University of Passau
Department of Informatics and Mathematics

Chair for Programming
Prof. Christian Lengauer, Ph.D.

Bachelor Thesis

Configuration Lifting for C Software Product Lines

Florian Garbe

Date: 02.04.2013

Supervisors: Prof. Christian Lengauer, Ph.D.
Dipl. Ing.-Inf. Jörg Liebig

Supervisor Contacts

Prof. Christian Lengauer, Ph.D.
Chair for Programming
Universität 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
Universität Passau
E-mail: joerg.liebig@uni-passau.de
Web: http://www.infosun.fim.uni-passau.de/cl/staff/liebig/

Statutory Declaration

University of Passau
Department of Informatics and Mathematics

I assure that this thesis is a result of my personal work and that no other than the indicated aids have been used for its completion. Furthermore I assure that all quotations and statements that have been inferred literally or in a general manner from published or unpublished writings are marked as such. Beyond this I assure that the work has not been used, neither completely nor in parts, to pass any previous examination.

....
(Date) (Signature)

Contents

Contents iv

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objective . . . 2
1.3 Structure of the Thesis . . . 3

2 Background 4
2.1 Software Product Lines . . . 4
2.2 The C Preprocessor . . . 6
2.2.1 File Inclusion . . . 6
2.2.2 Text Substitution . . . 7
2.2.3 Conditional Compilation . . . 7
2.2.4 Problems of CPP Usage . . . 9
2.3 Current Analysis Methods for C Product Lines . . . 10
2.4 TypeChef . . . 11

3 Approach 15
3.1 `#ifdef` to `if` Transformation . . . 15
3.1.1 General Idea . . . 15
3.1.2 Embedding Statements into `if` Statements . . . 16
3.1.3 Renaming Declarations . . . 17
3.1.4 Code Duplications . . . 18
3.1.5 Combining Transformations . . . 19
3.2 Problems . . . 21
3.2.1 Renaming Declarations and Function Definitions . . . 21
3.2.2 Keeping Code Duplications Minimal . . . 21
3.2.3 `Switch-Case` Blocks . . . 22
3.3 Solutions . . . 23
3.3.1 Declaration Use Map . . . 23
3.3.2 Reconstruction of `#if`, `#elif` and `#else` Conditions . . . 25
3.3.3 Brute-Force Code Duplications . . . 26

4 Evaluation 27
4.1 BusyBox Case Study . . . 27
4.2 Results . . . 31

4.3 Problematic Aspects 33

5 Related Work **36**

6 Conclusion **38**

7 Acknowledgments and Project Repository **40**

List of Figures **41**

Bibliography **42**

1 Introduction

1.1 Motivation

The way we manufacture things has changed drastically over the past decades. Products such as cars, used to be only individually crafted for each customer. But as more and more people were able to afford buying products, the manufacturing process changed leading to new production techniques, so called product lines. Ford was the first to utilize production lines in the automobile industry. Mass productions were much cheaper than building cars individually but after a while people were not satisfied with the standardized products. They wanted to be able to customize a product according to their needs. For example a big family needs a different vehicle than a person who is mainly driving alone. Thus began the time of mass customization in production lines which combines the advantages of cheaper production without missing out on a certain degree of individuality. Many companies developed different platforms for different parts used in their production processes. These platforms lead to a larger variety of products and lower production costs.

The same idea of generating individual products out of reusable components has been applied to software development. These so called software product lines combine several advantages of individual and standard software. On the one hand they make it possible to let the client choose which variant functionality of the product line he wants to include thus giving him a certain extent of customization. On the other hand these product lines offer improved quality through reuse because their functions are tested under different conditions.

Consequently software product line engineering has become more and more popular among software companies over the last decades. There are several advantages to developing a software product line instead of developing a single product. One advantage is that it is cheaper to build and maintain different products that share many functionalities and can be used for multiple systems. This leads to decreased development costs because the developing effort is greatly reduced.

Today there are different approaches for expressing and resolving variability in software product lines. In the context of product lines written in the C programming language the preprocessing tool CPP is a very commonly used tool.

CPP provides means to express different variants in the source code using

annotations. In order to generate a product CPP then resolves these annotations based on a feature selection and generates code. These product lines are also called preprocessor-based product lines.

With the help of software product line engineering one can develop software product lines and let the customer generate his very own program built according to his needs. The amount of possible different software product variants is possibly exponential to the number of features of the product line.

But this exponential variant diversity can be a huge challenge for the development and maintenance of product lines:
Software product lines, just like other software products, have to be tested. Software is never free of errors or unexpected behavior and these mistakes can cause huge damages. Most software is also far too complex to be analyzed manually so one has to use automated software verification tools to ensure that the software is safe.

Existing software verification or analysis tools such as type checking or model checking have been developed for single software systems only. But different products of the same product line have a varying grade of similarities and the current tools don't employ these similarities for their analysis which could save a lot of computation time. In order to check a product line with the existing methods, one would have to generate every possible product and run these tools sequentially on each product instead of just running it on the whole product line. Considering that the number of different and independent products one can create out of a product line with 33 optional features (2^{33}) already exceeds the population of the earth; generating these products is not feasible and if it is running analysis such as type checking on each of them produces a lot of redundant computations.

Software product line analysis strategies are being developed, but there are many different approaches such as family-based analysis or feature-based analysis and more [1]. These strategies keep track of similarities and differences between products to eliminate redundant computations. However, it is difficult to find the strategy that is best for a given product line and some of them do not scale very well while, others can not guarantee the absence of errors in a product line because they only check a portion of all possible products.

1.2 Objective

The objective of this thesis is to develop a tool for C software product lines, which rewrites a given product line to enable us to use the existing analysis and verification tools. This technique is called configuration lifting [2] and it turns the different product variants of a product line into a meta-product.

This meta-product includes the properties of all possible products. Configuration lifting only requires the transformation and generation of one product, instead of generating all possible products and eliminates the need of rewriting existing analysis and verification tools to make them variability-aware. We make this transformation by replacing the static variability of CPP annotations with runtime variability in the C programming language.

1.3 Structure of the Thesis

Section 2 introduces basic terms and definitions and summarizes important facts about background aspects concerning our work. Section 3 describes our approach for configuration lifting in general and points out problematic aspects of this approach. In Section 4 we evaluate our approach with a real world product line BusyBox and provide different statistics of the transformation process. The last section concludes our work and illustrates future work.

2 Background

This section presents definitions of important concepts in the environment of preprocessor-based product lines.

2.1 Software Product Lines

> Clements et al. [3]: "A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

This definition focuses on the core aspects of a software product line. They were especially designed to offer several advantages compared to creating individual software systems and to reduce the development costs, time to market and even decrease labor needs.
These advantages are achieved by strategic and planned reuse of core aspects of a product line among its derived products. This reuse explains the similarities between the different products of a product line [4].

Software product lines have to be able to express and manage variability, which enables the generation of different products in the first place. This is generally done through language extensions or in our case through preprocessor use. There can also be dedicated tool support for product generation. This is the case for BusyBox and the Linux kernel, where a framework called kConfig is responsible for product derivation.

Features are a central abstraction that is used in software product line engineering. Customers or developers generate different program variants from a product line by selecting a set of enabled features. We will now look at two different definitions of the term feature.

1. Kang et al. [5]: "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"

2. Apel et al. [6]: "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement,

to implement and encapsulate a design decision and to offer a configuration option"

The first definition is rather abstract and the second one is more technical. Features are often used as a communication basis between customers and engineers and manifest themselves as parts of software. They also show the differences and similarities between different products of the same product line [7].

We now have a set of different features for a product line. However not all possible combinations of feature selections are usually valid. The structure, which defines valid feature combinations, is called feature model.

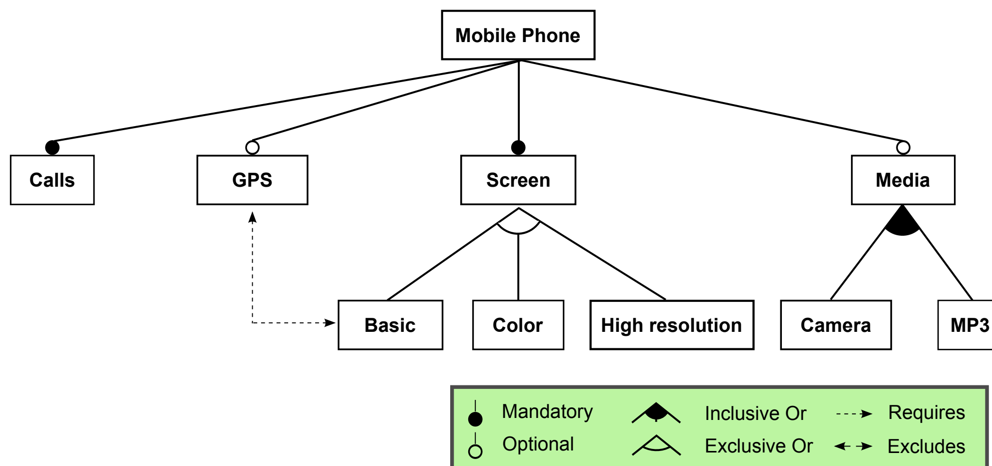


Figure 2.1: A sample feature model

The feature model is a logical representation of feature constraints in a product line [5]. As seen in Figure 2.1, the feature model of the product line `Mobile Phone` supports the mandatory features `Calls` and `Screen`. These two features are shared between all products of the product line `Mobile Phone`. Optional features, such as `Media`, are selectable features. Features can also be combined in a group under an exclusive-or relation such as `Basic`, `Color` and `High resolution`. This implies, that only one of these features can be chosen for generating a valid product.

Besides mandatory and optional features, during the feature modeling process as part of software product line engineering, a couple of different abstractions are commonly used, such as AND/OR/Implication/Mutual Exclusion etc. In our example Figure 2.1 the features `Basic, Screen` and `GPS` exclude each other. We can only choose one of these features to generate a mobile phone according to this specification.

These constraints further limit the amount of possible valid feature combinations which are used to make a product. This sets an upper limit to the number of possible generatable products for your software product line. A valid

feature selection for our example feature model would be: (`Calls, GPS, Basic, Screen, MP3, Media`). An invalid feature selection would be: (`Basic, Screen, Color, Screen`).

Today's software product lines can have several hundreds of features. This leads to an exponential number of possible products, which can be generated from said product line and makes sequential analysis and testing of products infeasible. The LINUX kernel for example has over 11,000 configurable features [8] and it is not even possible to calculate the number of valid products for this product line.

A very commonly used tool for developing software product lines in the C programming language is the preprocessing tool CPP [6].

2.2 The C Preprocessor

The C preprocessor CPP is a token-based text processing tool, which was developed over 40 years ago, to extend the capabilities of the C programming language. With CPP it is possible to introduce variability on the source-code level [9] and because it is token-based it can be used for any programming language [10, 11]. But CPP annotations are oblivious the underlying structure of the programming language and can be introduced on any level in the source code [9].

CPP provides low-level textual transformations on the source code which then can be compiled into a program [12]. To do so, CPP provides three basic directives.

2.2.1 File Inclusion

```
1 #include <stdio.h>
2 #include "userdefined.h"
3
4 main() {
5   printf("Hello World\n");
6 }
```

Figure 2.2: `#include` example

With the file inclusion (`#include`) directive to include the content of another file. CPP will scan the specified file as input and then continue with the rest of the current file. The output will now consist of the content of the included file followed by the output from preprocessing the rest of the current file after the `#include` directive. This also works recursively. If one of the included files has

another #include directive for a second file it will also process that file and its other directives will also be resolved.
#include <*file*> is used to include system header files. These are header files which declare interfaces to parts of the operating system. #include "*file*" is used for header files containing interfaces of source files from the analyzed own program. In the first case the CPP will look for the file in the list of system directories while in the second case it will check the directory containing the current file.

2.2.2 Text Substitution

```
1 #define SUM(a,b) (a + b)
2 #define BUFFER_SIZE 1024
3
4 main() {
5   int doubleBuffer = SUM(BUFFER_SIZE, BUFFER_SIZE);
6 }
```

Figure 2.3: #define example

With the help of the#define directive, CPP enables developers to specify textual substitution. #define directives can be used in different ways. The first define directive on Line 1 is a function-like directive. This is an identifier followed by a declaration of parameter names separated by commas within parentheses and the last part is the replacement token. When the CPP identifies a function-like #define directive invocation it substitutes the parameters in the replacement code by the corresponding argument.
The second #define directive on Line 2 is an object-like directive. It consists of a single identifier followed by tokens. If this #define is used in the source code, CPP will substitute the identifier for the replacement tokens.

2.2.3 Conditional Compilation

A conditional directive tells CPP whether or not to include the tokens in its annotation for the compilation. These conditional directives (#if, #ifdef, #ifndef, #elif, #else and #endif) will be referred to as #ifdefs. An #if directive can test arithmetic expressions while #ifdef directives check if a given identifier is defined. Together with #elif and #else these directories kind of resemble if statements, which exist in most programming languages including C. The important difference however is that these macro conditions will be tested at compilation time while normal if statements will be evaluated at runtime. The

```
 1 typedef struct T_node {
 2   int item;
 3   struct T_node *next;
 4 #if LINKED
 5   struct T_node *prev;
 6 #endif
 7 } node;
 8
 9 void insert(node *elem) {
10 #if SORTALGO == BUBBLESORT || SORTALGO == INSERTIONSORT
11   if (first->item
12 #if SORTORDER == 0
13       >
14 #else
15       <
16 #endif
17       elem->item) {
18   // sort ascending or descending with
19   //  bubblesort or insertionsort
20 #elif SORTALGO == MERGESORT
21   // sort with mergesort
22 #else
23   // sort with a default sort algorithm
24 #endif
25 }
```

Figure 2.4: Extract of a variable list implementation using CPP

conditions of `#ifdef`s are refered to as presence conditions. The tokens inside these `#ifdef`s are only compiled if the presence condition is evaluated as `true`.

In Figure 2.4 on Line 4 we see an `#if` directive with the presence condition `LINKED`, whereas on Line 10 the presence condition is `SORTALGO == BUBBLESORT || SORTALGO == INSERTIONSORT`. The last presence condition consists of two features, `BUBBLESORT` and `INSERTIONSORT`, which are then compared to the definition of `SORTALGO`. Presence conditions are either single feature constants or a boolean combination of one or more feature constants.

`#ifdef`s are used in different ways:
A simple `#if` or `#ifdef` directive is an optional directive. CPP will only parse the tokens inside the block for compilation if its presence condition is evaluated to `true`.
A `#if`(-`#elif`)-`#else` block offers alternative compilation. Exactly one of these directives has to be compiled.

As we can see in Figure 2.4 on Lines 4-6, conditional compilation can be used to offer optional inclusion of tokens. The `struct T_node` only has a `*prev` node if the feature `LINKED` is selected.

The `#if` directives on Line 10-23 offer alternative code inclusions. If the sorting algorithm is defined as "`MERGESORT`" the code on Line 11-18 will be included for compilation, otherwise it will try to match the condition of the next `#elif` statement. If neither the `#if`, nor any `elif` condition evaluates to `true`, the CPP will process the code inside the `#else` statement on Line 22.

Together with CPP and the code from 2.4, we now have the possibility to use different sort algorithms in our variable list implementation from Figure 2.4. Furthermore, we have the option to make nodes have references not only to the next node but also to its predecessor and to sort either ascending or descending.

2.2.4 Problems of CPP Usage

```
1  #define TEST1  100 // define a variable.
2  #define TEST2
3  /* process input based on the what was defined above */
4  #ifdef TEST1
5  TEST1 (test1) is defined
6  #ifdef TEST2   // another check (nested within the first)
7  The Value TEST2 (test2) is defined
8  #else          // else for #ifdef TEST2
9  TEST1 (test1) is defined and TEST2 is not defined
10 #endif         // #endif for #ifdef TEST2
11 #elseif TEST2
12 TEST1 is NOT Defined,  and TEST2 (test2) is defined
13 #else
14 TEST1 and TEST2 are not defined
15 #endif
```

Figure 2.5: Nested `#ifdefs`

Preprocessor usage itself can lead to many problems in the software development or maintenance process. `#ifdefs` can be nested and this makes it very difficult to understand a certain code fragment. When developers introduce new nested `#ifdef` directives as seen in Figure 2.5 on Line 6, these new directives are usually not tested in all possible configurations as the amount of configurations scales exponentially with the depth of the `#ifdef` nesting. This leads to a huge number of possible code paths [13].

Moreover CPP is a token-based preprocessing tool. Hence, these tokens are not restricted in any way. They are oblivious to the structure of the underlying programming language [9]. In the example of Figure 2.4 on Line 4-6 there is an annotation on the declaration level, while the annotations on Line 12-16 are on sub-expression level. These `#ifdefs` can introduce variability on any level in the program and this can make it extremely difficult to understand and analyze preprocessor code [14, 15, 16, 12].

2.3 Current Analysis Methods for C Product Lines

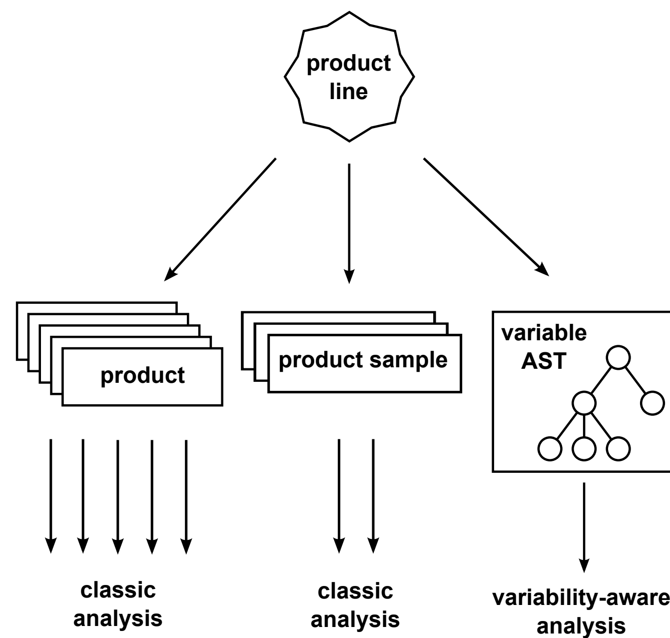


Figure 2.6: Different approaches for product line analysis

Currently there are different techniques for analyzing software product Lines written in the C programming language. We will take a brief look at three different approaches as illustrated in Figure 2.6:

- *Product-Based Analysis* first generates all possible products according to the underlying feature model. Then all these products are tested by using existing analysis methods that are not variability aware.
  This approach can be satisfying for very small product lines which have a small number of derivable products. The only advantage of product-based analysis is that it can utilize the existing analysis tools. However, for most product lines this analysis method does not scale. Even just generating all possible products can be infeasible because the number of products is up-to exponential to the number of features in the product Line. Another

disadvantage is that this approach does not exploit the similarities between the different products and thus performs many redundant computations [1].

- *Sample-Based Analysis* is defined by running product-based analysis on a subset of possible products.
  The most important factor for this analysis is how this subset is generated. There are many different sampling heuristics such as code coverage or pair-wise coverage [1]. Compared to product-based analysis this attempt runs faster because only very few out of all possible products are analyzed. This however leaves many products untested and even just running the sampling heuristics can be infeasible for bigger product lines such as the Linux kernel. This approach is basically product-based analysis with a trade-off: The number of tested products is smaller, which makes analyzing faster but only a fraction of all possible products are tested and this makes it harder to evaluate the results and draw a conclusion about the whole product line.

- *Family-Based Analysis* lift analysis techniques to make them variability aware. This involves making data structures variability aware and rewriting the analysis algorithms.
  This method exploits similarities of different products to speed up the analysis process by eliminating redundant computations. It is not required to generate and analyze every single product because the analysis runs on code artifacts with variability. This drastically reduces the analysis effort and scales better compared to product-based analysis while still analyzing the properties of all variants [17]. However, this approach requires redeveloping of existing analysis tools. Some of these tools were developed decades ago or over the course of many years. The redevelopment of these analysis tools requires immense development efforts [17].

2.4 TypeChef

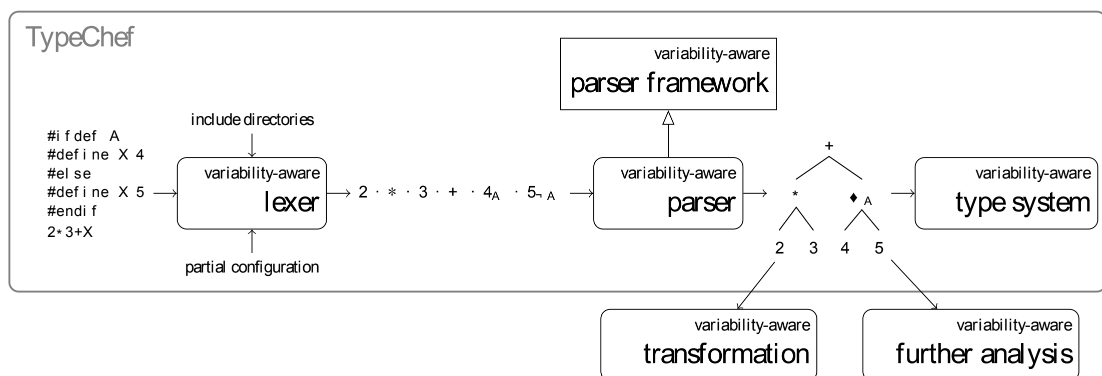


Figure 2.7: The TypeChef Framework

TypeChef is a tool developed for parsing and type-checking of preprocessor-based product lines written in C. TypeChef uses the family-based analysis approach. In order to implement these variability-aware type checks the framework has to process the source code in different steps as we can see in Figure 2.7 [18].

The first step is lexing the original source code including preprocessor directives. During this step the lexer resolves text substitution and file inclusion `CPP` directives, but leaves conditional directives untouched. As Figure 2.7 shows the input program is divided into different tokens. These tokens can have annotations, which can be seen for the definition of the parameter `X`. `X` can have two possible values: `4` and `5`. Hence, the lexer has to make annotations for the usage of the value `X` in the expression `2*3+X`: 4_{A} and $5_{\neg A}$.

The second step is parsing the token stream from the lexer. This variability-aware parsing framework enforces disciplined annotations during the parsing process. Disciplined annotations are declarations, definitions and directives that include a statement inside a function or fields inside a `union` or `struct` [18]. TypeChef is enforcing these disciplined annotations, because it has to convert variability on the token level into variability on a data structure, the variability-aware AST. This AST contains all variability information from `#ifdef` directives of the original source code.

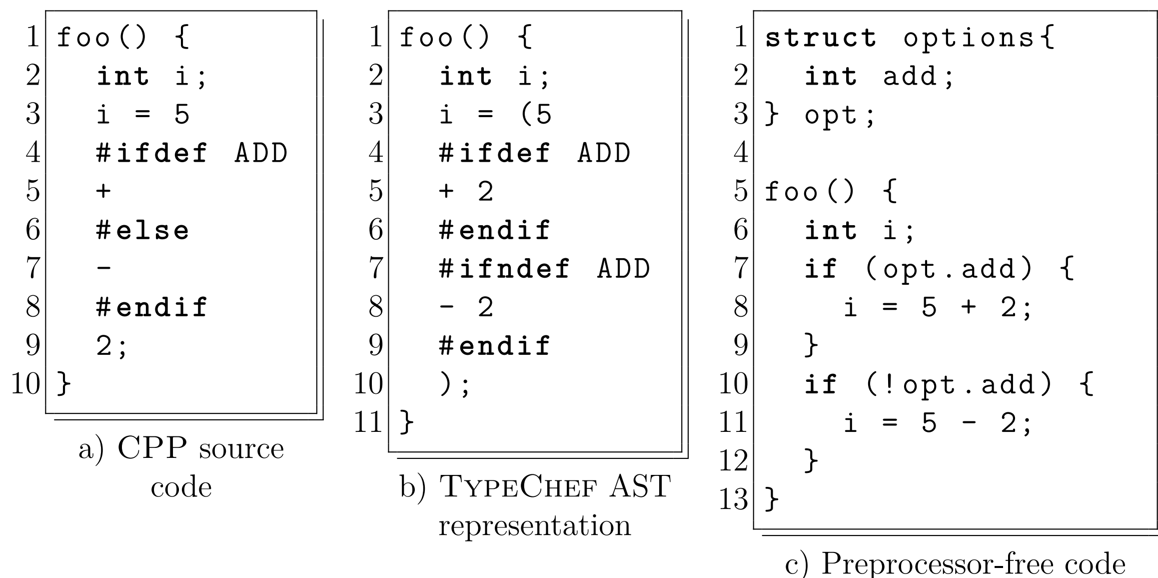


Figure 2.8: Enforcing disciplined annotations

Summarized, TypeChef is a variability-aware parsing tool for GNU C which can parse any C code with preprocessor directives.

In order to be able to create an AST structure, TypeChef has to be able to map elements from the source code to AST elements. In Figure 2.8 we can see

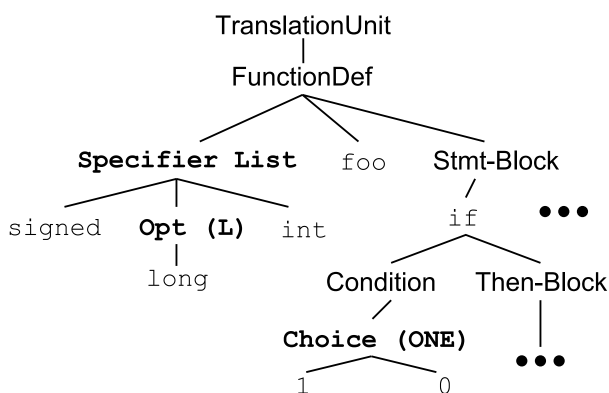
three different code listings, which represent three different stages of variability. The Figure 2.8 (a) shows CPP source code with one variable annotation on sub-expression level on Line 4-8, which varies the operation in this expression.

TypeChef has to transfer the variability on a token-based abstraction level to variability on data structures in the AST. Now because there is no AST representation for a list of optional operators TypeChef has to expand this annotation as we can see in Figure 2.8 (b) on Line 4-9. In this concrete example, we can see an annotated operator `+` and a parameter `2`. These two elements together can be expressed in the AST as an optional expression element, whereas the operator alone can not be resembled as an expression element. We now created a fully annotated AST representation.

For our configuration lifting approach, we have to transform variability from fully annotated sub-trees in the AST which leads to even more code expansions. Conditional variability in the C programming language can only be expressed with `if` statements. This restriction forces us to make code expansions on the level of statements or declarations rather than sub-expressions for example. Code expansions at the statement level make it possible to move these statements into the `then` body of an `if` statement. This can be seen in Figure 2.8 (c) on Line 7-11. We need to copy the whole assignment of `i` in order to create valid preprocessor-free code.

```
1  signed
2  #ifdef L
3    long
4  #endif
5      int foo() {
6        if (
7      #ifdef ONE
8          1
9      #else
10         0
11     #endif
12       ) {
13           // Code
14       }
15 }
```

a) Variable function definition



b) (Simplified) AST representation

Figure 2.9: Choice nodes represented in source code and AST form

To represent variability in an AST, TypeChef uses two data structures to capture `#ifdef` variability: `Choice` and `Opt` nodes. In Figure 2.9 we can see an example for a case where both of these data structures occur. The code on the left side shows a function definition with variability in its specifiers and an `if` statement with two alternative conditions.

`Choice` nodes are a tree-like data structure, which consists of three elements, a presence condition and two child nodes. If the presence condition of a `Choice` node is fulfilled the program path will evaluate the first inner child node, otherwise it will evaluate the second one. `Choice`s offer a representation of alternatives [19]. They can be visualized in a tree-like structure, in which each `Choice` node has two child nodes. The left child node indicates, that the presence condition of its parent was evaluated to `true`, similar the right node indicates that the condition was not fulfilled. In Figure 2.9 (a) on Line 7-11 and in the AST representation (b), we can see an example for a `Choice` node. Depending on the presence of the definition of `ONE`, CPP generates an `if` statement with a varying condition (`0` or `1`).

Besides `Choice` nodes, TypeChef also generates `Opt` nodes, which consist of a presence condition and an AST element. `Opt` nodes are always a part of a list and can occur on levels, where the standard C grammar allows variability in the form of lists, such as statement and parameter lists. The semantic of an `Opt` node is, that its AST element is only included in the program if the node's presence condition evaluates to `true`. The function specifier in Figure 2.9 (a) on Line 3 is only evaluated, if the condition `L` is defined. This function specifier can also be seen in the AST representation of Figure 2.9 (b), where we can see an `Opt` node for the `long` specifier with presence condition `L`.
Because `Opt` nodes can occur on so many different levels, dealing with them involves dealing with many different patterns. The TypeChef parser also converts `#elif` and `#else` directives from `Opt` nodes into independent `#if` directives as seen in the example of Figure 2.8 (b) on Line 7.

The issue is that most available tools are not able to deal with the variability which is introduced through preprocessor use. Moreover the generation and analysis of each possible derivable product is not feasible because of the sheer amount of possible different products. If we assume that generating and analyzing a single product would take one second then a product line with just 25 distinct and independent features will take more than a year of sequential analysis to be tested in all variants. This is why we present our configuration lifting approach as a possible solution

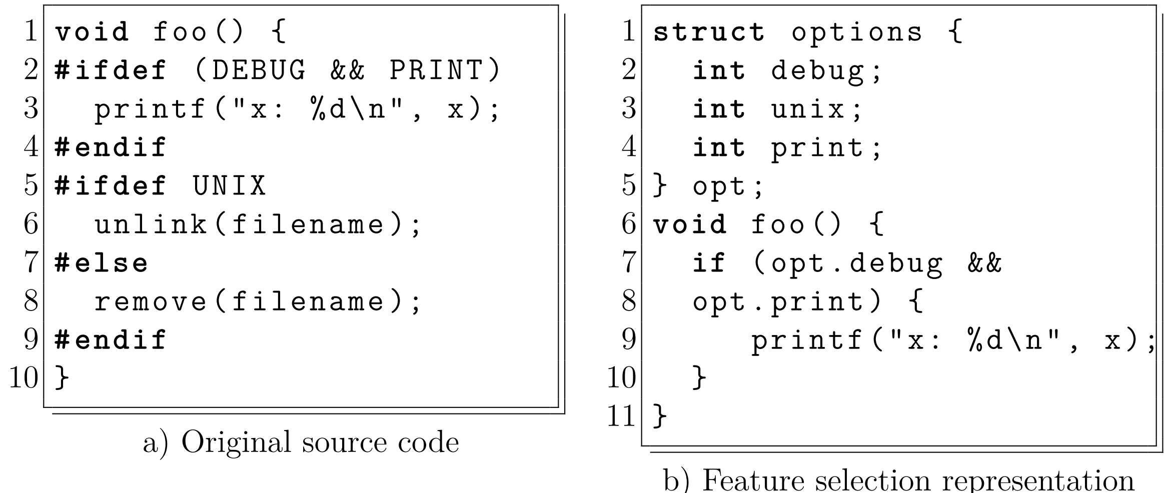
3 Approach

In this section we will cover the basic idea and different steps of configuration lifting while also looking at the challenges and possible solutions we encountered.

3.1 `#ifdef` to `if` Transformation

3.1.1 General Idea

The general idea is to replace the preprocessor code in a product line with preprocessor-free code constructs of the C programming language without changing the semantic of the product line. These replacements basically turn the static variability of preprocessor constructs into runtime variability and thus allow us to use verification tools such as model checking or static analysis with this newly created meta-product.



a) Original source code

b) Feature selection representation

Figure 3.1: Creating a feature selection representation

The first step is to create a C conform and preprocessor-free representation of presence conditions from `#ifdef`s of the original source code. As Figure 3.1 (b) shows, we created a `struct options`. The next step is to map the names of all

possible configurable features to `boolean` values in order to distinguish the feature selection status, this can be seen on Line 2-4. As the C programming language does not have a `boolean` data type we have to use the `int` data type instead. These fields can be assigned and evaluated at runtime, unlike the preprocessor flags which are evaluated at compilation time. The original presence conditions, such as `(DEBUG && PRINT)` can then be transformed into conditions for normal `if` statements as Line 6-7 of Figure 3.1 (b) show.

3.1.2 Embedding Statements into `if` Statements

We will now look at three different transformation techniques we utilize for configuration lifting. The first approach when dealing with preprocessor directives, is to replace them by normal conditional `if` statements. This ensures that an annotated statement is only executed if its presence condition is fulfilled. However we cannot simple replace all `#ifdef`s by normal `if` statements because first of all `#ifdef`s can occur on any level in the code. Even TypeChefs expansion of annotations does not remove all possible variability on a sub-statement level. Another reason is that `if` statements create a new block with its own visibility, which is not convenient for transforming declarations as these declarations would not be visible outside the `if` block and thus we would preserve the observable behavior of the code.

```
1  resetCache() {
2    cache = 0;
3  }
4  // ...
5  main() {
6    // ...
7  #ifdef RESET_CACHE
8    resetCache();
9  #endif
10 }
```

a) Original source code

```
1  struct options {
2    int reset_cache;
3  } opt;
4  resetCache() {
5    cache = 0;
6  }
7  // ...
8  main() {
9    // ...
10   if (opt.reset_cache) {
11     resetCache();
12   }
13 }
```

b) Transformed code

Figure 3.2: Embedding variable statements into `if` statements

So when looking at a basic example of an optional statement, such as in Figure 3.2 on Line 7-9, we just transform the presence condition into a C conform

condition for a new `if` statement and move the statement into the `then` body. The embedding technique can also be used in combination with the other techniques as we will see in later sections.

The example in Figure 3.2 (a) shows an annotated function call `resetCache();` on Line 8. We take the presence condition `RESET_CACHE` and transform it into a condition for our new `if` statement as seen in Line 7 of Figure 3.2 (b). Afterwards we use the original statement and embed it into the `then` body of our `if` statement as seen in Line 8. This way the statement `resetCache();` is only executed if the condition `opt.reset_cache` is fulfilled.

3.1.3 Renaming Declarations

In this section, we discuss the concept of renaming identifiers of variables and functions. Alternative `#ifdef`-(`#elif`)-`#else` blocks can use the same names for different declarations, because only one of the possible declarations will be evaluated. These constructs are often used to describe alternative data types for variables. In order to avoid name conflicts we have to rename the identifiers of annotated declarations, as well as declarations with optional sub-parts. A consequence of these renamings is that we have to replace references to the original declaration by conditional references as we will see later in Section 3.1.5.

```
1 #ifdef X64
2   long value = 0;
3 #else
4   double value = 0.00;
5 #endif
```

a) Original source code

```
1 struct options {
2   int X64;
3 } opt;
4
5 long _X64_value = 0;
6 double _NX64_value = 0.00;
```

b) Transformed code

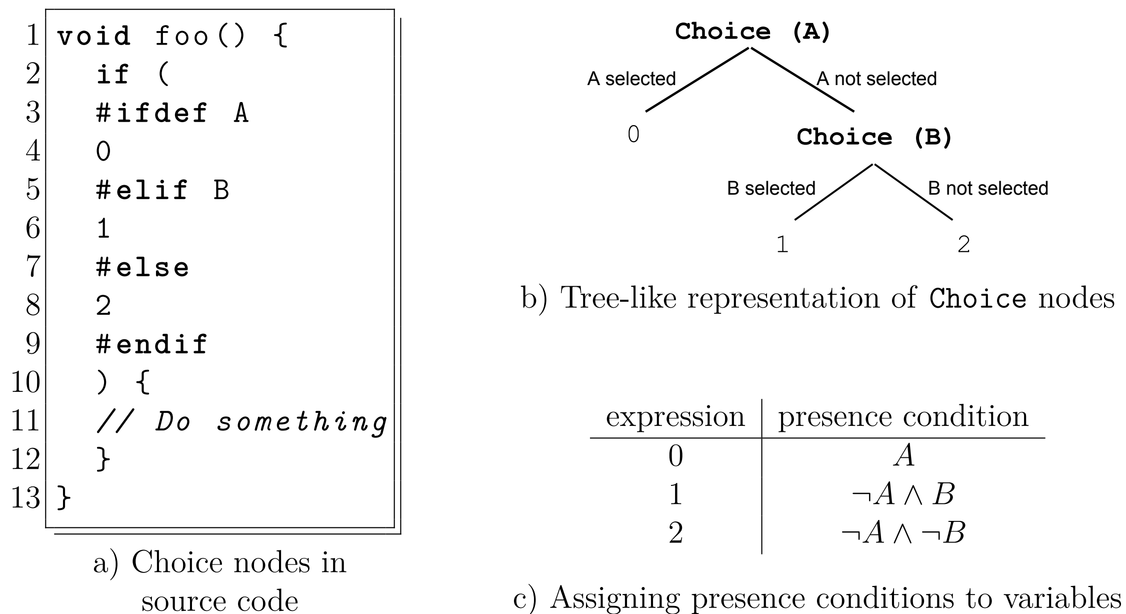
Figure 3.3: Renaming declarations and function definitions

Renamings are executed by adding a prefix in front of the original identifier. This prefix has to contain the presence condition information under which the original declaration occurred. If analysis tools for example report that one of the renamed variables does not meet its specification, this makes it possible to find out what feature combinations caused analysis errors by looking at its prefix. We apply renamings by mapping presence conditions to numbers and adding a prefix of the form `_x_`, where `x` is a number to the original identifier. For our transformation examples however we will just put a text representation of the presence condition in between the underscores to make it easier to see the condition under which a variable used to be declared.

In Figure 3.3 (a) on Line 1-6, we can see two alternative declarations for the parameter `value`. The first declaration on Line 1-3 is represented as an optional declaration with presence condition `X64`, while the second declaration is optional with the presence condition ¬`X64`. When looking at the different AST elements we detect optional elements such as declarations of variables, `struct`s, functions, etc., remove the presence condition and add a prefix to the identifier.
In order to generate valid code these renamings also have to be done every time one of these optional declarations is referenced. However, declarations can also be represented differently as Figure 3.5 (a) shows in Line 1-6. The declaration itself is not annotated, but it has variable type specifiers. So when looking at declarations, we also have to check the child nodes in the AST, in our case `Specifiers` and `Parameters`, for variability. In our example in Figure 3.5 (a) on Line 1-5, we spot two optional specifiers (`long` and `int`). Consequently we have to replace the original declaration with two new declarations.

3.1.4 Code Duplications

In order to deal with variability of `Choice` nodes or variability which involves several different presence conditions in `Opt` nodes we have to resort to code duplications. This technique has to be used together with renamings and/or statement embedding as simply duplicating statements or declarations and definitions creates either uncompilable code or a different program behavior.



expression	presence condition
0	A
1	$\neg A \wedge B$
2	$\neg A \wedge \neg B$

a) Choice nodes in source code

b) Tree-like representation of `Choice` nodes

c) Assigning presence conditions to variables

Figure 3.4: Evaluating `Choice` nodes

For example when handling declarations which possess multiple different annotated specifiers, we have to duplicate this declaration for every valid feature combination of these specifiers and afterwards rename the identifier to avoid name conflicts. Dealing with `Choice` nodes is similar, we take a look at the leaf nodes of the `Choice` tree as seen in Figure 3.4 and create a variant of the condition for each of these possible leaf nodes.

As we mentioned earlier in Section 2.4, `Choice` nodes consist of three subelements, a presence condition and two possible child nodes. Based on the value of the presence condition, either the first or the second child node is evaluated. We visualized the choice node of Figure 3.4 (a) on Line 3-9 in a tree-like representation in Figure 3.4 (b). If the first presence condition `A` is fulfilled, we will evaluate the left child node of `Choice (A)`. We do this for every possible path in our tree and generate the current presence condition of that path until we reach a leaf node.

The table of Figure 3.4 (c) shows the different variables and the presence conditions under which they are present. The `Choice` node of Figure 3.4 (a) on Line 3-9 has three different possible variants, depending on the feature selection. Consequently we have to duplicate the statement of this `Choice` node three times, once for each possible variant. We filter out `Opt` nodes, which do not satisfy the feature of the current variant and embed the remaining AST nodes into an `if` statement.

3.1.5 Combining Transformations

In order to make semantically correct transformations we often have to apply a combination of the previously introduced transformation techniques to AST elements at the same time. We have to apply our transformations on complete declarations, function definitions or statements because we can only duplicate or embed these AST elements. This means we have to check the inner elements of the previously mentioned AST elements in order to decide how to transform them. A function definition for example can be optional. In this case it is handled similar to declarations, as shown in Figure 3.3 (b). But a function could also return a different data type or have variable function parameters. In order to make valid transformations we have to distinguish between all these possible situations.

Figure 3.5 shows an example of a combined transformation. This code extract shows usages of alternative and optional declarations, as well as their usages in a function. In Line 6 of Figure 3.5 (a), we can see a declaration of `val` with different optional specifiers. We identify the different presence conditions and create a new declaration with a new identifier for each condition. This also implies that we have to rename further usages of `val` according to the presence condition of the statement where `val` is referenced. If a reference to `val` is used in an annotation-free context as seen in Figure 3.7 (a) on Line 9, it is necessary to du-

```
 1 #ifdef X64
 2  long
 3 #else
 4  int
 5 #endif
 6 val = 0;
 7 #ifdef X64
 8  long sec_val = 0;
 9 #endif
10
11 void saveValue(
12 #ifndef X64
13  int sec_val
14 #endif
15 ) {
16   val = val
17   #ifdef ADD
18    +
19   #else
20    -
21   #endif
22   sec_val;
23 }
```

a) Original source code

```
 1 long _X64_val = 0;
 2 int _NX64_val = 0;
 3 long _X64_sec_val = 0;
 4
 5 void _X64_saveValue() {
 6   if (opt.add) {
 7     _X64_val = _X64_val +
 8       _X64_sec_val;
 9   }
10   if (!opt.add) {
11     _X64_val = _X64_val -
12       _X64_sec_val;
13   }
14 }
15 void _NX64_saveValue(
16         int sec_val) {
17   if (opt.add) {
18     _NX64_val = _NX64_val
19       + sec_val;
20   }
21   if (!opt.add) {
22     _NX64_val = _NX64_val
23       - sec_val;
24   }
25 }
```

b) Transformed code

Figure 3.5: An example of transformation combinations

plicate that statement, embed it into an `if` statement and rename `val` according to its possible different declarations.

The function definition starting on Line 11 of Figure 3.5 (a) has to be cloned as well, because it has two possible different variants depending on the feature selection status of `X64`. The first variant, where `X64` is selected, has no function parameters, while the second variant has an annotated function parameter `int sec_val`.

3.2 Problems

3.2.1 Renaming Declarations and Function Definitions

One of the challenges is to make sure to rename all usages of declarations or function definitions. Once the original declaration is renamed we have to make sure that all of its usages are also renamed accordingly. Otherwise there would still be usages with the identifier of the original declaration left and this creates erroneous code, because these usages do not have a declaration any more. Variability-aware renamings in the presence of preprocessor directives are a quite complex task [20].

Renamings get even more complex, when dealing with variable declaration-use cases where a parameter has several possible annotated declarations but also unannotated usages. This occurs seen a lot when dealing with conditional types for different system architectures such as X86 and X64, for example in the LINUX kernel. To maximize the performance of a product running on a system with limited resources it is often important to choose different data types as Figure 3.5 (a) shows in Lines 1-5.

When dealing with variable declaration-use cases we have to look at the identifiers of each statement and decide how to transform the statement. If the current presence condition of the statement implies one of the conditions of the possible declarations of our parameter we can simply rename the parameter. If this is not the case we have to create a new statement for every possible declaration of that parameter, embed it into a new `if` statement and then apply the proper renaming.

Duplicated function definitions which are caused by optional specifiers or optional function parameters have to be handled in a similar way. Every time there is a function call to the original function definition, the statement where the function call took place has to be embedded into an `if` statement to make sure that the right variant of the function is executed according to the current circumstances.

3.2.2 Keeping Code Duplications Minimal

Keeping code duplications minimal is very important to avoid slowing down analysis and verification tools. Of course it is possible to apply the brute-force attempt and just create a variant of every source code file for every possible feature combination. But this analysis approach is infeasible and usually does not scale. Hence, it is very important to make these transformations on the finest level of granularity as possible. We also have to take into account that variability can occur on multiple distinct AST levels.

A function definition for example can have optional specifiers *specs*, as well as optional parameters *params*. In this case it is necessary to look at *specs* and *params* individually. Then calculate the possible presence conditions for each level and store them in two separate lists. In the next step, we then remove multiple occurrences of presence conditions across these two lists to make these lists are disjunct to each other. The last step is to calculate a list of presence conditions for the whole function definition. This is done by calculating the Cartesian product of these two lists. The elements of the result lists are then combined by boolean conjunction.

```
1  #ifdef STATIC
2    static
3  #endif
4    foo(int value
5  #if (OFF&&SET)
6      , int offset
7  #endif
8      ) {
9        return (value + offset);
10 }
```

Figure 3.6: Variability on different levels

In the example code in Figure 3.6, the possible presence conditions for the specifiers of the function `foo` are $STATIC$ and $\neg STATIC$ while the conditions of the function parameters are $OFF \wedge SET$ and $\neg OFF \vee \neg SET$. The next task is to determine the list of presence conditions for the different possible function definitions. This is done by combining both lists as we described earlier: $\{STATIC, \neg STATIC\} \wedge \{OFF \wedge SET, \neg(OFF \wedge SET)\} = \{STATIC \wedge (OFF \wedge SET), STATIC \wedge \neg(OFF \wedge SET), \neg STATIC \wedge (OFF \wedge SET), \neg STATIC \wedge \neg(OFF \wedge SET)\}$. This means due to conditional preprocessor directives we have to create four different variants of the function `foo`.

3.2.3 `Switch-Case` Blocks

Another challenge are `switch-case` blocks with `#ifdef` directives. These annotations can occur on many different possible levels and influence the control flow. `#ifdef`s around `break`, `default` and `return` statements can lead to many possible different exit points depending on the feature selection.

Generally a `switch` statement consists of an expression *ex*, a block of `case` statements and a `default` statement. *ex* is then compared for equality with the

condition of the first `case` statement of the `switch` block. If they match then the code inside the `case` statement is executed otherwise the program will jump to the next `case` statement and test for equality of expressions again. If one of the expressions of a `case` statement matches the `switch` statement expression and there is a `break` statement inside the `case` block, the program continues its execution with the statement after the `switch` statement. However, this is not the case if there is no `break` statement, then control is transferred to the next `case` statement. This continues until a `break` statement is executed.

If a switch case structure contains variability, it is very difficult to make code transformations on the level of `case` statements. There are many aspects which have to be taken into consideration because the preprocessor usage can occur at every possible part of the `switch` block. The program flow can get very complex (e.g. with annotated `break` or `case` statements). In order to make transformations on the `case` statements while preserving the behavior of the original `switch` statement, it is necessary to insert `label` statements as jump targets and `goto` statements at the right places. This task is very difficult in combination with other transformation techniques.

3.3 Solutions

3.3.1 Declaration Use Map

To enable consistent identifier renamings during the transformation, we have implemented a declaration-use map in TypeChef. In order to realize variability-aware type checking the TypeChef framework has a type system module. To make valid renamings we implemented a hook into the type system, which enables us to store information from the type-checking process, which otherwise would not be preserved in the TypeChef framework.

First, while the type checking process looks at declarations and function definitions, we put the identifiers of these elements as keys into a new data structure `Map(Id -> List(Id))`, the declaration-use map. This makes it possible to get usages of variables of all scopes, as we check for equality of the identifier element instead of just looking at the name of a parameter, which could be declared multiple times in different scopes or under different presence conditions as Figure 3.3 shows on Line 2 and 4. Second, when looking at expressions where the previously mentioned identifiers of declarations are used, we add identifiers of usages into the value field of our declaration-use map.

This implementation stores information of all identifiers from the variable AST. This includes: global `variables`, function parameters, `struct/enum/union` identifiers, identifiers of the fields inside these `structs/enums/unions`, as well as

their usages. We also store the identifiers of function declarations and map them to their corresponding function calls.

The important aspect for our `#ifdef` to `if` transformation is that when renaming a declaration or function definition, we have to keep track of the presence condition of this declaration. This is done by getting the identifier list of that declaration from the declaration-use map and then map each usage to a list of possible presence conditions. When looking at statements we can now decide if an identifier inside a statement was declared under different presence conditions.

```
1  void foo() {
2    #ifdef PRECISE
3      float value;
4      value = 1.33;
5    #else
6      int value;
7      value = 1;
8    #endif
9    value += 30;
10   #ifdef PRECISE
11     value = 1.33;
12   #endif
13 }
```

a) Original source code

Name	Declaration -> List of usages
foo	L. 1 -> List()
value	L. 3 (`float`) -> List(L. 4, L. 9, L. 11)
value	L. 6 (`int`) -> List(L. 7, L. 9)

b) Declaration-Use Map

Name	Usage	(Declarations, Presence Condition)
value	Line 4	(Line 3, `PRECISE`)
value	Line 7	(Line 6, ¬`PRECISE`)
value	Line 9	(Line 3, `PRECISE`), (Line 6, ¬`PRECISE`)
value	Line 11	(Line 3, `PRECISE`)

c) Mapping usages to presence conditions

Figure 3.7: Variability-aware declaration-use mapping

As we can see in Figure 3.7 (a) on Lines 3 and 6 we have different declarations of the parameter `value` (`float` and `int`). Our declaration-use map puts both of their identifiers as keys into our map, as they are different declarations which occur in different `#ifdef` blocks. The first use case in Line 4 is associated to the declaration of Line 3 because of its presence condition. Similar the use of `value` on Line 7 belongs to the declaration on Line 6.
The next usage on Line 9 however, belongs to both possible declarations, as it is independent of any feature selection or presence conditions. The consequence is that we have to add this identifier to both declaration entries of our map. Table 3.7 (b) shows, how the declaration-use map looks like for this simple code example.

In order to create semantically equal code with our transformations we have to keep track of the possible presence conditions under which an identifier can be used. This is shown in Table 3.7 (c). We identified the use of `value` on Line 9 from the original code as an annotation-free usage. We also know that we have

two different possible declarations for this parameter. In this case it is necessary to keep track of the different presence conditions under which `value` is declared to be able to duplicate the statement in Line 9 and embed it into an `if` statement.

3.3.2 Reconstruction of `#if`, `#elif` and `#else` Conditions

In order to keep code duplications minimal, it is important to reconstruct the `#if`, `#elif` and `#else` structure of `Opt` nodes. Because the TypeChef parser generates independent presence conditions for each `#elif` and `#else` directive, we lose the information about the original structure and relations between these presence conditions. An `#ifdef` directive with two possible different evaluations (a simple `#if` followed by an `#else` block) and presence conditions, which depend on n different features, should not be expanded to 2^n different variants.

To reconstruct the structure of the original code, we process the complete AST in one pass. Each `Opt` element e of the AST has a presence condition (which is *true* for annotation-free `Opt` nodes) *pc*. To process the node e, we take a look at the next level of `Opt` nodes inside of e and collect their presence conditions in a list l. We then compare the elements of l pairwise and check if two consecutive presence conditions mutually exclude each other. If they do, we put these presence conditions in a new result list r. We continue this process until the original list is empty or the result of a mutual exclusion is `false`. Then we look at the generated list r and check if all if its presence conditions combined with conjunction imply the context *pc*. If that is the case we found presence conditions which used to belong to an `#if #else` structure.

In the example in Figure 3.8 (a), the context under which the variable `value` exists is Z. The number of different feature constants in the example is four (A, B, C and D). The table in Figure 3.8 (b) shows how the presence conditions of each specifier. To calculate the presence conditions for our code duplications we check if each pair of consecutive presence conditions mutually excludes each other. The truth table of Figure 3.8 (c) shows that the presence conditions #1 is mutually exclusive to #2 and #2 is mutually exclusive to #3. The last step is to check if their combination implies the context Z: $((\#1 \wedge \#2 \wedge \#3) \Rightarrow Z) \Leftrightarrow true$.

Therefore, we know that the presence conditions of the specifiers were annotated in an `#if-#else` structure and we generate three different variants of the parameter `value`. If we look at the presence conditions, we can see that they consist of four different features (`Z`, `A`, `B`, `C`). So instead of generating $2^4 = 16$ variants we only generated three variants. This leads to a reduced number of code additions and reduces the load on subsequent analysis algorithms.

```
 1 #ifdef Z
 2 #ifdef A
 3   float
 4 #elif (B && C)
 5   short
 6 #else
 7   double
 8 #endif
 9   value = 0;
10 #endif
```

a) A conditional construct

#	presence condition	specifier
1	$Z \wedge A$	float
2	$Z \wedge \neg A \wedge (B \wedge C)$	short
3	$Z \wedge \neg A \wedge \neg(B \wedge C) \wedge F$	double

b) Presence conditions of the specifiers

Z	A	B	C	#1	#2	#3	$\#1 \wedge \#2 \wedge \#3$
1	0	0	0	0	0	1	1
1	0	0	1	0	0	1	1
1	0	1	0	0	0	1	1
1	0	1	1	0	1	0	1
1	1	0	0	1	0	0	1
1	1	0	1	1	0	0	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	1

c) Truth table

Figure 3.8: Reconstructing the `#if`-`#elif`-`#else` structure

3.3.3 Brute-Force Code Duplications

In Section 3.2.3 we have shown that making transformations on a fine level of granularity is very complex for `switch-case` blocks. In order to avoid possible erroneous transformations when dealing with variability in `case` statements, we apply brute-force duplications on the whole `switch-case` block.

We look at all the features which are used in the `#ifdef` directives inside the `switch-case` block *sc* and generate all possible feature combinations for *sc*. Then we test if each feature combination is valid according to the feature model. This can narrow down the number of valid feature combinations and generated code fragments drastically. For each valid combination, we apply regular `#ifdef` to `if` transformation on the statement *sc*.

4 Evaluation

4.1 BusyBox Case Study

To test our approach of configuration lifting, we selected the BusyBox tool suite as our case study. BusyBox is a medium-sized product line, which implements standard Unix utilities. The focus of BusyBox are systems with limited resources, such as embedded systems and its configuration makes it possible to include and exclude commands or features. We analyze BusyBox version 1.18.5[1]. This version consists of 522 files and is configurable in 792 different features [17].

We chose BusyBox as our case study because it is a real-world large-scale product line, which implements variability in the form of C preprocessor directives. Moreover, there is an existing feature model that specifies valid feature combinations.

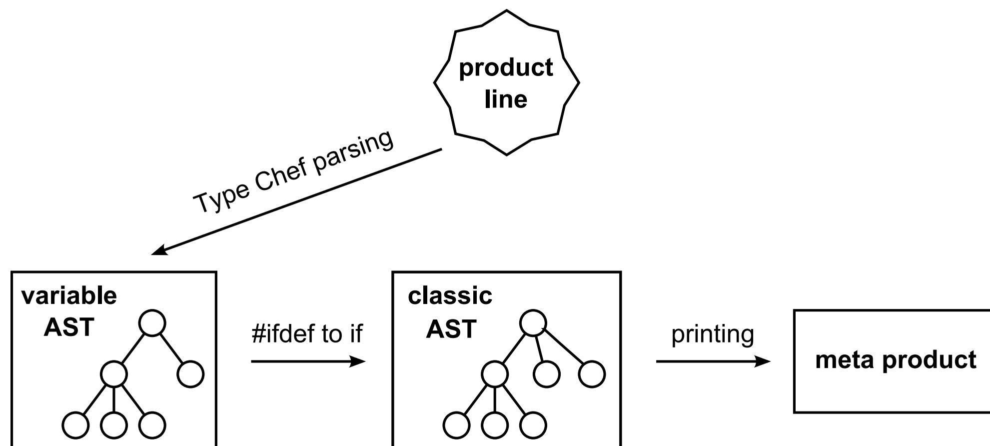


Figure 4.1: `#ifdef` to `if` setup

We use the variability-aware parsing frame work TypeChef to generate a variable AST for each source code file as seen in Figure 4.1. The configuration lifting approach is implemented as a part of the TypeChef framework and is still work in progress[2]. The lifting process itself turns a variable AST into an

[1] http://www.busybox.net/downloads/
[2] https://github.com/aJanker/TypeChef/

AST without variability by applying different transformation strategies as seen in Section 3.1.

In order to verify our configuration lifting approach, we type check the result AST of our transformation. We also provide statistics concerning variability in the BusyBox tool suite and how this variability affects the generation and size of our meta-product. We take a look at the number of AST nodes as to compare the size of the meta product to the original source file. The reason for this is that lines of code are a bad indication in our case. For example renaming an annotated parameter as seen in Figure 3.5 (a) on Line 7-9 turns three lines of code into one line of code as seen on Line 3 of Figure 3.5 (b).

Out of 522 source files we could successfully parse 519 files with the TypeChef framework. Our configuration lifting implementation produces a result that passes TypeChef's type check on 412 files. All numbers presented in this thesis are obtained from these 412 files. We illustrated statistics for our configuration lifting approach in Figure 4.4 where we show example data of results on 11 randomly selected BusyBox files, as well as average and maximum values across the 412 files we mentioned before.

We ran measurements on a machine running Windows 7 Prof. with Phenom II 965, 3.2 GHz and 8 GB RAM. For the remainder of this section we will be talking about our observations and interpret the numbers seen in Figure 4.4.

Metrics

In order to analyze our transformation process we introduce different metrics.

Number of Feature Constants (NF)

The NF metric shows the number of different feature used in a source file. This number gives an indication for the configuration space of a file. This number is acquired by collecting different feature constants from presence conditions inside the file. Our code example from Figure 3.6 consists of three feature constants (`STATIC`, `OFF`, `SET`).

Number of AST Nodes (NOAN)

The NOAN metric reflects the number of AST nodes in the variable AST. When looking at this numbers one has to keep in mind, that TypeChef partially preprocessed files. Consequently the AST of a file f includes the information of all `#include` directives as explained in Section 2.2.1. Hence, the number of AST nodes can be very high for small source files.

AST Node Growth (ANGRO)

The ANGRO metric is the growth factor of AST nodes which occurs because of our configuration lifting. A 10% growth on a file with 1 000 AST nodes implies that the `#ifdef` to `#if` transformation introduced 100 new AST nodes. In this case the final number of nodes in the result AST would be 1 100 nodes.

Number of Declarations (NOD)

The NOD metric shows the number of declarations in the variable AST. This number includes declarations of parameters, as well as `struct`, `enum` and `union` members, but it does not include function parameters, as well as function definitions. The number of declarations in Figure 3.5 (a) consists of two declarations (`val` and `sec_val`)

Ratio of Annotated Declarations (ROAD)

The ROAD metric reflects the ratio of the number of annotated declarations to total declarations in the variable AST. This number indicates how variability is utilized on the declaration level. The ratio of annotated to total declarations in Figure 3.5 (a) is $1 : 2 = 50\%$. The declaration of `sec_val` on Line 8 occurs under presence condition `X64`, while the declaration of `val` on Line 6 is annotation-free.

Declaration Growth (DGRO)

The DGRO metric is the growth factor of declarations which occurs through code duplications due to alternative declarations of the same identifier. In Figure 3.5 (b) on Line 1-2 we introduced two new declarations for the original parameter `val` because there were two possible alternative declarations for `val`. In order to avoid name conflicts we had to rename each of the two new declarations. In this example the number of declarations is three. The original number of declarations was two, so the declaration growth is $(3/2) - 1 = 50\%$. This implies that our transformation increased the number of declarations by 50%.

Number of Functions (NOF)

The NOF metric shows the number of function definitions in the variable AST.

Ratio of Annotated Functions (ROAF)

The ROAF metric reflects the ratio of annotated to total function definitions in the variable AST. This number indicates how many variability is utilized on the function definition level.

Function Definition Growth (FGRO)

The FGRO metric is the growth factor of function definitions when comparing the number of function definitions in the variable AST to the transformed AST. This number is important because function duplications can be a huge factor of growth in AST nodes. In Figure 3.5 we had to duplicate the original function `saveValue` and create two different versions (`_X64_saveValue` and `_NX64_saveValue`). The consequence is that we had to duplicate the AST nodes inside the function body as well.

Number of `if-elif` Statements (NOIE)

The NOIE metric shows the number of `if` and `elif` statements in the variable AST. This number is important because it will be used as a comparison for the transformed AST.

`if-elif` Statement Growth (IEGRO)

The IEGRO metric reflects the growth in `if` and `elif` statements which occurs through statement embedding and statement duplications. This number is the most deciding factor for the growth in AST nodes. There are different scenarios in which we have to add new `if` and `elif` statements. The first scenario are variable declarations. Figure 3.7 (a) Line 9 shows an example of a case where an identifier has two possible interpretations which depend on the selection of the feature `PRECISE`. In these variable declaration-use cases we have to embed the original statement in a new `if` statement. The second scenario are annotated statements. In order to make sure these statements are only executed if their presence condition is fulfilled, we also embed them into a new `if` statement. These two cases are the source of growth of `if` and `elif` statements.

Renamed Identifier Declarations (RD) and Renamed Identifier Uses (RU)

The RD and RU metric count the number of renamings in two different forms. First, we count how many declarations of identifiers we renamed (RD). Second, we count how often renamed identifier usages had to be renamed as a consequence

(RU). These renamings include all identifiers: declarations, `struct`, `enum`, `union` members, as well as function definitions and their usages.

Parsing Time (PT) and Transformation Time (TT)

The PT and TT metric reflect the run times of the parsing process (PT) and our configuration lifting process (TT) in milliseconds.

4.2 Results

First, we take a look at the way our transformation changes the size of the source file. We notice that the percentage growth of AST nodes is mostly concentrated in the 0.6 to 2.0% area. If our transformations do not increase the number of AST nodes significantly, this implies that these source files utilize `#ifdef` directives mostly as optional code fragments instead of alternative code fragments. Another consequence is that using existing analysis and verification tools on these files should take about the same time as testing one variant of the source file. However, there are still files such as `hdparm` which produce a lot of overhead. In the case of `hdparm` our transformation increases the number of AST nodes by 118.57 %, which concludes to more than twice the amount of original AST nodes.

Second, we talk about how our configuration lifting affects declarations. The number of declarations in general is pretty high because a lot of these declarations are included in the form of header files and these included files are also part of the AST. Most of the BusyBox files include a library file (`libbb.h`). This file consists of 31 `#include` directives which can also include even more files. It is also important to look at the ratio of annotated declarations to annotation-free declarations. Across the 412 transformed files which we successfully type checked, this ratio is 7.99% on average. But although 7.99% of the declarations are annotated, the growth in declarations caused by our configuration lifting on average is only 1.08%. This implies that most of the declarations are optional, as shown in Figure 3.5 (a) on Line 8. Declarations have to be duplicated if they are implemented with conditional types or if a symbol is declared in alternative `#ifdef` directives. An example for a declaration duplication can be seen in Figure 3.5 (b) on Line 1-2. However, if the growth of declarations is high this indicates a high use of alternative declarations. In file `hdparm` for example, we found an array declaration which depends on the selection of six different features. This leads to $2^6 = 64$ different variants of this declaration and all these variants are valid according to the underlying feature model.

Third, we look at function definitions. On average in BusyBox 0.95% of

the functions are annotated and the average growth in function definitions through configuration lifting is 0.16%. Duplicating functions can possibly be a huge factor of growth in our transformation result. A function with just an optional parameter has to be duplicated in its entirety. This leads to a significant growth depending on the size of the function body. However this is rarely the case.

Fourth, we analyze `if` and `elif` statements. The number of `if` and `elif` statements gives us an indication of how often we applied our embedding statement transformation from Section 3.1.2. We analyze the growth of `if` and `elif` statements which are introduced through our transformations. This number varies a lot between different files and depends on two different factors. The first factor are references to variable declarations. Figure 3.7 (a) Line 9 shows an example of a case where an identifier has two possible interpretations which depend on the selection of the feature `PRECISE`. In these variable declaration-use cases we have to embed the original statement in a new `if` statement. The second factor are annotated statements. In order to make sure these statements are only executed if their presence condition is fulfilled, we also embed them into a new `if` statement. These two cases are the source of growth of `if` and `elif` statements. The configuration lifting of the file `hdparm` introduces 219 new `if` and `elif` statements. Compared to 53 `if` and `elif` statements in the source file this is a growth of 481.25%. On average however, the number of `if` and `elif` statements only increases by 18.57%.

Fifth, we look at the two numbers concerning renamings. On average our transformations executed renamings on 323 identifier declarations and 172 usages of identifiers. Compared to the average number of declarations and function definitions (3 327) this accounts for 9.7% of all identifier declarations. The number of renamed identifier usages is pretty low but this is another consequence of header file inclusions, as only very few declarations from header files are referenced.

Last, we tracked the time it takes to parse files with TypeChef, as well as the time it takes to perform configuration lifting on the variable AST. Our configuration lifting implementation rewrites the AST by visiting AST nodes multiple times. The transformation times are in the 300-2500 ms range and about 2-8 times faster than TypeChef's lexing + parsing process.

In Figure 4.2 we can see different type check timings for 11 BusyBox files (the same files as seen in Figure 4.4). We ran TypeChef's type checking on the original variable AST (seen in blue), as well as on the configuration lifted AST (seen in orange). The timings are very similar for most files. The configuration lifted AST from `hdparm` has more than twice as many AST nodes as the original AST. The consequence is that TypeChef's type checking on the transformed AST takes more than three times as long as type checking the original AST.

These runtimes give a rough indication of how much our transformations affect type checking runtimes compared to running family-based type checking.

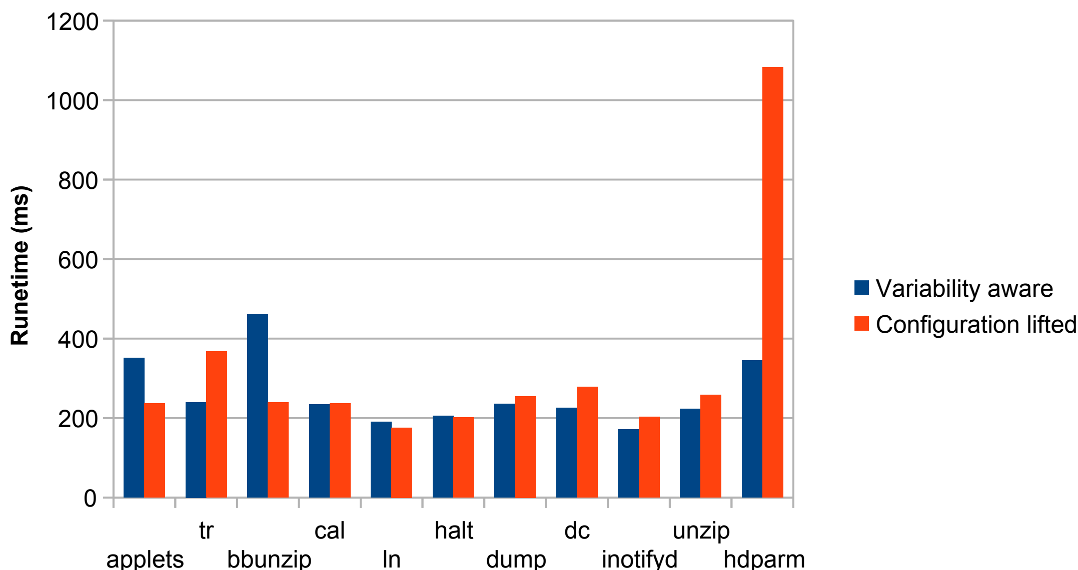


Figure 4.2: Type checking runtimes

Considering that the number of AST nodes stays pretty constant on average we can conclude that type checking our configuration lifted result should take about as long as type checking the original file.

4.3 Problematic Aspects

In this section we will highlight problematic aspects which we encountered during our analysis of configuration lifting on the BusyBox tool suite.

One of the problems are declarations which depend on many different independent features. Figure 4.3 shows an excerpt of the partially preprocessed file `find.c`. What we can see there is a declaration which depends on the selection of 19 independent features. This declaration of `params[]` can have $2^{19} = 524\,288$ different variants and is used for evaluating program arguments. With our implementation we run into out of memory errors when computing feature combinations for more than 17 independent features. It is questionable if this huge variant diversity is intended and required. For example, it could be possible that additional refinements in the feature model drastically reduce the number of possible variants of this declaration. Our configuration lifting approach produces a lot of overhead in cases like these. In the example of file `hdparm` we encountered a similar case with $2^6 = 64$ variants of a declaration. And this caused a huge increase in `if`, `elif` statements, as well as a growth of 118.57% in AST nodes.

Another problematic aspect are variant calculations. These calculations

```
1  static const char params[] __attribute__((aligned(1))) =
2    "-a\0" "-o\0"
3  #ifdef CONFIG_FEATURE_FIND_NOT
4    "!\0"
5  #endif
6  #ifdef CONFIG_FEATURE_FIND_PRINT0
7    "-print0\0"
8  #endif
9  // ... 17 similar ifdefs with different conditions
10 #ifdef CONFIG_FEATURE_FIND_LINKS
11   "-links\0"
12 #endif
```

Figure 4.3: Exponential variants of a declaration

can get complex when dealing with variability on different levels and variable declaration-use cases at the same time. In some cases the TypeChef framework also associates incomplete presence conditions to `Opt` nodes. The consequences are that we generate more variants than actually needed and this leads to type checking errors because e.g., renamings are not applied properly. The current configuration lifting implementation is still ongoing work. In order to fix cases where we create code which yield type checking errors, we have to refine our calculations and transformation patterns.

`switch-case` statements with a high grade of variability or references to variable declarations are also a major source of erroneous or excessive code generation. In order to solve these problems we have to find a sound way of transforming these statements on a finer level of granularity while still preserving the observable behavior. We have to find a strategy that can deal with variability by introducing statement `labeled`-statements and `goto` statements to mimic all possible control flow scenarios.

file name	NF	AST nodes		Declarations			Functions			If-Elif stmts		REN		TIME	
		NOAN	ANGRO	NOD	ROAD	DGRO	NOF	ROAF	FGRO	NOIE	IEGRO	RD	RU	PT	TT
applets	52	85 698	0.70 %	3 294	8.29 %	1.03 %	24	4.17 %	0.00 %	8	25.00 %	326	171	3 658	474
tr	51	88 304	9.64 %	3 320	8.04 %	4.64 %	27	0.00 %	0.00 %	37	213.51 %	322	169	2 030	621
bbunzip	69	90 668	4.14 %	3 415	8.55 %	1.67 %	38	31.58 %	15.79 %	43	120.93 %	382	217	2 309	386
cal	52	97 208	1.34 %	3 496	12.90 %	1.12 %	30	0.00 %	0.00 %	29	62.07 %	485	274	2 327	410
ln	49	86 207	0.63 %	3 286	8.13 %	1.07 %	24	0.00 %	0.00 %	17	0.00 %	322	169	1 948	337
halt	53	89 299	1.07 %	3 329	8.62 %	1.08 %	25	4.00 %	0.00 %	17	47.06 %	347	193	2 434	400
dump	49	93 844	0.59 %	3 377	7.91 %	1.01 %	35	0.00 %	0.00 %	88	0.00 %	320	169	2 391	361
dc	50	103 131	0.96 %	3 776	7.07 %	0.98 %	42	2.38 %	0.00 %	19	15.79 %	327	182	2 653	393
inotifyd	49	86 894	0.68 %	3 308	8.07 %	1.06 %	24	0.00 %	0.00 %	21	9.52%	322	171	2 540	1 081
unzip	61	93 330	2.60 %	3 482	8.24 %	1.38 %	29	6.90 %	0.00 %	65	40.00 %	364	188	3 588	496
hdparm	55	113 350	118.57 %	3 632	8.04 %	2.73 %	53	18.87 %	0.00 %	272	481.25 %	421	392	4 069	2 203
average	50	86911	1.23 %	3300	7.99 %	1.08 %	27	0.95 %	0.16 %	25	18.57 %	323	172	2274	411
max	69	141661	118.57 %	4360	12.96 %	4.64 %	86	37.50 %	15.79 %	272	481.25 %	498	392	4781	2203

NF: number of feature constants; **NOAN**: absolute number of AST nodes in source file; **ANGRO**: AST node growth through transformation; **NOD**: absolute number of declarations in source file; **ROAD**: ratio of annotated to annotation-free declarations; **DGRO**: declaration growth through transformation; **NOF**: absolute number of functions in source file; **ROAF**: ratio of annotated to annotation-free functions; **FGRO**: function growth through transformation; **NOIE**: absolute number of if and elif statements; **IEGRO**: if and elif statement growth through transformation; **RD**: renamed declaration identifiers; **RU**: renamed identifier references; **PT**: parsing time ms; **TT**: transformation time ms

Figure 4.4: Configuration lifting statistics for BusyBox

5 Related Work

Current software projects are getting bigger. Moreover, with the addition of embedded systems, we have many different system architectures. Product lines can drastically cut development costs because they reuse core aspects across all derivable products. This makes software product lines very important for today's development process. But in order to analyze these product lines, it is inevitable to come up with new analysis strategies [21, 22]. Brute-force generation and verification of each possible derivable product is infeasible for most software product lines.

The general configuration lifting approach has been explored by Post and Sinz [2]. They describe three different stages where configuration lifting has to be applied: kBuild (feature model), Make and Preprocessor Code. For the last stage they manually transformed a Linux device driver (`sound/oss/ad1848.c`). This driver can be generated in 32 (almost identical) variants. Post and Sinz came to the conclusion that applying configuration lifting to this device driver gave them a huge speedup in analysis time. The convenient and sequential analysis time for this driver takes $32 \times 71s$ (time for the minimal variant) $\sim 37min$, while analyzing the configuration lifted product took $73s$. The reason for the huge speedups is that the 32 different variants have a very large common code base. However, they had to resort to manually rewrite the source code.

Another very similar approach has been presented by Apel et al. [23] with the intention of detecting feature interactions. Their technique variability encoding is also based on Post and Sinz work on configuration lifting. Apel et al. propose to verify feature interactions by generating a product simulator. This product simulator is able to simulate the behavior of all derivable products of a product line. The goal is to make information about variability and dependencies between features available to model checking tools. This is done by encoding variability information into the code base of the product simulator. Apel et al. analyzed carried out a case study on AT&T E-Mail Client and came to the conclusion that they were able to detect all feature interactions without generating all possible feature combinations. In their case study the variability encoding approach is about ten times faster than the brute-force approach.

Liebig et al. analyzed different analysis methods [17] (product-based, sample-based and familiy-based analysis) which were explained in Section 2.3. They also used the TypeChef framework and tested the different analysis meth-

ods on BusyBox and the Linux kernel. These two different software product lines were analyzed variability-aware for type-errors and dataflow analysis. Liebig et al. came to the conclusion that for type-checking BusyBox their family-based analysis takes as much time as testing eight variants sequentially. The family-based analysis approach takes all variants into account which is not the case for sample-based analysis. Consequently this approach is very promising and shows the potential of making analysis methods variability-aware.

6 Conclusion

We presented our configuration lifting techniques as an approach of encoding the variability from conditional preprocessor directives (`#ifdef`s) into constructs of the standard C programming language. We highlighted our intention to keep the variability from `#ifdef`s on a fine level of granularity where possible. Then we applied our transformations on the BusyBox tool suite and analyzed different results.

Our experimental results for 412 BusyBox files show that configuration lifting produces a small amount of growth in AST nodes (on average 1.23%). The reasons for this are that most variability occurs in the form of header file inclusions and optional annotations instead of alternative annotations. Also the number of new functions we introduced on average is very low (0.16%).
The biggest factor for the growth of AST nodes is the introduction of new `if` and `elif` statements as a consequence of statement embedding and code duplications. The reasons we have to embed existing statements into new `if` and `elif` statements are references of identifiers with more than one possible declaration (e.g. a declaration with alternative types) and annotated statements as seen in Figure 3.5.
We found that our transformation process is rather quick in comparison to the parsing process of TypeChef. On average it takes 411 ms to transform an AST of a BusyBox source file with 50 features. Applying configuration lifting needs a lot less computation time than parsing a source file.

We also found out that TypeChef's type checking the configuration lifted ASTs takes about as long as testing the original AST. But in very rare cases where our transformations introduced a lot of additional code the type checking process on the transformed AST can take much longer compared to checking the original AST.

However, we also encountered problematic aspects during our transformation process. Code duplications which are exponential to the amount of presence conditions can cause complications. Dealing with more than 15 different and independent presence conditions and thus generating 2^{15} variants e.g., of a declaration is not feasible.

With TypeChef's `PrettyPrinter` module we are able to turn our configuration lifted AST into source code. This source code can then be utilized

by conventional analysis and verification tools. In further work we want to refine our transformation process and apply existing analysis and verification tools on our generated meta-product. We also want to develop an implementation of behavior-preserving transformations on `switch-case` statements on a finer level of granularity.

Another important task is to transform LINUX drivers and look for erroneous configurations with model checking tools. If we want to make information about valid feature combinations available for verification tools we have to implement a feature model representation in C. This reduces analysis effort as only valid feature combinations are tested.

7 Acknowledgments and Project Repository

First I want to thank my supervisors prof. Christian Lengauer, Ph.D. and Dipl. Ing.-Inf. Jörg Liebig. Special thanks go to Jörg Liebig for his constant feedback during my work. He helped me in understanding the SCALA programming language and how to use and extend the TYPECHEF framework.

Moreover I want to thank the Chair for Programming at the University of Passau for recruiting me as a student research assistant. They invited me to present my work at a national meeting of product-line researchers. Furthermore I'd like to thank Alexander von Rhein for many fruitful discussions and feedback at later stages of this thesis.

Finally, I thank my student colleague Andreas Janker. Together we developed the variability-aware declaration-use map, which is described in Section 3.3.1.

Our configuration lifting implementation is integrated into the TYPECHEF project and published as open source under GPL 3.0[1]. This project is still ongoing work and the current development version can be retrieved from github at:

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

The version we used to generate the data for statistics presented in this thesis can be found at:

`https://github.com/aJanker/TypeChef/archive/v0.2.zip`

[1] `https://github.com/ckaestne/TypeChef/blob/master/LICENSE`

List of Figures

2.1 A sample feature model 5
2.2 #include example 6
2.3 #define example 7
2.4 Extract of a variable list implementation using CPP 8
2.5 Nested #ifdefs 9
2.6 Different approaches for product line analysis 10
2.7 The TypeChef Framework 11
2.8 Enforcing disciplined annotations 12
2.9 Choice nodes represented in source code and AST form 13

3.1 Creating a feature selection representation 15
3.2 Embedding variable statements into if statements 16
3.3 Renaming declarations and function definitions 17
3.4 Evaluating Choice nodes 18
3.5 An example of transformation combinations 20
3.6 Variability on different levels 22
3.7 Variability-aware declaration-use mapping 24
3.8 Reconstructing the #if-#elif-#else structure 26

4.1 #ifdef to if setup 27
4.2 Type checking runtimes 33
4.3 Exponential variants of a declaration 34
4.4 Configuration lifting statistics for BusyBox 35

Bibliography

[1] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. *School of Computer Science, University of Magdeburg, Germany, Technical Report FIN-004-2012*, 2012.

[2] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, pages 347–350. IEEE CS, 2008.

[3] P. Clements, L. Northrop, et al. A framework for software product line practice. *SEI Interactive*, 2(3), 1999.

[4] K. Pohl, G. Bockle, and F. Van Der Linden. *Software product line engineering*, volume 10. Springer, 2005.

[5] 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.

[6] C. Kästner and S. Apel. Feature-oriented software development. In *Generative and Transformational Techniques in Software Engineering IV*, pages 346–382. Springer Berlin Heidelberg, 2013.

[7] K. Lee, K. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. *Software Reuse: Methods, Techniques, and Tools*, pages 62–77, 2002.

[8] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, J. Sincero, and W. Schröder-Preikschat. Configuration Coverage in the Analysis of Large-Scale System Software. In *Proc. Workshop an Programming Languages and Operating Systems (PLOS)*, pages 1–5. ACM Press, 2011.

[9] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 191–202. ACM, 2011.

[10] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proc. Int. Conf. Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM Press, 2011.

[11] Y. Padioleau. Parsing C/C++ Code without Pre-processing. In *Proc. Int. Conf. Compiler Construction (CC)*, pages 109–125. Springer, 2009.

[12] L. Vidács and Á. Beszédes. Opening up the C/C++ Preprocessor Black Box. In *Proc. Symp. Programming Languages and Software Tools (SPLST)*, pages 45–57. University of Kuopio, Department of Computer Science, 2003.

[13] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. pages 185–197, 1992.

[14] M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Software Engineering (TSE)*, 28(12):1146–1170, 2002.

[15] Y. Hu, E. Merlo, M. Dagenais, and B. Lague. C/c++ conditional compilation analysis using symbolic execution. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 196–206. IEEE, 2000.

[16] P. Livadas and D. Small. Understanding code containing preprocessor constructs. In *Program Comprehension, 1994. Proceedings., IEEE Third Workshop on*, pages 89–97. IEEE, 1994.

[17] 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, Technical Report MIP-1212, Passau, Germany: Department of Informatics and Mathematics, University of Passau, 2012.

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

[19] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Software Engineering and Methodology (TOSEM)*, 21(1):6, 2011.

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

[21] A. Classen, P. Heymans, P. Schobbens, and A. Legacy. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 321–330. ACM, 2011.

[22] A. Classen, P. Heymans, P. Schobbens, A. Legacy, and J. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 335–344. ACM, 2010.

[23] S. Apel, Hendrik Speidel, Philipp Wendler, A. von Rhein, and Dirk Beyer. Feature-aware verification. *arXiv preprint arXiv:1110.0021*, 2011.