
**University of Passau**
*Department of Informatics and Mathematics*
Chair for Programming

# Bachelor Thesis

## Implementing Conditional Compilation Preserving Refactorings on C Code

Andreas Janker

| | |
|---|---|
| Date: | March 21, 2013 |
| Supervisors: | Prof. Christian Lengauer, Ph.D. |
| | Dipl. Ing.-Inf. Jörg Liebig |

**Janker, Andreas**

*Implementing Conditional Compilation Preserving Refactorings on C Code*
Bachelor Thesis,
University of Passau, 2013.

**Supervisor Contacts**

Prof. Christian Lengauer, Ph.D.
Chair for Programming
University of Passau
E-Mail: Christian.Lengauer@uni-passau.de
Web: http://www.infosun.fmi.uni-passau.de/cl/staff/lengauer/

Dipl. Ing.-Inf. Jörg Liebig
Chair for Programming
University of Passau
E-Mail: Joerg.Liebig@uni-passau.de
Web: http://www.infosun.fim.uni-passau.de/cl/staff/liebig/

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Motivation

*Refactoring* plays a fundamental role in today's software development process during the phase of implementation and maintenance of a software system. By the use of refactoring the internal structure and corresponding non-functional features, such as code extensibility, readability, or maintenance are improved while preserving the external observable behavior of the program [2].

Almost every modern *integrated development environment* (IDE) (e.g, ECLIPSE[1] or XCODE[2]) provides several ways of refactoring the source code of a software system and allows the developer to apply them quickly. Refactoring techniques are available for a wide range of programming languages, including C. Although C is a particular old language it is heavily used for legacy and system programming, as well as for developing state-of-the-art operating systems and software, such as the LINUX KERNEL or the VLC MEDIA PLAYER. Unfortunately, modern IDEs only offer rudimentary refactoring support for C code. This is caused by the existence and the use of the *C preprocessor* (often referred as CPP) and its directives as part of an independent extension of the C programming language. These directives are used for the inclusion of header files, macro expansion, conditional compilation, line control, and diagnostics [3]. Header file inclusion and macro expansion help the programmer to keep the code easy to read and to maintain by offering the reuse of the same source code(-fragments) across different files. Conditional compilation directives (commonly referred as `#ifdef` directives) enable to configure the software according to the desired available application behavior. These directives annotate which part of the source code should be included in the compile process under a certain configuration. Especially *software product lines* (SPL) written in C, use conditional compilation to implement variability. A software product line (SPL) is a family of related program variants that share a common code base [4], whereas a *feature* is representing a (single) observable behavior of a software. Program variants are created by choosing a configuration set with certain selected/deselected features. With the use of conditional compilation, features are mapped directly in the implementation unit.

[1] `http://eclipse.org/`
[2] `https://developer.apple.com/xcode/`

The CPP directive syntax differs to the general C syntax and is not part of the C grammar. As a consequence, the first step in the translation process of software written in C, which is called preprocessing, is to evaluate and to remove these directives. Generally this job is done by the CPP. The result is pure C code without any directives.

Refactoring pure C code is offered by all state-of-the-art IDEs. However, refactoring preprocessed code is not useful, only this certain configuration is refactored - the original annotated code stays the same and there is currently no successful way back. Current IDEs ignore these directives or only look at one certain preprocessing configuration. It can be stated that the applied refactoring may produce invalid code for other chosen configurations and destroys as a result the variable product generation of a software product line. In order to keep this variability "alive", refactorings need to be applied either in a brute force manner on all possible configurations or preferably, *variability-aware* directly on the original source code. Fortunately, researchers began to develop *variability-aware* source code parsing and analyzing infrastructures for C code [5].

## 1.2 Objective

Due the lack of a fully functional refactoring engine for C code and the widespread use of the C programming language[3], we explore in this thesis the development of techniques to perform one basic and two complex refactoring operations on C code annotated with `#ifdef` directives while preserving its variability:

- *renaming identifiers* (e.g., function names, structures or variables)
- *extract function*
- *inline function*

We implement these refactorings as part of a simple C source code editor, called Morpheus[4]. This prototype editor is a proof of concept that these variability preserving refactorings actually work and that they are complete and correct.

For evaluation purpose, we apply the implemented refactorings on an open source C project: BusyBox[5]. BusyBox is a toolsuite providing the most common UNIX tools stripped in single, standalone executables. It describes itself as

[3]The TIOBE index ranks the C programming language as one of the leading languages over the last 25 years. Details can be found here: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[4]The name is chosen as a tribute to the ancient greek god *Morpheus*, who has the ability to transform himself into any human form. Here in this thesis, source code is transformed (morphed) into a new structure, while preserving the observable behavior and variability.

[5]`https://www.busybox.net/`

*"the swiss army knife of embedded Linux"* and it is runnable on a widespread range of operating system environments such as Linux or FreeBSD. BusyBox is a medium scale software product line with a total amount of 792 unique features and 206815 lines of code. Its variability, represented by its features, is accomplished by annotating the source code with conditional compilation directives.

## 1.3 Structure of this Thesis

Before presenting our approach of a variability-aware refactoring engine, we first introduce in Section 2 some necessary background knowledge: what are exactly software product lines, what is their connection to conditional compilation directives and how is variable source code represented as data structure to apply analysis and transformations on it. Further, we have a closer look at the C preprocessor and its features, especially conditional compilation, and explain how the CPP works and is used in feature orientated software development.

In Section 3 we describe the motivation for a developer to refactor source code in the first place, present in detail three refactoring operations we aim to implement in this thesis with our proof-of-concept tool Morpheus and show how current state-of-the-art IDEs, such as Eclipse or XCode, apply them in the presence of `#ifdef` directives.

Section 4 presents the used source code analysis and parsing infrastructure, TypeChef, and describes the strategy of our three refactoring mechanism offered by Morpheus.

The presented strategies for refactoring variable C source code get evaluated for correctness, completeness and performance in Section 5.

After evaluating our developed refactoring engine, we give in Section 6 a short overview about current and past academic research work in implementing, specifying and verifying refactoring techniques for C source code in the presence of conditional compilation directives.

Finally, we reflect our approach and its result and give an outlook on possible future work.

# 2 Background

This chapter describes some essential (technical-) background knowledge in order to fully understand our motivation and approach in refactoring C code.

## 2.1 Software Product Lines

Today's software market has changed dramatically over the last few years. A good example of this are operating systems (OS). A few years ago, the application field of an operating system was very limited – it was basically run either on a desktop pc at home or in offices, or on servers. In general, the hardware environment (despite of the performance issue) on all systems was the same, machine input was performed via mouse/keyboard, etc. However the application field evolved: today's operating systems now run on mobile devices, on home entertainment systems, and even on household devices. Each application field has different requirements and offers other preconditions such as different hardware environment or the way it interacts with the user. Developing and maintaining a unique variant for each application field is infeasible. To resolve this challenge an idea from the industrial engineering area has been transferred to the software engineering area – a *product line*. Industrial manufactures recognized early that most of their products share many and only differ in certain aspects. For example, two cars may have different colors but the used engine stays the same. The same idea can also be applied to software engineering. The basic features of an operating system, such as memory management and scheduling, are available in each variant, whereas the support, e.g. for a touch input device is optional.

A *software product line* (SPL) is a set of similar program variants using a common code base [4]. Commonalities and differences between single variants of a product line are described in the term of features. A *feature* is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer a configuration option [6]. To produce a single program variant of a software product line, a set of features is chosen and then a specific product is generated. This is generally realized by mapping the features to its corresponding implementation units. Various approaches exist to project the variability of a software product line and its features to its source base [7]. Possible approaches among those are e.g. domain

specific languages, overloading, inheritance, or *conditional compilation*. In this thesis we focus on C and the projection of variability to the source code by the use of the CPP feature *conditional compilation* (see Section 2.2.3), because of its widespread use in open source C projects such as BusyBox, the Linux kernel, or OpenSSL.

## Feature Modeling

Before concentrating on the way how features are represented in C source code, we first have to introduce a model, which represents the commonalities and variabilities of a set of features belonging to a software product line [8]. This *feature model* is typically represented by a *feature diagram* and visualizes the hierarchically arranged features of the model in a rooted tree (a feature diagram of a car is shown in Figure 2.1) [9]. The *root feature*, car, is included in every variant. Every other feature only can be included if its parent is part of the product. *Mandatory features* are features, that have to be included if its parent is a part of the product. In our example these are the chassis, the engine and the gearbox. On the contrary an *optional feature* is a feature, that may or may not be included into the product. In our example we can produce a car with or without rooftop. The inclusion of a feature may require some more features. Let's have a look at the feature engine: we have to specify exactly which engine we want in the product. We can choose between an gasoline powered engine or an electric engine or choose both. This feature constraint is called an *OR group*. Furthermore there is a *alternative (XOR) group*; in our example the transmission: a car must have exactly one specific sort of transmission - either manual or automatic.

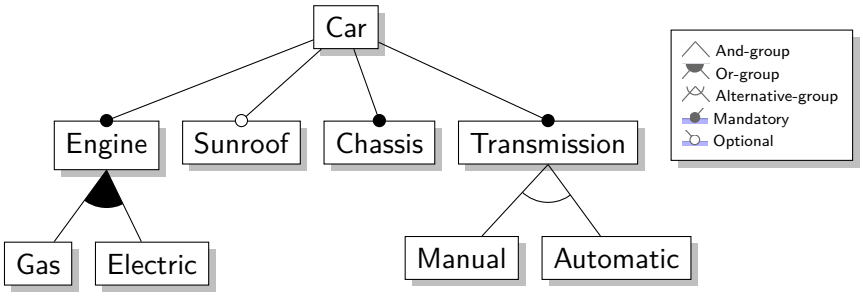



Figure 2.1: A simple feature model of a car.

Instead of representing a feature model by a rooted tree, its semantics can be covered by a *propositional formula* as *presence condition*, where a single feature is represented by a boolean variable [9]. Table 2.1 shows the terms to represent the relationships between features in a feature model.

| Relationship | Boolean Term |
|---|---|
| F (root feature) | F |
| optional child feature C of feature F | $(C \Rightarrow F)$ |
| mandatory child feature C of feature F | $(C \Leftrightarrow F)$ |
| $F_1$, $F_2$, ..., $F_n$ or group of feature F | $((F_1 \vee F_2 \vee ... \vee F_n) \Leftrightarrow F)$ |
| $F_1$, $F_2$, ..., $F_n$ alternative group of feature F | $((F_1 \vee F_2 \vee ... \vee F_n) \Leftrightarrow F) \wedge (\bigwedge_{i<j} (\neg(F_i \wedge F_j)))$ |

Table 2.1: Translation of feature model relationships to boolean terms.

## 2.2 The C Preprocessor

The C Preprocessor (CPP) is a macro processor that is used for source code transformation [10]. By the means of different directives (a.k.a macros) the preprocessor provides functionality for file inclusion, text substitution, conditional compilation, line control, and diagnostics. Its syntax is independent from the underlying programming language. The preprocessor is intended to be used in the source code of the C language family (*C*, *C++*, *C#* and *Objective C*), however it can be abused to process other text files. In this thesis we focus on conditional compilation, because of its use for the implementation of variable software. Nevertheless, for understanding how the preprocessor works, other functionality of CPP relevant to this work are also explained shortly.

### 2.2.1 Macro Definition and Expansion

A *macro* is a named code or text fragment. There are two kinds of macros: *function-like* and *object-like*. Each macro type is created by the `#define` directive, followed by an identifier, in case of a function-like macro directly followed by a parameter list, and the replacement list as shown in Figure 2.2. After a macro has been defined, the preprocessor replaces all subsequent appearances in the source code by the macro replacement list. This process is called *macro expansion*.

```
1 // object-like macro
2 #define <identifier> <replacements>
3
4 // function-like macro
5 #define <identifier>(<parameters>) <replacements>
```

Figure 2.2: Macro definition

***Object-like*** macros are used to create symbolic names for constants. For example

```
1 #define PI 3.14
```

defines $\pi$. Each subsequent occurrence of the macro `PI` in C code will be replaced by 3.14 during preprocessing.

```
1 #define PI 3.14
2 #define PERIMETER(r) (2*r*PI)
3 ...
4 double perimeter = PERIMETER(5);
```

Figure 2.3: Function-like macro before expansion

```
1 double perimeter = 2*5*3.14;
```

Figure 2.4: Function-like macro after expansion

***Function-like*** macros take parameters and act like function calls. For example Figure 2.3 shows a definition of perimeter function macro. After preprocessing it will be expanded to the code listed in Figure 2.4.

The directive `#undef` with the macro identifier as argument allows to undefine a macro. All subsequent appearances of the macro identifier will no longer expanded by the preprocessor. The macro can now be redefined with same name.

Furthermore several macros are predefined by the C preprocessor[1]. These macros can be used without supplying a definition.

### 2.2.2 File Inclusion

The CPP directive `#include` includes an external (header) file in C/C++. A header file is typically used for the inclusion of externally defined declarations and macros to be shared between several source files. By using the `#include` directive, the programmer benefits from the automated inclusion of required declarations and macros, instead of manually copy them into the source file which makes the code harder to maintain and is often a cause for errors. Basically every program written in C uses this directive. Figure 2.5 shows the most basic C program, "Hello World!". In the first line we can see the `#include` directive for the preprocessor.

[1] All predefined macro identifiers are listed at: `http://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html#Predefined-Macros`

During the preprocessing the directive is replaced with the content of the file `stdio.h`, which declares, among other functions and macros, the called function `printf()` for printing out the desired text.

```
1 #include <stdio.h>
2 int main() {
3     printf("Hello, world!\n");
4     return 0;
5 }
```

Figure 2.5: `#include` directive in a "Hello, world!" program written in C

### 2.2.3 Conditional Compilation

*Conditional compilation* directives allows the programmer to advise the preprocessor which part of code whether to include or exclude in the output passed to the compiler. Conditional compilation is mainly used for two reasons:

1. To express differences between variants of a software product line and project features of a SPL into the implementation unit (shown in Figure 2.8).
2. To implement variability in definition of macros.

The syntax of conditional directives is analog to the classic if-elseif-else condition construct offered by almost every programming language. A conditional directive begins with one of the following expressions: `#if`, `#ifdef` or `#ifndef`, followed by an conditional expression and ends with `#endif`. Additionally the directives `#elif` followed by an conditional expression for nested conditions and `#else` are available.

**#ifdef** Figure 2.6 shows the usage of an `#ifdef` directive. The preprocessor will include the controlled text into the output if the MACRO_IDENTIFIER has been defined in the current scope, otherwise no text will be included. Sometimes it is necessary to check if a macro is not yet defined. This can be evaluated by using `#ifndef` directive. Its syntax is the same as of the directive `#ifdef`.

```
1 #ifdef MACRO_IDENTIFIER
2    // text to include if macro is defined
3 #endif
```

Figure 2.6: Usage of `#ifdef` directive

**#if** The `#if` directive evaluates the value of an arithmetic expression following the directive. A valid expression can hereby be an arithmetic operator. During preprocessing the expression will be calculated. If the result is not zero, the controlled text will be included. In Figure 2.7 the usage of the directive is shown.

**#else** With the help of `#if` and `#ifdef` directives it is possible to include code into the preprocessor output. In case the condition fails it is currently not possible to provide an easy alternative. The `#else` directive allows to include an alternative into the output if the previous conditions fail. In Figure 2.7 its usage is shown.

**#elif** By the use of the `#elif` directive nested conditions can be realized. Nested conditions are used to implement more than two alternatives. It can be realized by putting a further `#if` or `#ifdef` directive in a controlled text block by a `#else` directive or by the use of the `#elif`. The syntax of `#elif` is the same as of the directive `#if` as described in Section 2.2.3. Figure 2.7 shows how to use this directive.

```
1 #if ARITHMETIC_EXPRESSION
2     // text to include if expression is not zero
3 #elif Arithmetic_expression
4     // text to include if alternative condition is fulfilled
5 #else
6     // macro not defined? this text will be included
7 #endif
```

Figure 2.7: Sample usage of `#elif` and `#else` directives.

**defined** The directives `#ifdef` and `#ifndef` allow us to determine wether a macro is defined or not. Unfortunately, these directives are unable to test the definition of more than one macro at once. Furthermore the `#elif` for nested conditions only takes an arithmetic expression as argument and cannot test the existence of macros. To solve this issues, the directive `defined macro_identfier` can be included in the arithmetic expression of `#if` and `#elif` directives. It returns `1` if the macro in the argument is defined, otherwise `0` is returned.

Figure 2.8 shows the use of conditional compilation and its directives in our car product line (feature model in Figure 2.1). The example program prints out the configured car by the stackholder. The used `#ifdefs` test the existence of macros that represents different features (engine, body, etc.). Based on defined macros as feature representation, the right alternative is included into the program and used for compilation.

Real life software product lines contain far more features then our presented software product line of a car. For example, the Linux kernel has about 8000 separate features. As consequence, the used conditions become far more complex, but the idea of the approach is still the same and is commonly used for the implementation of variable software [11].

```
1  #include <stdio.h>
2  #include <string.h>
3  int main() {
4          char car[] = "";
5          char body[] = "- a chassis\n";
6          strcat(car, body);
7  #ifdef ENGINE
8    #ifdef GAS
9          char gas[] = "- a gas engine\n";
10         strcat(car, body);
11   #endif
12   #ifdef ELECTRIC
13         char electric[] = "- an electric engine\n";
14         strcat(car, electric);
15   #endif
16 #else
17   #error
18 #endif
..         // a few more lines of code with the other
..         // features would go here
87         printf("The car has this features: %d\n", car);
88         return 0;
89 }
```

Figure 2.8: C code snippet of a car SPL.

## 2.3 Variability - Aware Abstract Syntax Tree

In order to perform analysis or transformations on source code, a data structure representing the syntactic structure of the source code is required. For this purpose generally a tree based data structure is chosen, called *abstract syntax tree* (AST). An *abstract syntax tree* represents the syntactic structure of the source code of a certain programming language in tree form, while excluding unnecessary syntactic details [12, 13]. Each source code construct is denoted as a tree node.

As described in Section 2.2.3, C source code, especially in software product lines, is annotated by `#ifdef` directives. It indicates when specific features relate to the inclusion or exclusion of the annotated code fragment in the compilation unit. Due to the fact that CPP directives are not a part of the C programming language, their projected variability into the source code base is not included in the corresponding classic AST representation. In order to analyze and to transform C source code variability-aware, it must be possible to identify the presence conditon of each tree node. Therefore, the described AST is enriched with information on variability by adding the following extensions to the AST [14]:

- Wrapping each tree node with information about its presence or absence in the compilation unit according its surrounding `#ifdef` directives as *presence condition* (see Section 2.1).
- Introducing two new node types to the AST, representing conditional variability:
  - a node `choice`.
  - a node `one`.

```
1  double pi =
2  #ifdef E
3              3.1415926;
4  #else
5              3.14;
6  #endif
7  double foo(double r) {
8      double conv = 100.00;
9  #ifdef C
10     r = r * conv;
11 #endif
12     double res = 2 * r * pi;
13     return res;
14 }
```

Figure 2.9: Example source code with CPP directives for AST in Figure 2.10.

In Figure 2.9 we see an example code fragment and its corresponding AST representation in Figure 2.10. In Line 1 the value of the variable `pi` gets declared, under the configuration `E` in Line 3 the value `3.1415926` gets assigned, otherwise in Line 5 `3.14`. This conditional variability is represented by the newly introduced node `choice` (yellow) in the AST. Its child nodes, wrapped under which *presence*

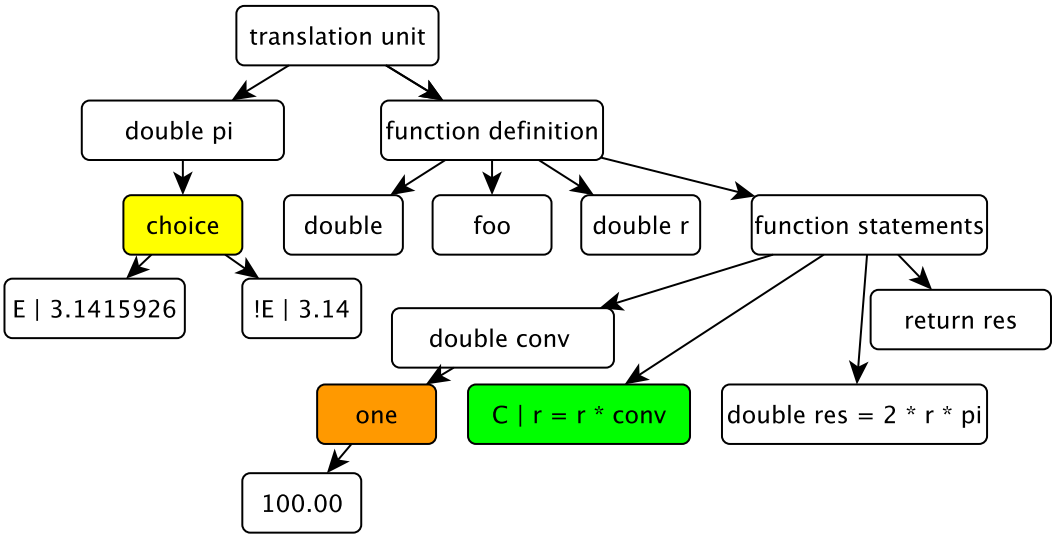



Figure 2.10: (Simplified) AST representation for the source code in Figure 2.9.

*condition* they are chosen, contain the corresponding assignment value. For assign values without any choices, the node `one` (orange) is introduced. In our example it represents the assignment of the variable `conv` to the value `100.00` in Line 8. In Line 10 of our example code listing we see an optional statement, only included to the compilation unit in case Feature `C` is selected. In the AST representation this optional statement is represented by the green node. The surrounding conditional compilation condition of this statement is wrapped around this node as *presence condition*. For simplifying the graphical tree representation we reduced the AST in Figure 2.10 by merging some nodes into a single node and by removing the presence conditions of all nodes which are included in every configuration.

# 3 Refactoring - Why and What?

So why does a developer want to refactor source code in the first place? The motivation and definition for refactoring is rather obvious as defined by Fowler [2]:

> *"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."*

By applying this definition on the goal of our thesis, we aim to implement techniques to *restructure* C source code, while *preserving* its *variability* and *observable behaviour*, in the presence of conditional compilation directives [15].

## 3.1 Why to Refactor?

In his book [2], often seen as the canonical reference for refactoring, Fowler describes several reasons why refactoring of source code is important:

**Design Improvement:** Software evolves in its development process – source code is added or removed, a design pattern is adopted, requirements change, etc, basically the common occurring problems of the software development lifecyle [16]. These changes are mostly made without a full understanding of the whole design and therefore the coding quality gets worse. Code smells such as duplicate, or unused code begin to grow. Refactoring helps to eliminate these code smells and to improve overall design of the developed code.

**Increasing Readability:** Generally source code for a software product is developed as proof of concept. This causes some negative aspects such as unstructured, or unreadable code. Refactoring techniques support the developer to structure its developed code afterwards and make it easier to read.

Still, refactoring is not the *"Silver Bullet"*, as Fowler states it, for curing all software illnesses, or brighten up unstructured and unreadable source code, but yet an *indispensable* technique for improving the quality of code.

## 3.2 What to Refactor and the Resulting Challenges?

We have clarified so far, why refactoring code is a good thing. Next we have to determine what to refactor and what challenges lay before us in implementing and applying refactorings on C code with `#ifdef` directives. In this subsection we present each one of the three refactorings we implement in our editor MORPHEUS, describe the current state of these refactorings in production IDEs, such as ECLIPSE and Apple's XCODE, and give a short overview of the challenges in implementing them variability-aware.

### 3.2.1 Rename Identifier

Renaming an identifer is the most common and the most used refactoring mechanism [17]. In the C programming language we can rename all languages elements identified by a name: variables, functions, structures, unions, enums, and user-defined type-definitions [10]. The idea behind this refactoring is simple: for instance we take a function declaration, rename it with a new name and all associated calls of this function get the same new name.

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

a) Code before refactoring.

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

b) Code after refactoring.

Figure 3.1: Renaming an identifier in XCODE 4.5.1.

As stated before, current IDEs already offer refactor mechanism for C code. But the use of `#ifdef` directives to implement variability (e.g., of a software product line) causes current IDEs to produce an invalid refactoring result. To illustrate the challenges in renaming an identifier, we try to apply this refactoring with Apple's XCODE 4.5.2. Figure 3.1a shows the used example source code. It contains a globally defined variable `global`; its value is variable by the use of `#ifdef` directives. Under the condition `A` its value is `1` otherwise it is `0`. To improve the readability of the source code readability, we now to try to rename the variable from `global` to `activated`. We can see in Figure 3.1b the result using the

renaming mechanism of XCode, only the variable declaration under the default condition has been renamed. As consequence under the condition `A` the refactored code is no longer valid, because the variable `activated` in Line 7 has not been initialized and will lead to an error during compilation. In order to perform a correct refactoring, which would also rename the missed declaration by XCode, the refactoring engine has to be aware of variable declaration and definition of an identifier combined with its appearing usages, as seen in our example, in which the variable `global` in Line 7 refers two different declarations.

The next difficulty in performing this refactoring is *shadowing*. Shadowing occurs when a variable is declared in its local scope as well as in an outer scope (e.g., as a global variable). In the presence of `#ifdef` directives, shadowing may occur only under a certain condition, as consequence without checking for shadowing under each condition, applying a renaming (and also almost every other refactoring) can lead to false results with changing the observable behavior of the refactored code fragment. Figure 3.2 illustrates this problem: the globally declared variable `a` in Line 1 gets renamed to `b`. As result the locally declared variable `b` in Line 4 will shadow the renamed associated variable in Line 6 under the feature `A`. The challenges in detecting shadowing is also present in extract function as well as in inline function.

```
1 int a = 5;
2 void foo() {
3 #ifdef A
4     int b = 3;
5 #endif
6     a++;
7 }
```

a) Code before renaming.

```
1 int b = 5;
2 void foo() {
3 #ifdef A
4     int b = 3;
5 #endif
6     b++;
7 }
```

b) Code after renaming.

Figure 3.2: Variable shadowing.

### 3.2.2 Extract Function

*"Three strikes and you refactor"* [2] is a rule of thumb to determine when it is time to apply refactoring on your developed code. This so called *rule of three* describes best the motivation for the refactoring method *extract function*: the first time you code, one just does it. The second time one does it again, aware of the duplication. But by the third time one should consider to extract the duplicate code into a new function. In order to perform this task efficiently, we want to avoid rewriting the code. As result, the idea of an automated extract function mechanism is born: by

selecting the considered statements to extract, a new function with this statements and necessary function parameters is introduced and the selection is replaced by a call to the introduced function. In Figure 3.3 a simple example is shown: Figure 3.3a is the origin source code, where the statements from Line 3 till 7 of the `if` statement should be extracted into a new separate function `debug()`. The correct result, with the introduced function, is shown in Figure 3.3b.

```
1  void foo(int debugging) {
2     if (debugging) {
3       printf("Debug on");
4  #ifdef F
5       printf("Enabled");
6  #endif
7       printf("Debug off");
8     }
9  }
```

a) Code before extraction

```
1  void debug() {
2      printf("Debug on");
3  #ifdef F
4      printf("Enabled");
5  #endif
6      printf("Debug off");
7  }
8  void foo(int debugging) {
9      if (debugging) {
10       debug();
11     }
12 }
```

b) Code after extraction

Figure 3.3: Extract function example.

```
1  void debug() {
2      printf("Debug on");
3      printf("Debug off");
4      }
5  void foo(int debugging) {
6      if (debugging) {
7        debug();
8  #ifdef F
9        printf("Enabled");
10 #endif
11     }
12 }
```

Figure 3.4: Extract function with Eclipse CDT.

Current IDEs offer as well this refactoring technique. Despite the fact, that it is the third most used refactoring operation by programmers developing with the Eclipse IDE [17], current IDEs also lack of the awareness of variability and as consequence their offered mechanism is considered as broken: Figure 3.4 shows the result of performing this refactoring with CDT 8.1.2 for Eclipse Juno 4.2. Compared to the expected, correct result in Figure 3.3b, we see on first sight, the implementation of this refactoring in Eclipse simply ignores the presence of `#ifdef` directives in Line 4 till 6 of the original code fragment and extracts only the statements, which are not surrounded by `#ifdef` directives. As consequence, the statement order gets changed, which alters the external observable behavior of the refactored C source code.

The challenge now is, to detected occurring variability and extract this variability as well. Especially, certain identifiers and even hole statements may be only visible at a single configuration, and therefore this variability must propagated further, for example in a variable number of function parameters or in statements surrounded with `#ifdef` directives according their variability condition.

### 3.2.3 Inline Function

```
1  #ifdef ADD
2  int bar(int i) {
3     return i + i;
4  }
5  #else
6  int bar(int i) {
7     return i * i;
8  }
9  #endif
10 int foo() {
11    int j = 5;
12    int i = bar(j);
13    return i;
14 }
```

a) Code before inlining

```
1  int foo() {
2     int j = 5;
3  #ifdef ADD
4     int i_2 = j;
5     int i = i_2 + i_2;
6  #else
7     int i_2 = j;
8     int i = i_2 * i_2;
9  #endif
10    return i;
11 }
```

b) Code after inlining

Figure 3.5: Inline function example.

Inlining a function is the opposite of *function extraction*. We hereby take a function and its statements and replace associated function-calls with the statements. On first sight, this refactoring technique looks rather counter-productive, because it makes the code less readable and therefore harder to maintain. But the motivation to perform this kind of refactoring is more a technical issue: the C programming language is considered as one of the leading programming languages for legacy and system programming, an application field which generally offers limited system resources. In order to receive an acceptable performance, developers are often forced to optimize their developed software. Thus, one possible approach is to minimize the overhead caused by the used programming language. By eliminating a function with its occurring calls and inline its statements, the costs of calling the function and the return statement are removed. This refactoring is mainly used for inlining small, often called functions. However, the C language offers the keyword `inline` which instructs the compiler to inline a function at compile-time. Unfortunately, this instruction is rather an hint than an instruction, the decision wether a instructed function gets inlined or not is made by the

compiler at compile-time, according its internal rulings and its configuration for code optimization[1]. To give the programmer full control over its implementation unit, manual inlining has to be performed. This refactoring technique has been specified in Garrido's refactoring catalog for C [18] and Fowler [2], but it is currently not offered by any state-of-the-art IDE for C code. For Java source code it is the fifth most used refactoring technique in Eclipse [17].

Conditional compilation directives allow programers to implement a function variable: different return types can be chosen, the amount of parameters may differ, etc. Therefore we need to inline these functions according the variability where the call occurs, as well as the overall variability of the function itself.

### 3.2.4 Further Refactoring Techniques

The presented refactoring techniques above are only a selection of a wide range of available operations. Garrido et al. developed a hole catalogue of possible refactorings on C source code, such as moving variables into a structure, converting variables into pointers, etc. [19]. We have chosen these three refactorings, because they are, as a recent study has shown [17], the most frequently used refactorings in day to day software developing work. Furthermore, the challenges in implementing these refactorings cover most of the problems of further refactoring mechanism, such as shadowing, variability-awareness, liveness analysis, etc [2, 18].

[1] A short overview how GNU C compiler deals with the keyword `inline` is given at: `http://gcc.gnu.org/onlinedocs/gcc/Inline.html`

# 4 Variability - Aware Refactoring Engine: Morpheus

In this section we present our approach and solutions to the challenges in variability preserving refactorings on C source code in the presence of preprocessor directives.

## 4.1 General Overview of Morpheus

In this thesis we implement three refactoring techniques on C source code in a small refactoring editor, MORPHEUS. Due to the fact, we are unable to prove the correctness of our tool in respect to all possible C situations, we present this tool as proof of concept of our approach. The architecture of MORPHEUS consists of two separate components (see Figure 4.1):

1. a graphical frontend for displaying the input and refactored source code, and for triggering the desired refactoring mechanism.
2. a backend model for parsing, analyzing and transforming C source code in the presence of CPP directives.

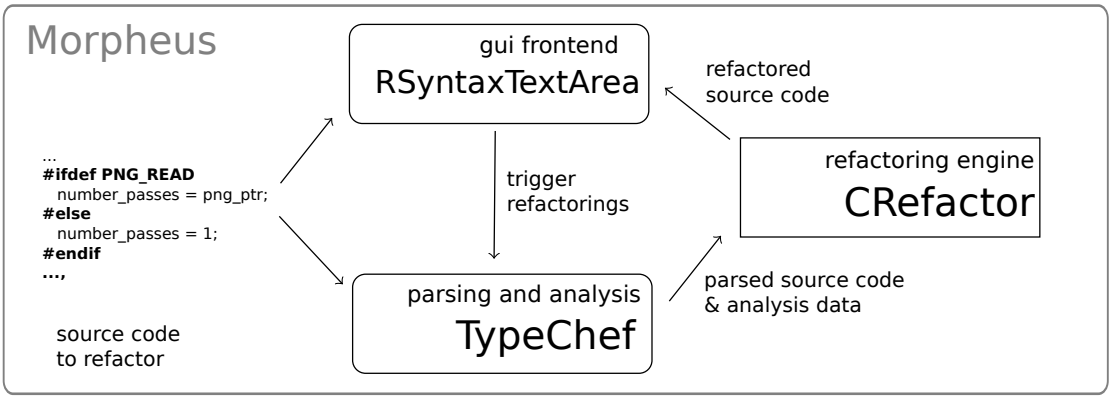



Figure 4.1: An overview of the architecture of MORPHEUS.

Refactorings can only be applied reasonably by providing a *graphical user interface* (GUI), in which the user can select relevant code statements to refactor.

Therefore, we extend RSyntaxTextArea[1], an open source syntax highlighting text component for Java Swing by adding the functionality to trigger the requested refactorings by the user.

In order to apply refactorings, input source code has to be parsed and analyzed first. For this purpose, we use TypeChef[2], a variability-aware parsing and analyzing infrastructure for C code [1]. We use the provided infrastructure to convert the input source code file into a *variability-aware abstract syntax tree* (AST) as described in Section 2.3, on which we can perform our refactorings.

Our main contribution for implementing variability-preserving refactorings on C source code, is our refactoring engine itself. We extend the modular architecture of the TypeChef project with the subproject CRefactor. CRefactor implements all refactoring techniques as explained in Section 3.2.

## 4.2 TypeChef

As illustrated in Section 4.1, before applying refactorings we need to parse the input C source code and represent it as an *abstract syntax tree*. Parsing C code, especially in the presence of preprocessor directives, is a highly difficult task. Different approaches to parse unpreprocessed C code have been proposed and developed over the last decades (see Section 6 for related academic research work). These methods vary from a brute-force approach, meaning parsing and analyzing all possible variants, to heuristics and partial parsing and analysis, meaning identifying common and repeatedly occurring preprocessor directive patterns such as a common file include. They all have been proven to be false, incomplete or simply impractical because of an exponential runtime even for small projects [20]. Recent advances in parsing unpreprocessed C source code led to a variability-aware parsing and analysis infrastructure, called TypeChef. This infrastructure has been developed as research project by Kästner et al. [1, 20, 21, 22]. TypeChef provides a sound and complete representation of C source code annotated with `#ifdef` directives in an abstract syntax tree. The variability of the `#ifdef` directives is represented as choice nodes. TypeChef has been proven sound, complete reasonably performant to fulfill this task. As result, we are able to use it as black-box component for our refactoring editor Morpheus, because it successfully fulfills our requirement of generating a variability-aware input source code representation. In the scope of this thesis, we do not argue about the approach Kästner et al. took with TypeChef [1].

[1] RSyntaxTextArea is available for free at the developer's website:
`http://www.fifesoft.com/rsyntaxtextarea/`

[2] The TypeChef project homepage with source code and further informations:
`http://ckaestne.github.com/TypeChef/`

### 4.2.1 TypeChef Architecture and Function Principle

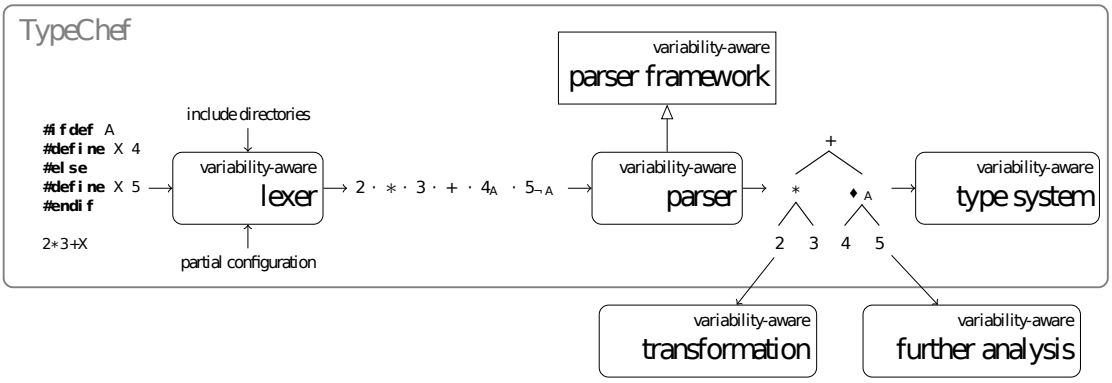



Figure 4.2: Architecture of the TypeChef infrastructure [1].

Figure 4.2 shows the architecture of TypeChef. Our refactoring approch will be implemented as subproject CRefactor of the TypeChef project in the *variablity-aware transformation* node. To understand our refactoring approach, we shortly explain the basic functionality of TypeChef. For deeper technical background knowledge and verification of the used infrastructure we refer the interested reader to [1].

The first component of TypeChef is a *variability-aware lexer*. This lexer reads in the C source code file with some configuration parameters and performs *partial preprocessing* of the input. During the partial preprocessing process, the lexer splits the input code fragment into token streams and annotates these tokens with a *presence condition*, this is a projection of the variability, represented by the conditional compilation directives of CPP. It indicates under which variant the token stream is included in the compilation unit. Furthermore, during this lexing phase all necessary include files are included and macros are expanded. File inclusion is performed analog to the file read-in process, by splitting the code fragment into tokens with a presence condition. Macro expansion is simply performed by replacing each occurrence of a macro with its variability-aware expansion token.

The next step is parsing of the token stream, which is generated by the lexer, into a single, variability-aware *abstract syntax tree*. Variability-aware parsing is a complex issue and to describe it in detail is not part of the scope of this thesis, so we only give a very limited overview. In short, the parsing strategy of TypeChef is to process the input token stream in a single pass; in case a token with variability is parsed, the token is split into different *parsing contexts*. The different parsing contexts are joined together as soon as their presence condition is equal into a variability-aware *choice* or *opt* node of the resulting AST.

After parsing, the resulting variability-aware AST is type-checked by the variability-aware type-system TypeChef to detect syntax and type errors in all

possible configuration combinations.

### 4.2.2 Variabilty-Aware Declaration/Usage Map

In order to retrieve a correct refactoring result, we require a variability-aware mapping between definitions and declarations of all identifiers and its corresponding usages. Fortunately, TypeChef already offers a variability-aware type-system. During the type-checking process of the AST, the required information of the (possible variable) declaration of an used identifier is temporarily present. To preserve this knowledge, we implemented a *variabilty-aware declaration/usage map* as a hook into the type-system of TypeChef called CDeclUse. This map contains all occurring usages of a declaration according to the declaration's presence condition as well as all occurring, variable declarations of a single identifier's usage.

Based on the example code listing in Figure 4.3, we describe the idea behind this mapping and the resulting map. In our example, the variable `i` is declared and used in several seperate statements with different presence conditions. Under condition `F` it is declared with the primitive data type `float` in Line 3, otherwise it is declared as an `integer` in Line 6. After the declaration, in both cases a value according to the chosen data type is assigned (Line 4 and 7). In Line 9 the variable is used in a non-variable statement. Finally, if condition `F` is selected, the variable is used once more in Line 11.

```
1  void foo() {     declaration = {}; use = {};
2  #ifdef F
3     float i;      declaration = {}; use = {(i,3),(i,9),(i,11)};
4     i = 0.1;      declaration = {(i,3)}; use = {};
5  #else
6     int i;        declaration = {}; use = {(i,7),(i,9)};
7     i = 7;        declaration = {(i,6)}; use = {};
8  #endif
9     i += 2;       declaration = {(i,3),(i,6)}; use = {};
10 #ifdef F
11    i += 0.8;     declaration = {(i,3)}; use = {};
12 #endif
13 }
```

Figure 4.3: Code example with an alternative variable declaration and its associated entries in the declaration/usage mapping with the referenced name and line of the identifier.

The creation of our variability-aware declaration/usage map relies on the

type-system of TypeChef and the data it stores in its variable scope environments. During traversing the AST in the type-checking process, each occurring identifier is checked for its correct declaration and all subsequent appearances are connected to each other in a consistent way according the declaration. This fills the environment instances of the type-system with variability-aware information about identifiers, ast entries, and their corresponding type as well as corresponding declarations. To extract and preserve the required data, namely the declaration of an occurring identifier, we hook our map creation process into the existing type-system. By traversing the AST, each identifier gets visited for type-checking. As final step in the type-checking process, each visited AST entry will be passed to our variability-aware declaration/usage map filling pattern. In this step, we analyze the incoming entry according its AST representation type and discuss the three possible AST representation types below:

```
1  int foo(int i);            // declaration of foo()
2
3  int foo(int i) {           // definition of foo()
4      if (i < 1) {
5        return 1;
6      }
7      int j = foo(i - 1); // usage of foo()
8      int fac = i * j;
9      return fac;
10 }
```

Figure 4.4: Example of a forward declaration of `foo()` and illustration of *declaration*, *definition* and *usage* of an identifier.

**Declaration:** In case the type of the input is a declaration, we add a new declaration which contains the identifier of the entry to our map.

**Definition:** If the type of the input is a definition, we first look up in the type-system scope environment instances whether the definition of the entry has been declared before. This is the case with forward declarations. A forward declaration in C is a declaration of an identifier without providing a full definition. In the C programming language, forward declaration commonly occurs for functions; the function gets declared in a header file, while the including source code file defines the previously declared function. In figure 4.4 we show a basic example of this technique. If no forward declaration is detected, the identifier of the definition will be added to our map as declaration. In case of a forward declaration, the type-system returns the previously occurred declaration. For consistency reasons, we now remove the

retrieved declaration from our map and add the definition as declaration with the removed declaration as usage. This operation is performed, because the type-system will return the currently added definition for subsequent usages of this identifier.

**Usages:** All other identifiers are treated as usages: the type-system of TypeChef provides us a variability-aware result, containing all possible and variable declarations are consistent with the presence condition of the usage's identifier. In our example in figure 4.3 the type-system would return for the variable `i` in line 9, both declarations of `i`. Whereas for the identifier `i` in line 11 only the correct declaration in line 3 would be returned. We add the identifier as usage to all of its declarations to our map as well as the usage itself with its declarations.

The variability-aware declaration/usage map for our example in Figure 4.3 can be seen on the left-hand side of the example source code. It shows for each identifier and each source code line, the associated variability-aware declaration/usage map entry.

## 4.3 General Approach

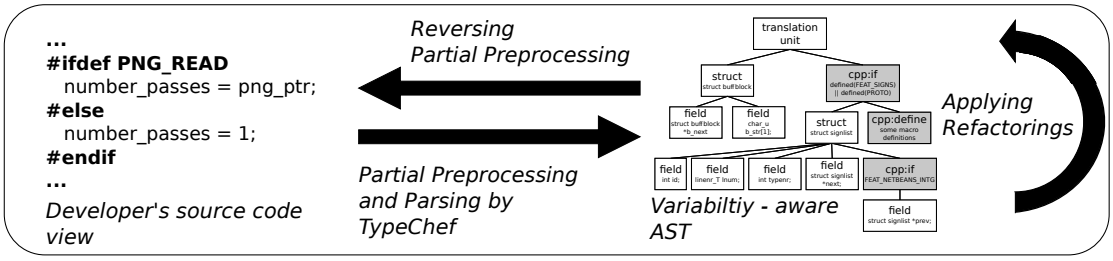



Figure 4.5: Our approach at a glance

In figure 4.5 we illustrate our general approach. As a first step, we parse the source code and perform *partial preprocessing* on the source code. In this step all `#include` directives get resolved and all macros get expanded. In the next step the partial preprocessed code is transformed in a variability-aware single *abstract syntax tree*. These first steps are described in the previous Section 4.2 . Based on the information provided by the type-system of TypeChef, we can perform our refactorings on this AST. The logical next step would be to reverse the partial preprocessing and transform the code into its original representation. Unfortunately, to perform this reversing transformation some further research work has to be done. Because of this limitation, in this thesis we can only show our refactorings on the partially preprocessed code and its *pretty printed* outline, containing fully expanded macros and resolved `#include` directives.

## 4.4 Rename Identifier

In section 3.2.1 we discussed the principle and challenges of performing the refactoring *Rename Identifier*. This section describes our strategy of applying this refactoring on C source code.

### 4.4.1 Requirements

Before applying the *Rename Identifier* refactoring some essential requirements must be fulfilled [2, 23, 19, 24, 25]:

- The new name chosen by the user is valid according to the ISO C standard [26].
- By renaming an identifier and its subsequent occurences no other identifier gets shadowed locally or globally.

### 4.4.2 Mechanism

To apply the refactoring *Rename Identifier*, the following steps have to be performed:

#### 1. Identify the Identifier in the AST

The way how to initiate the refactoring *rename* is in practically the same in every IDE: the developer selects the identifier he wants to rename in the editor window of the IDE, he enters the desired new name, presses enter, and some magic is done in the background; in the end the identifier and all associated identifiers have been renamed. This procedure is also the same in our editor. So the first step in applying this refactoring is to identify the correct corresponding AST entry according the user's selection. During the parsing process, every entry of the AST gets annotated, which position range, namely the starting line and column, and ending line and column, it occupies in the original source code representation. We also retrieve this on the selection range made by the user in the editor window. In order to get all eligible identifiers according to the user selection in the AST, we traverse it, and check each occurring position range of the identifier, if it matches the user's selection range. After traversing the AST, a list of possible identifiers which can be renamed is displayed (see figure 4.6). By selecting the desired identifier, the user can enter the new name and the next step is triggered.

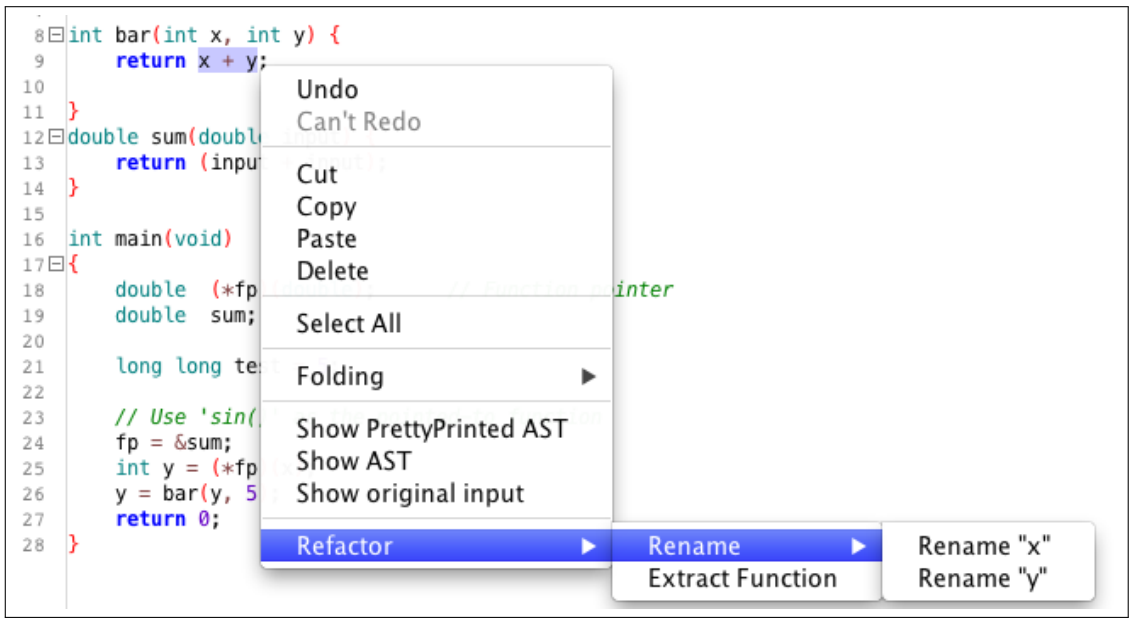



Figure 4.6: Selection menu for renaming an identifier.

### 2. Verify the Chosen New Name

The next step is rather trivial: for a correct refactoring result, the new name of the identifier must be valid as defined in the ISO C standard [26]. Therefore, we verify the chosen name according to the C naming convention as a regular expression and match it against a black list containing all reserved language keywords. In case an invalid name is chosen, the refactoring process is terminated.

### 3. Shadowing and Replacing

After determining the correct identifier and verifying the chosen name, we are finally able to actually perform the refactoring in the AST. Based on our implemented variability-aware declaration/usage map in section 4.2.2 we identify the definitions and all associated usages of the chosen identifier in the next step.

Afterwards, we traverse the list of all associated identifiers: each identifier gets checked for shadowing individually, because especially functions or globally defined variables and their associated identifiers may appear in several different scopes and/or conditional compilation conditions. Therefore, it may occur that only single identifiers are affected by shadowing. In order to inspect if the current identifier and its new name may be affected by shadowing, we use the information provided by the type-system of TypeChef. As mentioned in section 4.2.2 during type-checking process TypeChef creates variability-aware scope environments which contain information about the visibility of identifiers in the current scope.

Contrary to the data necessary to create a mapping between declarations and usages, this information is preserved after successfully finishing the type-checking. Therefore, we are able to determine whether a chosen new name of an identifier is occupied in the current scope of the identifier. Our shadowing detection is performed in a conservative manner: in case we detect an identifier visible in the current scope with the same name, we refuse the renaming, even if introducing new shadowing may have no effect on the observable behavior. We argue that preserving the observable behavior is far more import for the correctness of the refactoring than being able to use the trickiest variable shadowing. Furthermore, in contrast to the motivation of renaming an identifier to improve the source code readability, the explicit use of variable shadowing does not help to improve the readability.

In case new shadowing would be introduced, the refactoring fails and the original input AST is returned. Otherwise, the old identifier object in the AST gets replaced by a new one, named with the chosen name. If the renaming of all references to the identifier succeeds, the refactoring process *renaming* is finished and in the editor window of Morpheus the refactored AST is displayed as *pretty-printed* AST representation.

## 4.5 Extract Function

*Extract Function* is a refactoring operation where selected code gets extracted into a new reusable, standalone function [2]. In this section we present the requirements and describe our strategy in implementing this widely used operation for C source code which is annotated with conditional compilation directives.

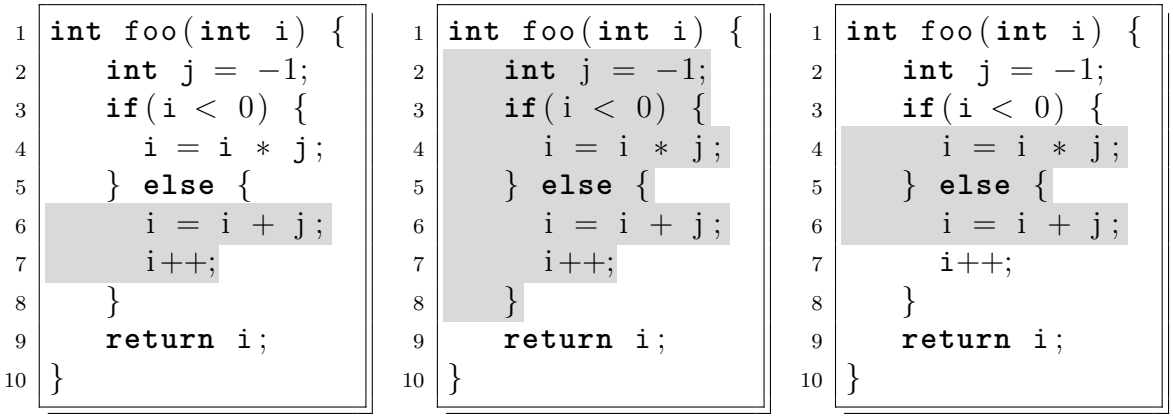
### 4.5.1 Requirements

Compared to the refactoring mechanism *Rename Identifier* presented in Section 4.4, *Extract Function* is considered as a *complex* refactoring technique [19] and several different conditions and constraints must be fulfilled to perform this refactoring mechanism correctly [2, 23, 24, 27]:

1. The selected source is valid for extraction. A selection is valid for extraction if:
   - The selected statements are part of the same function and completely selected.
   - Control statements such as `if`-statements or loops are not selected partly. Figure 4.7 illustrates two valid (Listings a and b) and one invalid (Listing c) selected code fragment for extraction.

- A jump statement is selected together with its jump target as well as all further jump statements with the same target.
- Flow control statements, such as `break` or `continue` statements are selected together with their associated control statement.
- The selection contains no `return` statements.

2. The chosen function name is valid according to the C naming convention and it is *unique* in the current file scope.
3. Used user-defined type-definitions, structures, unions and enumerations are visible inside the scope of the extracted function.




a) Valid selection. b) Valid selection. c) Invalid selection.

Figure 4.7: Three selections for extracting a function.

### 4.5.2 Handling of Function Parameters

Before we describe our mechanism of the refactoring operation *Extract Function* in detail, we first introduce the concept we choose for parameter handling. There are several problems which force us to use a non-trivial strategy. A function in the C programming language, like in every other programming language, takes a variable amount of parameters. By exiting the function either nothing (`void`) or a previously defined data-type is returned to the caller. Normally, function parameters of a C function are passed with the paradigm *call-by-value* [10]. By the use of this paradigm, the calling variables are copied and performed modifications on the passed variables are not visible at the calling location. We are only able to return a single variable or can use dirty workarounds, such as storing the performed

changes in structures or arrays, to return several variables which were changed in the scope of the selection. This brings a major disadvantage to the refactoring operation *Extract Method*: the selection has to finish at a assign expression to a single variable and changes on other variables only occur inside the scope of the selection. The problem is that variables might be modified in the selected part and used later in the unselected part of the function. So the modifications done in the extracted function must be visible to the code left in the original function, otherwise the external behavior would be changed.

In addition, the C programming language knows the concept of *pointers* [10]. Pointers do not store data-objects like normal variables, but instead they store the address of a certain memory region. By dereferencing a pointer, we can retrieve the stored data-object from the stored memory region of the pointer. With this principle, it is possible to pass an argument to a function parameter with the concept of *call-by-reference* instead of passing it with the means of *call-by-value*. Pointers instead, allow us to pass the memory region of a variable as parameter to a function, the concept of *call-by-reference*. As a consequence, we are able to perform changes on the passed variable and these changes are still visible at the calling location.

In our *Extract Function* approach we use the technique of *call-by-reference*. The use of pointers as function parameters enables us to pass several different variables to the function, change them, and all the changes are still visible at the calling location, instead of being limited to only one variable which can be returned. Further, it simplifies the extraction process. We must not assign the return value to the correct variable of the calling place and thereby far more extractions are possible.

As an example for this technique, we look at the selection in the code listing of Figure 4.8 a). The extraction of the selected statements is only possible due to the use of pointers as argument. Without the use of pointers, we would not be able to propagate the occurred swap of the values of variable `x` and `y`, since the C programming only allows us to return one single variable. With the use of pointers, the performed changes are visible in the calling location, as we pass only references to the extracted function in the Listing b) of Figure 4.8. This extraction would not have been possible without the use of *call-by-reference* as described in [10].

### 4.5.3 Mechanism

In this section we present the strategy of our refactoring engine Morpheus for extracting user-selected code fragments into a new resuable stand-alone method.

```c
#include <stdio.h>
int main(void) {
    int x = 100;
    int y = 200;

    printf("x: %d\n", x);

    int temp = x;
    x = y;
    y = temp;

    printf("x: %d\n", x);
    return 0;
}
```

a) Code before extracting the selection.

```c
#include <stdio.h>
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
int main(void) {
    int x = 100;
    int y = 200;
    printf("x: %d\n", x);
    swap(&x, &y);
    printf("x: %d\n", x);
    return 0;
}
```

b) Code after extracting the selected statements.

Figure 4.8: Example of the use of pointers as arguments.

### 1. Verification of the User-Selected Code Fragment for Extraction

A new function that contains user-selected statements which were extracted from another function can only be introduced if the selection is eligible as a body of a stand-alone function. As a consequence, the first step of our strategy is to examine the selected statements for meeting the requirements to be extracted into a new separate function. Our refactoring engine performs this verification process before the user can trigger the refactoring. Only if the selected statements are eligible for extraction, the refactoring mechanism is offered in the refactoring menu of the editor window of Morpheus. Like the previously described refactoring technique *Rename Identifier* we first have to determine the selected nodes in the variability-aware AST based on the user selection in Morpheus' editor window. The process is equivalent to the earlier presented technique in Section 4.4.2, with the difference that this time we look for nodes representing expressions and statements matching the selection range in the editor window instead of their children nodes, which may represent identifiers.

After determining the corresponding AST nodes according to the user-selection, we use a characteristic of the source code representation as variability-aware AST to verify if the selected expressions and statements are part of the same function and are validly selected as illustrated in Figure 4.7. In the AST statements which are nested in a function body or nested in statements with a body, such as `if`-statements, `switch`-statements or loops, are children of a parent node called *compound statement*. In case the selected statements are not part

of the same compound statement or are not even part of a compound statement in general, we detect that the selection is either invalid by statements which are only selected partial or the selection is not part of a function. In both cases, the selection is not eligible for extraction and this refactoring operation cannot be triggered by the user. To illustrate this mechanism we consider Figure 4.7. In Listing a) we see in Line 6 and 7 the selection of two statements which belong to the `else` statement in Line 5. This selection will be identified as two child nodes of the same compound statement representing the `else` branch of the `if`-statement from Line 3 till 8 and is eligible for extraction. Our mechanism to detect the corresponding AST representation of the selected code fragments always chooses the greatest statement node that represents the selection: in Listing b) we see the selection of an assignment statement in Line 2 as well as the selection of the whole `if`-statement from Line 3 till 8. This selection is valid for extraction because the `if`-statement and the assign statement are part of the same compound statement. The assign statements in Line 4, 6 and 7 are children of the `if` statement and as their parent node is part of the selection they are not examined to be in the same compound statement. An invalid selection for extraction is shown in Listing c). Our AST representation detection strategy will return again two nodes for both selected assignment statements in Line 4 and 6. However, this time both nodes are not part of the same compound statement and consequently we identify this selection as not eligible for extraction.

After this first verification step, a deeper examination of the selection is performed. We examine the selection for statements, which might break the control flow by extracting the selected statements. We take a more detailed look at the following *jump statements*, which transfer the control flow unconditionally:

**`goto`:** If the selection contains the jump directive of `goto` or its corresponding `label` statement, the selection will be only eligible for extraction if the statements between the targeted `goto label` and the associated `goto` statements are selected as well. The determination, in case the statements in between are selected as well, is complicated. This is caused due to the fact that the targeted `label` of the `goto` statement may be located before as well as after the location of the `goto` statement. The only limit set by the C programming language is that the j targeted location of the jump must be in the same function scope. Figure 4.9 shows a minimal example of such a variable jump. In Line 10 we see the jump statement `goto` with its target location `GOHERE`. Under condition `A`, our example code fragment would jump to Line 4, otherwise it jumps to Line 16. In order to extract a selection, which contains the `goto` statement of Line 10, correctly in a new function, the complete `while` loop from Line 7 to 13, as well as both function calls of `printf()` in Line 6 and 14, and both targeted `label` statements have to be selected. We verify this by traversing the AST from each `goto` statement to its targeted `label` and by checking if each visited AST node is part of the selection.

```
1  #include<stdio.h>
2  int foo(int i) {
3  #ifdef A
4      GOHERE: i = i * (-1);
5  #endif
6      printf("Start value: %d\n", i);
7      while (i != 0) {
8        i--;
9        if (i < 0) {
10         goto GOHERE;
11       }
12       printf("Current value: %d\n", i);
13     }
14     printf("Finished!\n");
15 #ifndef A
16     GOHERE: i = i * (-1);
17 #endif
18     return i;
19 }
```

Figure 4.9: Variable `goto` jump statement.

**`break` and `continue`:** Both statements may appear in iteration statements such as `for` loop, `while` loop and `do` loop. Additionally, the `break` statement can be used for ending a `switch` statement. In case the user-selected code fragment contains one of the described jump statements, we examine if their parent iteration or `switch` statement is completely selected as well.

**`return`**: A `return` statement returns the function to its caller. In case the selection contains a `return` statement we refuse currently in our approach the refactoring *Extract Function* due to the fact that our approach always generates the extracted function with the return type `void`. As a consequence if the selection contains a `return` statement, it will not be eligible for this refactoring.

If the selected code fragment in the editor window of Morpheus passes the presented verification process the option *Extract Method* will be added to the refactoring menu of the editor (illustrated in Figure 4.6) and can now be trigged by the user.

### 2. Verify the Function Name

By triggering the *Extract Method* option in the refactoring menu of Moprheus, a dialog is prompted for entering the name of the function that should be extracted. In the second step of this refactoring process, we verify the chosen name for potential shadowing and its correctness. As described in the refactoring operation *Rename Identifier* in Section 4.4.2, we verify the chosen name according to the C naming convention as a regular expression and match it against a black list containing all reserved language keywords. Additionally, we examine with the data which are deposited in the type-system of TypeChef if the newly introduced function name is unique in the global scope as well as in the scope where the introduced function will get called.

### 3. Liveness Analysis

To improve the reusability of selected code statements by extracting them from their origin function into a new stand-alone function, we need to generate as a next step the required parameters of the new function that will be introduced. Therefore, we perform a *liveness analysis* of each selected identifier. Liveness analysis computes all variables that are live for a given statement [28, 29]. We adopted and extended this common definition of liveness analysis by defining a variable as *live* in case it is *declared* or *used* outside of the selection scope.

Our methodology to detect live variables relies on the information deposited by the variability-aware type-system of the underlying analyzing and parsing infrastructure TypeChef and the variability-aware declaration/usage mapping which is presented in Section 4.2.2. Furthermore, during the liveness analysis process we examine the visibility of used user-defined data types, such as *type-definitions*, *enumerations*, *structures* and *unions*.

In order to detect the liveness of a variable we use the following strategy. We visit each variable which is part of the selection made by the user in the editor window. First, we examine its liveness outside of the selection scope. Therefor, we retrieve all associated declarations and usages of a variable from our variability-aware declaration/usage mapping. Each returned associated declaration and usage of the currently examined identifier is checked whether it is part of the selection or not. In case of not being part of the selection, the associated variable is stored for a later parameter generation. Second, we inspect the visibility of the used data type for each visited variable. Thus, we exploit the information which is aggregated by the type-system of TypeChef. The type-system reveals the data type of each variable. In case it is a basic type of the C programming language, such as `char` or `int`, no further examination is performed. Otherwise, because of the presence of variability, we look up all declarations and definitions of the revealed data type. We verify their visibility in the scope of the extracted function by being declared

and defined globally. In case a single definition or declaration would not be visible in the scope of the introduced function, the refactoring fails.

#### 4. Prepare Function Parameters and Statements

Based on the data which is retrieved during the previously described *liveness analysis*, we are finally able to generate the function. In order to generate the function, our refacotring engine CREFACTOR performs the following steps:

1. **Generating the function header:** as first step we generate the header of the extracted function which is about to be introduced. A function header in the C programming language consists of *function specifiers*, *return type*, *name of the function* and *function parameters*. These components are generated in this way:

   - *Function specifiers* are generated by adopting the specifiers of the parent function of the selection. They are adopted according to their presence condition.
   - The *return type* is always the same data type: `void`. In our approach, we do not return values from extracted functions as we pass variables, which are marked as live in the liveness analysis process, as *pointer* arguments to the extracted function. As we pass them as pointers, we use the mechanism *call by reference*, where a implicit reference is passed rather than a copy of the variable. That way all changes concerning the variable in the extracted function, can be seen by the caller.
   - We take as the *name* of the function the verified name chosen by the user of step 2.
   - *Function parameters* are generated according to the data, which are aggregated during the liveness analysis process. Each variable which is marked as live is a candidate for a parameter. We revise each candidate for its declaration locations, according their variability. In case a declaration is part of the selection, the refactoring is terminated, because we would have to introduce new declarations at the place where the call to the extracted function would occur. We argue that this is contra productive to the motivation of *Extract Method* in order to improve the structure of the source code. Otherwise, we retrieve the data type of the declaration and add it with the presence condition of its declaration to the parameter list of the function header. Thus, we preserve the variability of multiple declarations of an identifier and its possible variable visibility according to the chosen configuration.

   The extracted function itself gets the same *presence condition* as its original calling function.

2. **Prepare statements:** As a second step we generate the function statement body itself. This process is simple: in the AST we remove the selected statements from their original function. The next step is to introduce them in the new compound statement of the extracted function. Each statement is introduced in the new compound statement surrounded with their previous presence condition. As described in step 1, function parameters are passed as a pointer argument to the function. To map the call-by-reference mechanism into the function body, we have to *deference* all associated variables. Here, we also rely on the information retrieved from the liveness analysis process and convert all marked live variables inside the selection.

#### 5. Introduce Function

Finally, after generating the extracted function, we introduce it. The function gets inserted into the variability-aware AST right before the function where the statements got extracted. Next, we insert a call to the function at the place where the selected statements were originally located. The arguments of the function-call are retrieved from the parameter order of the extracted function header and added according to their variability condition as a parameter.

The refactoring operation *Extract Function* is now finished and after a valid type-checking result by TypeChef, the refactored AST is displayed pretty-printed in the editor window of Morpheus.

## 4.6 Inline Function

As described in Section 3.2.3, inlining a function is the final, complex refactoring mechanism offered by our refactoring engine *Morpheus*.

### 4.6.1 Requirements

The function, which will be inlined, must fulfill the following requirements in order to perform this refactoring correctly [2, 19, 23, 24]:

1. The selected function about to be inlined is neither *recursive* (a function is called recursive if it references itself [30]), nor has it invalid *multiple exit points*, which break the control flow of the function. The appearance of multiple exit points can be broken down into two basic appearances which are illustrated in Figure 4.10. Listing 4.10 a) shows a code fragment with *valid* multiple exit points, whereas Listing 4.10 b) shows *invalid* multiple exit points. A function that contains *invalid multiple exit points* can be defined

as a function with a return value *exiting early*, in case a certain condition is fulfilled.

```
1 int foo(int i) {
2    if(i < 0) {
3      return i;
4    } else {
5      return i++;
6    }
7 }
```

a) Valid multiple exit points.

```
1 int foo(int i) {
2    if(i < 0) {
3      return i;
4    }
5    i++;
6    return i;
7 }
```

b) Invalid multiple exit points.

Figure 4.10: Two functions with multiple exit points; code fragment b) is not eligible for inlining.

There may be possibilities to perform this refactoring technique for recursive functions and functions with multiple exit points, but Fowler states that one should not do this kind of refactoring in the presence of such complexities [2, 24].

2. After inlining, inlined identifiers do not shadow local or global identifiers and vice versa.

### 4.6.2 Mechanism

Morpheus uses the following strategy for inlining a function:

#### 1. Identify the Corresponding AST Representation and its Associated Function-declarations and -calls

Like the previously described refactoring techniques, the first step is to determine the AST representation of the code fragment selected by the user in the editor window of Morpheus. The identifying mechanism is basically the same as described in Section 4.4.2, with the only difference that instead of looking for identifiers in general, we look for identifiers of function-declarations and -calls. If the user-selection contains function declarations or calls, the selected function names are displayed in the inline function context menu and the user can trigger the refactoring process by selecting the desired name of the function. Before the actual refactoring starts, a popup window (Figure 4.11) allows the user to specify two configuration options for this refactoring:

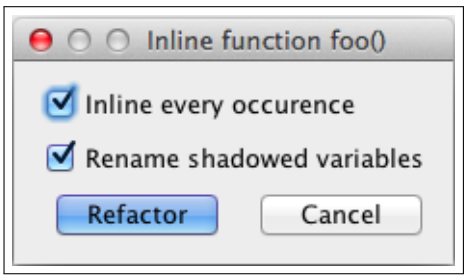



Figure 4.11: Option menu for inlining a function.

- *Inline every occurrence:* By selecting this option, every occurring call associated with the selected function will be inlined at its place and the declarations and definitions of the function get removed from the source code. Otherwise, only the selected function call gets inlined and all other associated calls as well as declarations and definitions remain untouched.
- *Rename shadowed variables:* As described in Section 3.2.1, shadowing is a challenge for all refactorings offered by *Morpheus*. By selecting this option our refactoring engine will rename consistently and the variability-aware, the inlined identifier affected by shadowing. If it is not selected, the refactoring will exit at the identified variable shadowing without applying any refactoring.

After identifying the correct AST representation and configuring the refactoring options, we need to retrieve all associated function calls and declarations. For this purpose, we use the information provided by our implemented *variability-aware declaration/usage map* presented in Section 4.2.2.

### 2. Analyze Function for Recursion and Multiple Exits

The next step is to verify if the function body that should be inlined is eligible for inlining. We have to evaluate two conditions: *recursion* and the presence of invalid *multiple exit points*.

The evaluation process for the *recursive* condition is straightforward: we scan all associated bodies of function-definitions for occurring function calls, and check if it calls itself. If so, the refactoring process is terminated.

Evaluating the condition *invalid multiple exit points* is more complex. In order to determine whether the inlining function has *multiple exit points* or not we examine the AST representation of the inlining function-definitions. Therefor, we look at each occurring `return` statement in the body of the definition; in case no `return` statement is found, we continue with the next step of our inlining function refactor mechanism. Each `return` statement is represented by a node in our variability-aware AST that is generated by TypeChef. We look up each

parent of our node of the return statement until we hit the node representing the definition of the function and evaluate for each visited node if an alternative branch or child node, which fulfills presence condition of our node, exists. If so, the function is not eligible for inlining. Besides, we consider the example in Figure 4.10: in Listing 4.10 a) we can see in Line 3 and 5 two `return` statements, both of their parent node is a node that represents the `if` statement from Line 2 to 7. We examine this node and retrieve that no alternative branch or child node exists. Furthermore, the parent of the node is the function definition. As a result the function is eligible for inlining, containing *valid* multiple exit points. In contrast to Listing 4.10 b): here we also find two `return` statements, one in Line 3, the other one in Line 6. By examining the node representing the `return` statement in Line 3, we retrieve a node for the `if` statement from Line 2 to 4. By applying our detection strategy, we detect that this node has child nodes, which fulfills the presence condition of our examined `return` statement. As a consequence, we detect multiple exit points which are *invalid* for inlining. The refactoring process terminates.

### 3. Inline Statements

Once we verified, that the function is eligible for this refactoring mechanism, we prepare the statements of the function for inlining.

At first, we inspect all occurring identifiers to be inlined for shadowing. The used mechanism is the same as described in Section 4.4.2 for detecting shadowing by renaming an identifier. In case new shadowing will occur by inlining one of the inspected identifiers, we either cancel or continue the refactoring process by renaming consistently all identifiers which are affected by shadowing, based on the selected option of the user to rename shadowed identifiers. Shadowed identifiers get renamed simply by adding an incrementing number to the end of their name until a non-shadowed name is found (referred to Figure 4.12).

The next step is to assign the parameters of the function according to their calling value. Therefore, we take each parameter, match it under the correct presence condition to its associated value in the calling place and introduce it as new declaration statement before the actual inlining statements. An example of this process can be seen in Figure 4.12, in Listing a) we see the function definition of `foo()` with two parameters: `i` and `j`. This function gets called in Line 7 with the variable `i` and the value `7`. After inlining the function, we can see in Listing b), the parameters of `foo()` which got renamed because of shadowing into `i_1` and `j_1`. In line 4 and 5, both variables get declared and assigned with the values of the arguments of the former function call.

The final step of preparing the inlining statements is to remove all occurring

```
1 int foo(int i, int j) {
2     int result = i + j;
3     return result;
4 }
5 int bar() {
6     int i = 5;
7     int j = foo(i, 7);
8     return j;
9 }
```

a) Code before inlining `foo()`.

```
1 int bar() {
2     int i = 5;
3
4     int i_1 = i;
5     int j_1 = 7;
6     int result = i_1 + j_1;
7     int j = result;
8     return j;
9 }
```

b) Code after inlining `foo()`.

Figure 4.12: Example for renaming shadowed variables at inline function.

`return` statements from the statements to be inlined and store them temporarily for a later inlining.

Before we can inline the prepared statements, we have to examine the location of the call where the function should be inlined to. Here, we have to make a distinction between two possible locations (illustrated in Figure 4.13):

- **Direct Call:** the function call occurs directly in a function body as a single statement, optional in the assign or declaration statement of a variable.
- **Nested Call:** the function call is nested as an *expression* into another statement, for example as condition of an `if` statement, or as parameter of an array or another function.

```
1 int foo(int i) {
2     i = i * i;
3     return i;
4 }
5 void bar() {
6     int i = 5;
7     int j = foo(i);
8 }
```

a) Direct Call in Line 7.

```
1 int foo(int i) {
2     return (i * (-1));
3 }
4 int bar() {
5     if (foo(5) > 0) {
6         return 0;
7     }
8     return 1;
9 }
```

b) Nested function call in Line 5.

Figure 4.13: Direct and nested function calls.

The location type of the call determines the way how we inline the function, because at locations where *nested calls* occur, only expressions instead of hole statements are allowed. In order to inline a function as a expression, we use

an extension of GNU C, called *compound statement expression*[3], which allows us to include loops, switches, and local variables within an expression. Logically, a *compound statement expression* does not take function parameters or offers `return` statements, but variables of the current scope are visible inside of the expression. The value of the last subexpression of a compound statement expression serves as the value of the entire construct (see Line 6 of Listing b) in the example inlining of a nested function call in Figure 4.14), otherwise the value of this statement type is `void`.

```
1  int foo(int i) {
2      int j = -1;
3      int res = i * j;
4      return res;
5  }
6  int bar() {
7      if (foo(5) > 0) {
8          return 0;
9      }
10     return 1;
11 }
```

a) Direct call in Line 7.

```
1  int bar() {
2      if ((({
3              int i = 5;
4              int j = -1;
5              int res = i * j;
6              res;
7          }) > 0) {
8          return 0;
9      }
10     return 1;
11 }
```

b) Nested function call in Line 5.

Figure 4.14: Example for inlining a nested function call with the use of compound statement expression.

Finally, we are able to inline the prepared statements, based on their location:

**Direct Call:** here we insert all inlining statements right before the occurring function call. Furthermore, we apply the current presence condition of the call to each statement. Afterwards, we replace the call by the previously removed `return` statement's value. For functions of the return type `void`, we simply remove the call.

**Nested Call:** first, we create the compound statement expression containing all prepared inlining statements. As its last subexpression, we add the value of the previously removed `return` statement. To finalize the inlining strategy, we replace the call by the generated compound statement expression.

In case the user has selected to inline the function at every calling place, this last step is repeated for every occurring call of the inlining function.

[3]A detailed explanation of compound statement expression can be found at: `http://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html`

#### 4. Remove function declarations (optional)

Based on the user's decision earlier to inline the function at occurring calls, we remove the now unnecessary function-declarations and function-definitions from the AST. The result of the refactoring is displayed as *pretty-printed* AST in the editor window of Morpheus.

# 5 Evaluation

In the beginning of this thesis we described in Section 3, that even industrial-strength refactoring engines used by developers on a daily basis fail to preserve the observable behavior and variability by refactoring simple code examples. For an efficient use of an automated refactoring engine, such as Eclipse CDT or our developed engine Morpheus, it is essential that code transformations are performed in a reliable way and the programmer should not be forced to inspect the output for correctness. Proving that our refactoring engine will preserve the observable behavior of the software as well as its variable configuration which is projected in the use of `#ifdef` directives, is an very important and, unfortunately, a non-trivial task [31].

To evaluate the correctness, feasibility and scalability, we apply our refactorings on self-constructed as well as on an open source medium-scale software product line, BusyBox. Thus, two different evaluation strategies are used:

**Manual code review:** We apply our three implemented refactoring techniques on self-constructed examples as well as on BusyBox. Afterwards, we review the output code and compare it with the expected result.

**Automated random refactorings:** In an automated test case, our presented refactoring operation *Rename Identifier* is randomly applied on every source code file of BusyBox. The result is verified by the means offered by our analyzing infrastructure TypeChef, as well as a manual code review on randomly chosen results.

Additionally, we measured the run time of each of our refactoring techniques to prove the scalability of our approach.

## 5.1 Setup

BusyBox[1] is a medium-scale software product line written in C. It is highly configurable, with a total amount 792 unique features, resulting in 206 815 lines of source code divided into 522 different files. Its configuration variability is projected into the C source code base by the use of conditional compilation directives.

[1] We use version 1.18.5 which can be retrieved at `http://www.busybox.net`

The BusyBox toolkit provides the most common UNIX (shell) tools stripped in single, standalone executables. As a real-life software product line, we use BusyBox to verify our refactoring operations on source code fragments, which are uses techniques we would have missed in our self-constructed examples.

As mentioned above, we verify the result by applying our implemented refactoring mechanism on self-constructed example source code fragments. For each refactoring we have chosen different code fragments that contain variant and behavior challenges according to the applied refactorings (e.g. shadowing, variable identifier declaration etc.).

## 5.2 Rename Identifier

We evaluated the refactoring operation *Rename Identifier* with both in Section 5 presented approaches.

### 5.2.1 Automated Refactoring

*Renaming* an identifier is a refactoring mechanism which is used by *every* developer who performs refactoring operations in Eclipse [17]. We consider the correctness and scalability of this basic and heavily used refactoring operation as essential.

To examine the correctness of our approach we developed an automated refactoring verification process. This test chooses randomly an identifier and invokes the refactoring operation *rename identifier* on the chosen identifier. Afterwards, we examine the resulting refactored variability-aware abstract syntax tree by the means offered by TypeChef: we verify its general syntactically correctness by the built-in type-checking mechanism of TypeChef. Further, we evaluate the soundness of this code transformation process by comparing the variability-aware declaration/usage map, presented in Section 4.2.2, with the map before the applied refactoring against the resulting map after refactoring. In case the amount of entries of each declaration and usage stays the same, the refactoring has been applied validly. Otherwise, by missing an associated identifier, the amount of entries in the mapping would change. Besides, an error can be detected during the type-checking process. In both cases, the refactoring is considered as incorrect.

We ran successfully the presented test on all 522 files of BusyBox. On each single source code file of BusyBox, the refactoring operation *renaming* has been applied three times on randomly chosen identifiers.

Table 5.1 shows a selection of the generated result during the automated test run. We have selected these 4 files to show the scalability of our approach. First, we measured the initialization time of TypeChef. Hereby, we measured

| File | Init | IDs | Identifier | Amount | Refactor | 1/IDs |
|---|---|---|---|---|---|---|
| data_align.c | 2390 ms | 8490 | `boundary` | 4 | 196ms | 49ms |
| expand.c | 2263 ms | 9337 | `size_t` | 265 | 10238ms | 38ms |
| unzip.c | 1778 ms | 8761 | `sigset_t` | 11 | 473ms | 43ms |
| rmdir.c | 1635 ms | 8221 | `pid_t` | 96 | 3968ms | 41ms |

Table 5.1: Runtime for random *renamings* on selected source code files of the BusyBox tool-suite.

how long the parsing and type-checking processes take. It can be seen that for source code files from the BusyBox tool-suite the initialization process is finished latest after about 2.5 seconds. Then, we determined the amount of identifiers in the variability-aware AST after the *partial preprocessing* process. Due to the fact that during this process all `#ifdef` directives get resolved, we have a large amount of identifiers in the AST. The column *identifier* shows the identifier, our automated test process renamed in this test and the column *amount* the total amount of this identifier with its associated identifiers. In the column *refactor* we can see the total runtime of the refactoring. It can be stated that even for a large amount of identifiers which have to be renamed, the operation is performed in a reasonable time, rename a total amount of 265 identifiers takes about 10 seconds. The last column *per id* shows the runtime for a single identifier. It can be seen our approach scales and for each identifier it takes only about 40ms to rename it. *Pretty printing* the refactored variability-aware AST into the editor window Morpheus is not part of this table, because we have measured for all files of the BusyBox a duration for printing the AST with a highest time expenditure of 50 ms.

### 5.2.2 Manual Code Review

To validate the correctness of our renaming strategy, we constructed several test cases. We present two of them, one that covers the challenges of the presence of variability and the second one that illustrates the problem of shadowing.

#### Variability

First, we evaluate the correctness in the presence of variability. For this purpose, we constructed two example code fragments, one that contains different declarations of a variable and another one which is slightly modified. Instead of a variable declaration of an identifier, this time the declarations and associated identifiers can only be chosen in an exclusive configuration condition.

The constructed C source code is illustrated in in Figure 5.1 a). In this code

```
1  void foo(int j) {
2  #ifdef A
3      int i;
4      i = j;
5  #else
6      float i;
7      i = 16.11;
8  #endif
9      i = 2 * i;
10 #ifdef A
11     i = i + 2;
12 #endif
13 }
```

a) Code before renaming `i`.

```
1  void foo(int j) {
2  #ifdef A
3      int r;
4      r = j;
5  #else
6      float r;
7      r = 16.11;
8  #endif
9      r = 2 * r;
10 #ifdef A
11     r = r + 2;
12 #endif
13 }
```

b) Code after renaming `i` into `r`.

Figure 5.1: Example for renaming a variable with different declarations.

fragment we want to rename the variable `i` in Line 9 into `r`. The variable `i` gets declared variable: under condition `A` it is declared as an `int` in Line 3, otherwise it is declared with the data-type `float` in Line 6. To preserve its variability and external observable behavior, all identifiers associated with both declarations need to be renamed. Figure 5.1`b` shows the result after renaming the variable into `r` with Morpheus. We can see, all associated identifiers of both declarations have been renamed correctly.

Figure 5.2 a) shows a modified version of the previously evaluated source code example in Figure 5.1 a). In both example code listings the variable `i` is prominently used, but in Figure 5.2 a) the declarations and usages of `i` in each configuration do not intersect. Like in Figure 5.1 a) variable `i`, gets declared in Line 3 and 6 under different conditions and with different data-types. But a usage, which will occur in both configuration conditions, does not appear. As we rename the variable `i` in Line 10 into `r`, only the associated identifiers of the declaration of Line 3 should be renamed. In Figure 5.2 b the result of applying this refactoring with Morpheus can be seen. Only the variables under the configuration `A` have been renamed, whereas the the variables of the other configuration remain untouched. The refactoring is correct.

Additionally, we tried to rename the variable `i` into `j` in both cases first. Our refactoring engine refused this refactoring, because j has already been declared as function parameter.

```
1  void foo(int j) {
2  #ifdef A
3      int i;
4      i = j;
5  #else
6      float i;
7      i = 16.11;
8  #endif
9  #ifdef A
10     i = i + 2.0;
11 #else
12     i = i + 2;
13 #endif
14 }
```

a) Code before renaming `i`.

```
1  void foo(int j) {
2  #ifdef A
3      int r;
4      r = j;
5  #else
6      float i;
7      i = 16.11;
8  #endif
9  #ifdef A
10     r = r + 2;
11 #else
12     i = i + 2.0;
13 #endif
14 }
```

b) Code after renaming `i` into `r`.

Figure 5.2: Example for renaming a variable with optional presence.

```
1  #ifdef A
2  int i = 5;
3  #endif
4  int foo(int j, int h) {
5  #ifdef A
6      j += i;
7  #endif
8      return j + h;
9  }
10 int bar(int y) {
11 #ifndef A
12     int i = y;
13 #endif
14     return i * y;
15 }
16 int foobar(int i) {
17     return i + i;
18 }
```

a) Code before renaming `i`.

```
1  #ifdef A
2  int r = 5;
3  #endif
4  int foo(int j, int h) {
5  #ifdef A
6      j += r;
7  #endif
8      return j + h;
9  }
10 int bar(int y) {
11 #ifndef A
12     int r = y;
13 #endif
14     return r * y;
15 }
16 int foobar(int i) {
17     return i + i;
18 }
```

b) Code after renaming `i` into `r`.

Figure 5.3: Example for renaming a variable in the presence of shadowing.

#### Shadowing

We evaluate the correctness of our refactoring approach in the presence of occurring shadowing caused by `#ifdef` directives. In Figure 5.3 a) one of our example code fragment for shadowing is listed. In Line 2 a global variable `i` gets declared in case feature `A` is selected. In the function body of `foo()` a further declaration of `i` can be seen in Line 12, in case feature `A` is not selected. Furthermore, in Line 16 the function `foobar()` gets declared with the parameter `i`. This parameter declaration shadows the globally declared variable `i` in the scope of the function `foobar()`.

First of all, we try to rename the declaration of `i` in Line 2. This refactoring fails correctly in our engine Morpheus, because after this renaming the occurrence of `i` in Line 6 would be shadowed by the function parameter `h` of the function `foo()`. Then we renamed `i` into `r`. The result in 5.3 b) shows that all associated variables have been renamed correctly, as well as the optional declaration of `i` in Line 12 to preserve the variability and observable behavior. Furthermore, the variable `i` declared as parameter of the function `foobar()` and its associated usages have not been renamed, as they are not a part of the globally declared variable.

## 5.3 Extract Function

*Extract Function* is a complex refactoring technique which needs to meet several requirements, as described in Section 4.5, to be applied correctly. As we are currently unable to evaluate the correctness of complex refactoring mechanism in an automated way, because useful correctness tests are missing. The correctness and scalability of our approach has been evaluated on self-constructed code fragments, covering a selection of challenges of refactoring C code and on our presented tool-suite BusyBox. BusyBox has been used in this evaluation to verify the correctness of our approach by the means of manual code review and for measuring the performance and the scalability of our engine.

### 5.3.1 Self-Constructed Code Fragment

During the implementation of our strategy for the refactoring operation *Extract Function* we constructed some sample code fragments to evaluate the correctness of our approach. In Figure 5.4 a) we see a example code fragment for this purpose. This example code listing focuses on the challenges in different data-types of a parameter of the extract function, the correct determination of the data-type of a parameter and our the avoidance of shadowing in strategy.

```
1  int i = 1;
2  typedef unsigned int unit;
3  struct lib {
4  #ifdef A
5      unit side;
6  #else
7      int side;
8  #endif
9      char title[30];
10 };
11 void foo() {
12     struct lib book;
13 #ifdef A
14     unit j = 5;
15 #else
16     int j = 3;
17 #endif
18     if (i > 0) {
19         i++;
20         printf("%d", i);
21     }
22     book.side = j;
23 }
```

a) Code before extracting the selected statements.

```
1  int i = 1;
2  typedef unsigned int unit;
3  struct lib {
4  #ifdef A
5      unit side;
6  #else
7      int side;
8  #endif
9      char title[30];
10 };
11 void bar(
12 #ifdef A
13     int *j
14 #endif
15 #ifndef A
16     unit *j
17 #endif
18     , struct lib *book,
       ↪ int *i) {
19     if ((*i) > 0)) {
20         (*i)++;
21     }
22     (*book).side = (*j);
23 }
24 void foo() {
25     struct lib book;
26 #ifdef A
27     unit j = 5;
28 #else
29     int j = 3;
30 #endif
31     bar(&j, &book, &i);
32 }
```

b) Code after extracting the selected statements.

Figure 5.4: Self-constructed code example for evaluating the refactoring operation *extract function*.

In the selection from Line 18 to 22 in Figure 5.4 a) variable `i`, which is declared globally, as well as variable `j` with a different data-type declaration according to which configuration has been selected and a structure `book` are selected for the refactoring operation *extract method*. The result after performing this refactoring operation on the selected statement is shown in Figure 5.4 b) with the newly introduced function `bar()` in Line 11 and call to this function in Line 31. We can see, as all variables are referenced outside of the scope of the extracted function, all parameters of the call have been correctly created as pointers. As described before, variable `j` has different data-types, according to the chosen configuration. In Line 13 and 16, parameter `j` has been introduced as pointer surrounded with the correct `#ifdef` directive. Additionally, in Line 18 the referenced structure with its correct data-type and the former global variable `i` are also passed as pointer. Due to the fact that we pass all externally referenced variables as pointer to the extracted function, we avoid the problem of possible shadowing as well as our extracted function still updates globally declared variables about modifications by the means of pointers.

### 5.3.2 BusyBox

In addition to our self-constructed code-fragments, we applied our refactoring *Extract Function* on the medium-scale SPL BusyBox. Because of its notably greater size compared to our samples and its real-life usage, we used BusyBox for evaluating the correctness with manual code review and used it especially to prove the scalability of Morpheus. In Table 5.2 a selection of different files of the BusyBox tool-suite is shown. On these files, we applied some refactoring on different functions and different selection complexities. We detected, our selection validation strategy is very expensive, as we selected in the file `unzip.c` only a small `if` statement, whereas our liveness analysis scales in a constant way, even for bigger amounts of statements as to be seen in the runtime of `telnet.c`.

| File | Init | Selected Stmts. | Parameter | Refactor | 1/Sel.Stmts. |
|---|---|---|---|---|---|
| man.c | 2569 ms | 9 | 3 | 1202ms | 134ms |
| telnet.c | 2263 ms | 24 | 10 | 1905ms | 80ms |
| unzip.c | 1778 ms | 3 | 4 | 583ms | 194ms |
| init.c | 2488 ms | 8 | 3 | 1117ms | 139ms |

Table 5.2: Runtime for different *extracted methods* on selected source code files of the BusyBox tool-suite.

## 5.4 Inline Function

We evaluated the correctness and scalability of the refactoring operation *inline function* with manual applied function inlining on self-constructed source code examples as well as on randomly selected source code files of the BusyBox tool-suite.

### 5.4.1 Self-Constructed Code Fragment

Figure 5.5 a) shows one of our self-constructed example code fragments. In this example code fragment we want to inline the function `foo()`. This function gets called twice in the scope of `bar()`. The call of `foo()` in Line 7 is nested into a `if`-statement, whereas the call of `foo()` only occurs under condition `A` in Line 14. A further function `foo()`, like function `bar()`, has two parameters `x` and `y`, which will lead to shadowing after the inlining process. Additionally, because of the fact that the function gets inlined twice in `bar()`, the variable `res` of the function `foo()` will lead to shadowing.

```
1  int foo(int x, int y) {
2      int res = x + y;
3      return res;
4  }
5
6  int bar(int x, int y) {
7      if (foo(x, y)) {
8          return 0;
9      }
10
11     int i = 5;
12
13 #ifdef A
14     i = foo(i, 7);
15 #endif
16
17     return i;
18 }
```

a) Code before inlining `foo()`.

```
1  int bar(int x, int y)  {
2      if (({
3        int x_2 = x;
4        int y_2 = y;
5        int res_1 = x_2 + y_2;
6        res_1;
7      })) {
8        return 0;
9      }
10     int i =  5;
11 #ifdef A
12     int x_1 = i;
13     int y_1 = 7;
14     int res = x_1 + y_1;
15     i = res;
16 #endif
17     return i;
18 }
```

b) Code after inlining `foo()`.

Figure 5.5: Self-constructed code example for evaluating the refactoring operation *inline function*.

Figure 5.5 b) shows the result after inlining the function `foo()` at every

occurring call and enabled renaming in case of shadowing. We can see, all affected variables have been renamed consequently. Further, in the `if`-statement in Line 2, the function has been inlined correctly as compound statement expression. The call to `foo()` of the original code fragment occurs only if feature `A` is selected. In the refactored result, we notice that our engine has preserved this variability by surrounding the inlined statements with the same presence condition of the call.

### 5.4.2 BusyBox

| File | Function | Stmt | Calls | Refactor | 1/Call | 1/Call*Stmt |
|---|---|---|---|---|---|---|
| diff.c | `add_to_dirlist` | 8 | 2 | 2547ms | 1273ms | 159ms |
| time.c | `run_command` | 12 | 1 | 693ms | 693ms | 57ms |
| showkey.c | `xset` | 3 | 2 | 1858ms | 929ms | 309ms |
| httpd.c | `setenv1` | 1 | 20 | 15061ms | 753ms | 753ms |

Table 5.3: Runtime for random *inlined functions* on selected source code files of the BusyBox tool-suite.

We also applied the refactoring *Inline Function* on the BusyBox tool-suite and validated the results with manual code reviewing of the result. Further, we used BusyBox to evaluate the performance and scalability our implementation. In Table 5.3 we can see the result of this validation on selected files of BusyBox. The table lists for each file the selected function which we have inlined and the amount of statements as well as the associated calls to the function. We measured the duration of the whole refactoring process and show it in column *refactor*. The other columns show the refactoring time per amount of calls in the program and per product of calls and statements, respectively. Our refactoring approach scales to the amount of occurred calls instead of the amount of statements to be inlined. This is caused due to the fact that after each single inlining of a call, we have to perform the type-checking process of TypeChef again in order to detect possible new shadowing, which is caused by the introduction of the inlined variables at the calling place. Nevertheless, we can see that our approach performs in a reasonable time, even for functions that are called frequently.

# 6 Related Work

Due to the extensive usage of the C programming language since the late 1970s for almost every major operating system, popular (open source) software systems and programming embedded devices, several academic and industrial research work on specifying, implementing, and verifying refactorings for C code have been made.

Especially Garrido et al. did some extensive year-long research work on this area [18, 19, 32, 33]. She defined a catalog of possible refactoring techniques, applicable on the C programming language. Furthermore, she developed an heuristic approach to face the challenges in refactoring C source code in the presence of conditional compilation directives. These heuristics have been implemented by her in a small refactoring browser called CRefactory, which offers some basic refactorings such as *renaming* C entities and macros, deleting unreferenced variables, or moving variables to a structure. Complex refactoring techniques, like *extract method*, have only been specified formally. Even though, the used heuristics to detect `#ifdef` variability and preserve them during the code restructuring process are not complete and sound.

Spinellis developed CScout, a web based refactoring engine for C code [34, 35]. CScout performs some analysis on C source code in the presence of CPP directives and offers the ability to rename and to remove unused identifier; complex refactorings are not offered. His analysis approach, as well as Garrido's, is heuristic based. He states that the presence of conditional compilation directives in the source code causes the parsing process to be guided by hand, because his heuristics are not complete. Furthermore, the absence of a variable shadowing detection leads in some cases to an invalid, observable behavior changing the refactoring result.

A brute force attempt in preserving the variability of software while applying refactorings on C code has been made by Vittek and his developed refactoring tool for source code written in Java and C, Xrefactory [36]. As proof of concept, he implemented the refactoring mechanism *renaming an identifier*.

Hafiz and Overbey recently also started implementing a variability-aware refactoring engine [37]. With their tool OpenRefactory/C they aim to implement until August 2014 basic refactoring mechansim such as renaming, extract method, move method, extract local variable as well as static analysis features such as control flow analysis or data flow analysis. Currently they are able to rename

variables in source code not annotated with conditional compilations directives[1].

In order to verify the preservation of the observable behavior on applying refactorings, Cavalcanti et al. presented their approach SafeRefactor [38]. They developed a plugin for Eclipse, which generates automated test cases on the selected code fragment before applying refactoring on Java source code. After the code transformation process is finished, these generated test cases are run again and compared with the pre-refactoring result. This way they detect, if the external observable behavior of the refactored code fragment has changed. They have shown, that their approach detects broken refactorings in the refactoring engine of Eclipse for Java source code with no false positives.

[1] A web demo of their refactoring engine can be tested under:
`http://www.openrefactory.org/demo.html`

# 7 Conclusion

In this thesis we have presented an approach for refactoring C code in the presence of conditional compilation directives while preserving its *variability* and *observable external behavior*. We developed, as proof of concept of our approach, the refactoring editor Morpheus on top of the variability-aware source code analyzing and parsing infrastructure TypeChef. Morpheus is offering one basic refactoring technique, *Rename Identifier*, and two complex refactoring techniques, *Extract Function* and *Inline Function*. With this tool we have shown that it is possible to refactor successfully C source code enriched with CPP directives without using an incomplete heuristic approach or a very expensive bruteforce attempt.

We outlined in Section 3, that current state-of-the-art IDEs for C code are unaware of source code variability which are represented by `#ifdef` directives. As consequence, their offered refactoring operations for variable C source code are broken by not preserving its variability or external observable behavior. Furthermore, we discussed the challenges in implementing variability preserving refactorings.

The implemented variability-aware refactoring engine as subproject of TypeChef was described in Section 4. In short, we explained how the used source code analysis and parsing infrastructure works. We illustrated in detail our strategy for each refactoring mechanism individually by using the variability-aware abstract syntax tree and type-system provided by TypeChef.

We evaluated the correctness and completeness of our presented approach in Section 5 on self-constructed code examples as well as on a real-life, medium-scale software product line, BusyBox. BusyBox is a highly configurable software product line with a total amount of 792 unique features and 206815 lines of C source code. The variability of this software product line is introduced via conditional compilation directives into the source code. Besides, we have shown that our refactoring strategies can be applied in a reasonable time on C source code which is annotated with `#ifdef` directives.

## 7.1 Future Work

Unfortunately the use of TypeChef as parsing and analysis infrastructure for C source code annotated with preprocessor directives brings one major disadvantage

for a productive use of the Morpheus refactoring engine in day to day developing work: as described in Section 4.2 during the parsing process TypeChef performs *partial preprocessing* on the input source code which results in a source code representation with fully expanded macros and resolved `#include` directives. Currently we are unable to reverse this process. Due to this fact, our refactoring tool can successfully refactor source code annotated with `#ifdef` directives, but cannot restore the original source code representation with applied refactorings, non-expanded macros and non-resolved `#include` directives. During the adaption of our presented refactoring strategies we have found a possible approach for this reversing mechanism. A solution may be a deeper mapping between the original source code representation and its corresponding nodes in the AST. After the refactoring, we could map the transformation changes in the AST nodes back to the corresponding original source code construct. Especially the refactoring operation *Rename Identifier* may benefit from this approach and behave as it is known in current state-of-the-art IDEs but without changing the observable external behavior while preserving the variability of the refactored source code.

Furthermore, the refactoring engine can be extended with further refactoring techniques, such as making a variable global, converting a variable into a pointer and vice versa or refactoring a preprocessor directive. An example for this purpose would be inlining a macro or renaming a macro.

For a complete IDE-like user experience, our refactoring engine as well as the utilized parsing and analysis infrastructure needs to become aware of the dependencies within the build system of a software product (line) to apply refactoring, such as renaming variables and functions declared in header files, globally on all source code files affected by the performed refactoring.

To prove the correctness and soundness of our refactorings in a incontrovertible way, an automated testing environment, like SafeRefactor for source code written in Java [38], is required for C source code.

In order to make an actual contribution to the daily work of programmers who develop software product lines in C, it would be a great benefit to implement our refactoring engine as a plugin for widely distributed IDEs, for example Eclipse.

# 8 Acknowledgement and Tool Availability

I thank Prof. Christian Lengauer, Ph.D. and Dipl. Ing.-Inf. Jörg Liebig for supervising this thesis. Special thanks to Jörg Liebig for his substantial support and constant guidance and for giving me the opportunity of participating in current research. Furthermore, I would like to thank the Chair of Programming at the University of Passau for hiring me as a student assistant during the time implementing the presented refactoring engine Morpheus and writing this thesis. I want to thank all staff members who helped me with their suggestions and time, especially Alexander von Rhein for reviewing this thesis in its very late phase. For our cooperative work in developing the variability-aware declaration/usage map CDeclUse, I thank my fellow student assistant Florian Garbe. I am thankful for my friend Andrea Muhr for correcting my spelling and grammar in this thesis.

The developed refactoring editor Morpheus is published as open source under GPL 3.0[1]. Morpheus is still an ongoing work and the current development state can be retrieved with all necessary dependencies at:

`https://www.github.com/aJanker/TypeChef`.

The development state of Morpheus at this stage is documented at:

`https://www.github.com/aJanker/TypeChef/archive/v0.1.zip`.

[1] A copy of the license can be retrieved at:
`http://www.gnu.org/licenses/gpl-3.0.html`

# Bibliography


[1] 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 *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824. ACM Press, 2011.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 2001.

[3] The GCC Team. The preprocessing language - the c preprocessor, January 2013. URL `http://gcc.gnu.org/onlinedocs/cpp/The-preprocessing-language.html#The-preprocessing-language`.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. 2002.

[5] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, 2012.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. *Algebraic Methodology and Software Technology*, pages 36–50, 2008.

[7] M. Anastasopoulos and C. Gacek. Implementing product line variabilities. *Software Engineering Notes*, 26(3):109–117, 2001.

[8] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[9] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 23–34. IEEE, 2007.

[10] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 105–114. IEEE, 2010.

[12] J. Jones. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design*, 2003.

[13] J. Overbey and R. Johnson. Generating rewritable abstract syntax trees. *Software Language Engineering*, pages 114–133, 2009.

[14] J. Michelotti, J. Overbey, and R. Johnson. Toward a language-agnostic, syntactic representation for preprocessed code. *Proc. 3rd Work. on Refactoring Tools (WRT)*, 2009.

[15] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-preserving refactoring in feature-oriented software product lines. In *Proc. Int. Conf. Systems Engineering and Modeling (ICSEM)*, volume 6, pages 73–81. ACM, 2012.

[16] I. Sommerville. *Software Engineering*. International Computer Science Series. 2007.

[17] G. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.

[18] A. Garrido and R. Johnson. Challenges of refactoring C programs. In *Proc. Int. Workshop Principles of Software Evolution (IWPSE)*, pages 6–14, 2002.

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

[20] C. Kästner, P. Giarrusso, and K. Ostermann. Partial Preprocessing C Code for Variability Analysis. In *Proc. Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 127–136. ACM Press, 2011.

[21] A. Kenner, C. Kästner, S. Haase, and T. Leich. Typechef: toward type checking #ifdef variability in c. In *Proc. Workshop on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010.

[22] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. 2012. to appear.

[23] W. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, 1992.

[24] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In *ACM SIGPLAN Notices*, volume 45, pages 286–301. ACM Press, 2010.

[25] R. Ramos, E. Piveta, J. Castro, J. Araújo, A. Moreira, P. Guerreiro, and M. Pimenta. Improving the quality of requirements with refactoring. *Simpósio Brasileiro de Qualidade de Software*, 6:312–318, 2007.

[26] ISO JTC. Iso/iec 9899: 2011. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm`.

[27] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[28] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Priniciples, Techniques, and Tools*. Pearson/Addison Wesley, 2007.

[29] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Large-Scale Variability-Aware Type Checking and Dataflow Analysis. Technical Report MIP-1212, Department of Informatics and Mathematics, University of Passau, 2012.

[30] E. Dijkstra. Recursive programming. *Numerische Mathematik*, 2(1):312–318, 1960.

[31] M. Schäfer, T. Ekman, and O. de Moor. Challenge proposal: Verification of refactorings. In *Proc. Int. Symp. Principles of Programming Languages (POPL)*, pages 67–72. ACM, 2009.

[32] A. Garrido. Software refactoring applied to c programming language. Master's thesis, University of Illinois, 2000.

[33] A. Garrido and R. Johnson. Analyzing multiple configurations of a c program. In *Proc. Int. Conf. Software Maintenance (ICSM)*, pages 379–388. IEEE, 2005.

[34] D. Spinellis. Cscout: A refactoring browser for c. *Science of Computer Programming*, 75(4):216 – 231, 2010.

[35] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, November 2003.

[36] M. Vittek. Refactoring browser with preprocessor. In *Proc. European Conf. Software Maintenance and Reengineering (CSMR)*, pages 101 – 110, march 2003.

[37] M. Hafiz and J. Overbey. Openrefactory/c: An infrastructure for developing program transformations for c programs. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 27–28. ACM, 2012.

[38] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, and M. Cornélio. Saferefactor-tool for checking refactoring safety. *Tools Session at SBES*, pages 49–54, 2009.

# Statutory Declaration

Hereby I declare that I have written this bachelor thesis by my own. Furthermore, I confirm that no other sources have been used than those specified in the bachelor thesis itself. This thesis, in same or similar form, has not been made available to any audit authority yet.

Passau, March 21, 2013

Andreas Janker