE)*, 28(5):449–462, 2002.

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

[42] E. Murphy-Hill, C. Parnin, and A. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering (TSE)*, 38(1):5–18, 2012.

[43] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 140–151. ACM, 2014.

[44] C. Nie and H. Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2):1–29, 2011.

[45] Y. Padioleau. Parsing C/C++ Code without Pre-processing. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 109–125. Springer, 2009.

[46] 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.

[47] 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.

[48] A. Reynolds, M. Fiuczynski, and R. Grimm. On the Feasibility of an AOSD Approach to Linux Kernel Extensions. In *Proceedings of*

the Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), pages 1–7. ACM, 2008.

[49] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping Stones over the Refactoring Rubicon. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 369–393. Springer, 2009.

[50] S. Schulze, M. Lochau, and S. Brunswig. Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 33–40. ACM, 2013.

[51] S. Schulze, O. Richers, and I. Schaefer. Refactoring Delta-Oriented Software Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 73–84. ACM, 2013.

[52] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 73–81. ACM, 2013.

[53] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-based Variability. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 33–42. ACM, 2010.

[54] G. Soares, R. Gheyi, and T. Massoni. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering (TSE)*, 39(2):147–162, 2013.

[55] D. Spinellis. Global Analysis and Transformations in Preprocessed Languages. *IEEE Transactions on Software Engineering (TSE)*, 29(11):1019–1030, 2003.

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

[57] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Feature Consistency in Compile-Time Configurable System Software. In *Proceedings of the EuroSys Conference*, pages 47–60. ACM, 2011.

[58] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014.

[59] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE, 2009.

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

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

[62] 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.