BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATIC GENERATION OF CODE CHANGE PATTERNS

AUTOMATICKÉ GENEROVÁNÍ ŠABLON ZMĚN KÓDU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DANIEL KŘÍŽ**
AUTOR PRÁCE

SUPERVISOR **prof. ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

146333

Institut:	Department of Intelligent Systems (UITS)
Student:	**Kříž Daniel**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Automatické generování šablon změn kódu**
Category:	Software analysis and testing
Academic year:	2022/23

Assignment:

1. Get acquainted with DiffKemp, a tool for automatic analysis of semantic differences between versions of software. Focus on the way DiffKemp handles user-defined patterns of semantically equivalent changes.
2. Study existing algorithms for graph generalization and for computing a maximum common subgraph. Explore ways to apply these algorithms on program representation graphs.
3. Propose a method to automatically generate a code-change pattern from a given number of similar code differences identified by DiffKemp. The format of the pattern must comply with the way DiffKemp represents patterns of changes.
4. Implement the proposed method inside the DiffKemp framework.
5. Demonstrate usefulness of your solution by generating several different patterns from changes found in the Linux kernel or other projects supported by DiffKemp. Apply the generated patterns within analysis of semantic equivalence of multiple versions of the chosen projects and discuss the obtained results.
6. Summarize and discuss the achieved results and their possible further improvements.

Literature:

- Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale C projects. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 329–339. IEEE (2021)
- Malík, V., Šilling, P., and Vojnar, T.: Applying Custom Patterns in Semantic Equality Analysis. In: The 10th Edition of the International Conference on NETworked sYStems (NETYS). to be publised. Springer (2022)
- Kann, V.: On the approximability of the maximum common subgraph problem. In: Annual Symposium on Theoretical Aspects of Computer Science. Springer (1992).
- Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 727–739 (2017)

Requirements for the semestral defence:
The first two points of the assignment and at least some initial work on the third point.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**
Consultant:	Malík Viktor, Ing.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

The aim of this thesis is to propose a method for automatic generation of custom code change patterns in LLVM IR language for DiffKemp, a tool for analyzing semantic differences between version of large scale projects. The goal is to enable automatic generation of changes between versions of a project with values, global variables or structure types. This has been achieved by finding the common pattern between changes and then generating its variants, which differ in usage of global variables and types. The proposed solution was implemented as an extension of DiffKemp and our experimentation on small programs shows that out proposed method is able to yield at least partially satisfactory results.

Abstrakt

Cílem této bakalářské práce je navržení metody automatického generování šablon změn kódu v jazyce LLVM IR pro DiffKemp, nástroj pro analýzu sémantických rozdílů mezi verzemi rozsáhlých programů. Dále je cílem umožnit automatické parametrizování změn mezi verzemi projektu pomocí hodnot, globálních proměnných a strukturových typů. Toho bylo dosaženo pomocí nalezení společné šablony mezi změnami a následným generováním jejích variant, které se liší v použití globálních proměnných a typů. Navržené řešení je implementováno jako rozšíření nástroje DiffKemp a naše experimentování na malých programech ukázalo, že námi navržená metoda je schopná vytvořit alespoň částečně uspokojivé výsledky.

Keywords

DiffKemp, LLVM, LLVM IR, static analysis, GNU/Linux Kernel, automatic pattern inference, automatic pattern generation

Klíčová slova

DiffKemp, LLVM, LLVM IR, statická analýza, GNU/Linux Kernel, automatické odvozování šablon, automatické generování šablon

Reference

KŘÍŽ, Daniel. *Automatic Generation of Code Change Patterns.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

V dnešní době existují projekty, pro které je nejvyšší prioritou stabilita. Takové projekty investují nezanedbatelné množství času a zdrojů do dokazování, zda některé změny v kódu nezavedly neočekávané vedlejší účinky, které by mohly tuto stabilitu ohrozit. Ve snaze tento proces automatizovat se vývojáři mohou zkusit obrátit na statickou analýzu semantických rozdílů. Avšak problémem těchto metod je častá závislost na formálních metodách, které dokážou být přesné, ale na druhou stranu také vysoce výpočetně náročné, a proto je jejich použitelnost na rozsáhlé projekty značně omezena, protože dobře neškálují s velikostí projektu.

Naštěstí se ale už objevují analyzátory, které se zaměřují na škálovatelnost a použitelnost na velmi rozsáhlé projeky. Jedním z těchto nástrojů je i DiffKemp, analyzátor sémantických rozdílů mezi různými verzemi projektu. DiffKemp se snaží být napůl cesty v nikdy nekončícím kompromisu mezi přesností analyzující metody a rychlostí výpočtu, a proto je schopný poskytnou výsledky za rozumně dlouhou dobu, a i přesto uživatele informovat o sémantické ekvivalenci funkcí mezi dvěma verzemi programu. Aby byl schopný dosáhnout takové škálovatelnosti, DiffKemp musí použít řadu technik. Na začátku analýzy přeloží cílový porovnávaný program do mezijazyka projektu LLVM (LLVM IR) a následně se pokusí verze programu porovnat po jednotlivých instrukcích. Avšak takové porovnání není vždy možné, protože to by vyžadovalo, aby kód mezi verzemi byl i syntakticky stejný, a proto může porovnávání po instrukcích přinést vysoké množství falešných neekvivaletních výsledků. Aby DiffKemp mohl dovolit korektní porovnávání po instrukcích, tak často jak je jenom možné, tak provádí řadu transformací a umožňuje i hledání instancí šablon, které jsou známé svou schopností zachování sémantiky. Vedle těchto sémantiku zachovávajících vzorů může uživatel nástroje definovat i své vlástí šablony změn v LLVM IR, pomocí kterých může specifikovat změny, které budou ignorovány během analýzy. Díky tomuto může nástroj ignorovat i změny, které jsou specifické pro konkrétní projekt a pro které se vývojáři rozhodli, že jsou bezpečné (takové změny můžeme tedy označit za sémantiku měnící), pokud by se objevily během vývoje. Ale naneštěstí, protože jsou tyto vzory zapsány pomocí mezijazyka LLVM IR, který nebyl navržen pro ruční psaní, tak dokáže být definování vzorů náchylné na chyby a dokáže být i náročné na čas a schopnosti.

Vzhledem k výše zmíněným informacím tato práce navrhuje metodu automatického odvozování a generování vlastních šablon změn kódu, aby uživatel měl možnost jenom poskytnout vybrané změny, které by měly být ignorovány během dalších běhů programu. Toto odvozování je založené na parametrizaci grafů běhu program CFG pomocí hodnoty. Produkt takového odvozování může být dále parametrizován pomocí globálních proměnných a strukturových typů, ale protože aktuálně v nástroji DiffKemp neexistuje žádná podpora pro parametrizaci instrukcí (protože taková změna by vyžadovala rozsáhlé změny v hlavní funkcionalitě analýzy) a nebo žádné dostupné prostředky na zobecnění, tak musíme generovat více šablon pomocí jejích variant, které vznikly na základě použití různých proměnných a typů.

Navržená metoda je zakomponována jako rozšíření nástroje DiffKemp a umožňuje vygenerovat šablony z poskytnutých změn s podporou generování variant těchto šablon z výše zmíněných důvodů. Rozšíření navíc dokáže stanovit i rozsah platnoti šablony, který je využitelný při optimalizaci detekování šablon při analýze.

Experimenty ukázaly, že naše implementace má určité nedostatky v generování aplikovatelných šablon z parametrizovaných změn, ale na druhou stranu jsme schopni úspěšně vygenerovat aplikovatelné šablony. které jsou neparametrizované.

Automatic Generation of Code Change Patterns

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of prof. Ing. Tomáš Vojnar PhD. The supplementary information was provided by Ing. Viktor Malík I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Daniel Kříž
May 9, 2023

Acknowledgements

I would like to thank my consultant Ing. Viktor Malik for all his guidance and all his help with the understanding of DiffKemp and writing of this thesis. I would also like to thank the company RedHat for supporting our work.

Contents

Chapter 1

Introduction

There are projects for which the stability is the main concern, such projects invest significant amount of time and resources into proving that changes made to their codebases did not cause any unforeseen consequences that would compromise stability of said systems. In order to at least partially automize the proving process, developers may turn to use *static analysis of program semantic differences*. The goal of such analysis is to collect some information about the behavior of a program from its source code without executing it under its original semantics [19]. But the problem with such methods is that they usually depend on computationally intensive formal methods, which can be on one hand accurate, but on the other hand they may take extended periods of time and high amount of resources to provide some benefit. Therefore, their application to large-scale projects is fairly limited as they do not scale well with the size of the project.

But fortunately, there are analyzers that focus themselves on scalability and applicability on large-scale projects. One of such tools is DiffKemp a tool focused on finding semantic differences between versions of a project. DiffKemp seeks a middle-ground in the everlasting trade-off between an analysis method accuracy, and its speed. It provides results reasonably fast, while still being able to provide the user with information about semantic equivalence of functions between two version of a program. To achieve high scalability DiffKemp utilizes a few different techniques. At the beginning of the analysis it translates the compared program to LLVM intermediate representation (LLVM IR) language and then attempts to compare them *instruction-to-instruction*, but such comparison may not always be possible, as it would require the two versions of the program to be syntactically the same and therefore, it would produce a high amount of false non-equivalence results. In order to allow the instruction-to-instruction comparison as often as possible DiffKemp performs a number of various transformations and allows to search from predefined patterns that are known to preserve semantics. Next to these so-called *semantics-preserving change patterns (SPCPs)*, the user may define their own *custom change patterns* in LLVM IR which enable to specify which kinds of changes should be ignored during comparison. This made it possible to ignore project-specific changes, that are considered safe (hence, they could even be viewed as *semantics-altering*) and may occur during development. However, they are encoded in LLVM IR, which was not designed with programmers ergonomics in mind and therefore, defining CCPs is quite error-prone, and can get time-consuming and cumbersome.

This thesis proposes a method for automatic inference and generation of CCPs, so that the user may only provide chosen changes that should be ignored in future runs. This inference is based on parametrization of *control-flow graphs (CFGs)* by value. The product

of this inference can be further parametrized by global variables and structure types, but as there is no support for parametrization of instructions (as that would require extensive changes in the core parts of DiffKemp) or any accessible means of type generalization we have to generate more patterns in variation to the encountered variables and types. Then we are able to determine pattern ranges, which denote the first differing instruction between pattern sides and an end of such pattern.

This thesis is organized as follows. Chapter 2 presents DiffKemp in more detail together with overview of LLVM IR. Chapter 3 is devoted to the description of custom code change patterns and algorithms used for their matching. Chapter 4.3 describes the design of proposed method for automatic pattern generation. Chapter 5 provides details about the implementation of proposed method as an extension of DiffKemp. Chapter 6 evaluates the implementation. Finally, Chapter 7 concludes the thesis.

Chapter 2

DiffKemp

DiffKemp is a *static analysis* tool for finding *semantic differences* between versions of the same project which aims for high scalability, with applicability to large-scale industrial grade projects such as the Linux kernel.

The necessity for such a tool springs from two fundamental assumptions:

1. Automatic checking of semantic equivalence of programs nowadays usually relies on heavy-weight formal methods, which are very accurate but tend to be slow and consequently have problems with scalability

2. Existing light-weight methods are fast and scalable, but on the other hand do not perform a proper semantic equivalence analysis, rather than a textual or a syntactic equivalence analysis.

To overcome these limitations, DiffKemp takes a middle ground as it can analyze large-scale projects in a matter of minutes while not providing many *false-positive* results. This is achieved using a combination of three concepts:

- *Per-instruction* comparison on the level of a low-level language, in this case of *LLVM intermediate representation* (LLVM IR). Such approach is fast, simple, and very scalable, but it may produce numerous *false non-equivalence* results.

- In order to prevent such *false-positives*, DiffKemp *pre-processes* the code using various static analyzes and code transformations (such as function inlinening, constant propagation and redundant, and dead code eliminations).

- For cases where none of the mentioned approaches is sufficient, DiffKemp contains a list of *semantics-preserving change patterns (SPCPs)* which are evaluated as semantically equal, if they match some case that would otherwise be evaluated as non-equal.

In Figure 2.1 we can see the overview of the DiffKemp architecture. Simply put, users provide two versions denoted as *old* and *new*, of the project to the *Snapshot generator*, which then provides so-called LLVM IR snapshots of the project versions. These are passed to the *Snapshot comparator*, which then compares their semantics using the *Analysis core*. We can look at this core in two different ways:

As we can see in the Figure 2.1, *Analysis core* also contains a set of *built-in semantics-preserving patterns*, those are the SPCPs mentioned earlier. It can also take conditional, user-defined *Custom Change Patterns*, which are further explained in Chapter 3. Prior to

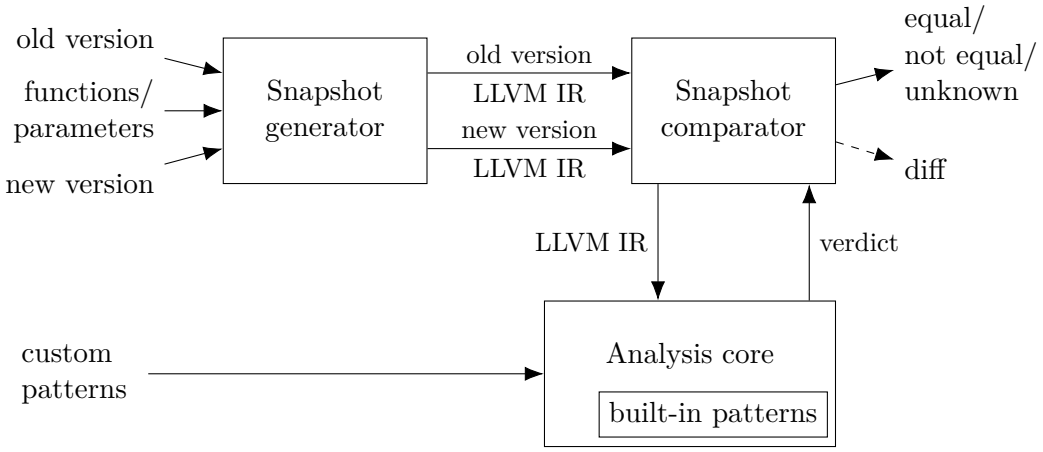


Figure 2.1: Diagram of DiffKemp architecture presented in [11]

our work, such patterns had to be specified manually and the main focus of this thesis is a method for automatic inference of these patterns. Our contribution is further described in Chapter 4.

In the rest of this chapter, we describe the underlying concepts of DiffKemp. Most information regarding DiffKemp and its parts comes from [14] and [13]. The chapter is organized as follows. In Section 2.1, we explain the internal representation of programs, LLVM IR. A top-level algorithm for function equality, together with its definition, are presented in Section 2.2. Finally, in Section 2.3, we explain usage of SPCPs and briefly introduce custom code change patterns, which are discussed in further detail in Chapter 3.

2.1 Representation of Programs

Static analyzers typically use some form of internal representation of the analyzed programs, usually based on graph structures such as *control-flow graphs* (CFGs). DiffKemp takes advantage of an already existing representation used by LLVM-based compilers, the so-called *LLVM intermediate representation (LLVM IR)*. This brings many advantages, as not only the language/representation is already specified and widely used (therefore, we can expect it to have certain qualities) but it also makes it possible to extend DiffKemp to other languages than C (since many languages can be compiled to LLVM IR).

LLVM IR breaks higher-level language features into lower-level instructions, which makes it possible to simplify reasoning about program semantics, hence providing us with means for further optimization and static analysis. In addition, as opposed to native assembly instruction, it is architecture independent.

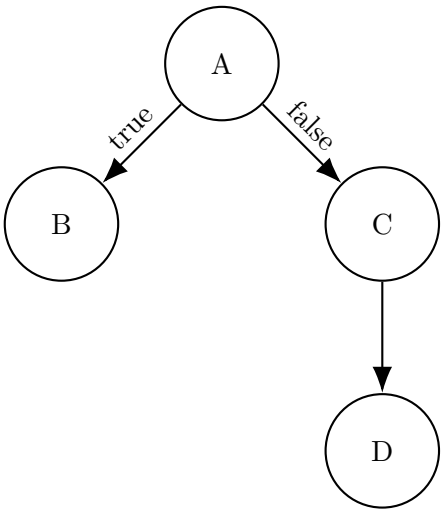
In this subsection, we are going to explain some basic building blocks of LLVM IR. We have chosen a top-down approach for this explanation meaning that we are going to describe higher-level constructs, such as control flow graphs and basic blocks, first. Then, we will focus on lower-level constructs, such as instructions, metadata, and values. All of these concepts are explained in higher detail in [6].

Control-flow graph (CFG)

A control-flow graph is the highest structure of representation of programs as it represents *flow of control between branches*. In LLVM IR, each function corresponds to a single *CFG*. It is a directed graph which can be defined as follows:

$$CFG = (BB, E). \tag{2.1}$$

That means that a *control-flow graph* consists of a finite nonempty set BB of *basic blocks* and a finite set $E \subseteq BB \times BB$ of edges that represent control flow between basic blocks and its underlying branching. The difference between a *CFG* and a general directed graph is that *CFG* has exactly one entrance basic block that has no predecessors. The whole flow may end in multiple ending basic blocks, which represent different `return` statements from the function, however, every *CFG* can be simplified to have only a single ending basic block.



(a) Corresponding CFG

```c
const int answer = 42;
int global_val = 10;
int func(int x) {
    // Basic block A
    if (x < global_val) {
        // Basic block B
        return answer;
    } else {
        // Basic Block C
        ++x;
    }
    // Basic Block D
    return x;
}
```

(b) Example function in `C`

Figure 2.2: Code with CFG example

In Figure 2.2, we give an example of a code in C (Figure 2.2b) and its corresponding *CFG* (Figure 2.2a). Each basic block has an assigned letter (from A to D) which is marked in the code with a comment. In the *CFG*, we have a single entry basic block (A) which branches into basic blocks (B) and (C) depending on the result of a boolean expression between program variables. Both branches end with a `return` statement, resulting in a transfer of the control to the outer scope, or the end of the program.

Basic Block (BB)

A basic block is a labeled list of LLVM IR *instructions* that do not contain any branching instructions in the middle of the block. In addition, it has to end with exactly one *terminator instruction* (either a jump to a different basic block or a return from the function).

```
1  @answer = constant i32 42, align 4
2  @global_val = global i32 10, align 4
3  define i32 @func(i32 noundef %0) #0 { ;label, basic block A
4    ;basic block B
5    %2 = alloca i32, align 4 dbg 10 ;metadata
6    store i32 %0, i32* %2, align 4
7    %3 = load i32, i32* %2, align 4
8    %4 = load i32, i32* @global_val, align 4
9    %5 = icmp slt i32 %3, %4
10   br i1 %5, label %6, label %7
11
12 6: ;label, basic block C
13   store i32 42, i32* %2, align 4
14   br label %7
15
16 7: ;label, basic block D
17   %8 = load i32, i32* %2, align 4
18   ret i32 %8
19 }
```

Figure 2.3: Code example of basic blocks

Figure 2.3 shows a commented structure of LLVM IR for a sample function written in C. For basic blocks, the most notable are labels which denote the start of each block (with an implicit entry basic block, right after the function header) and can be used as target of jumps and branching. For the sake of brevity, labels are marked with a comment in the code.

As stated before, basic blocks cannot contain branching instructions except for the end. Therefore, branching is done by calling one of the terminator instructions with reference to a label as a target, thus ending span of such basic block, as can be seen on Lines 10 and 14 in the example in Figure 2.3.

When branching occurs at the end of a basic block, then it can take one of two forms:

1. An unconditional branching, which is the simpler one of the two. It is simply done without checking any condition and the control flow traverses to the next BB. An example of such branching can be seen in Figure 2.2a as the edge between node C and D.

2. A conditional branching, that is dependent on some condition and the flow can take one of two courses: either the condition has been evaluated as *true* and the corresponding edge is taken or it has been evaluated as *false* and the flow descends to the other branch. Example of such branching can be seen again in Figure 2.2a, but this time it is the relation between the node A and the nodes B and C.

Instruction

As is mentioned in Section 2.1, LLVM IR remotely resembles some kind of assembly language. Each instruction has an operation *op*, it a takes possibly empty list of operands $(o_1, o_2, ..., o_i)$, and it can yield some value v. In such a case, it creates a new variable,

because LLVM IR adheres to the so-called *static single-assignment (SSA)* property, which requires that each variable can be assigned to at most once. For the purpose of this work, we introduce the following notation. For representing the just described instruction:

$$v = op(o_1, o_2, ..., o_i). \tag{2.2}$$

Instructions capture a semantics of program as they are the input to the last step of a computer program compilation, translation to the corresponding assembly of a target machine. As the main goal of DiffKemp is to apply static analysis to answer the question of semantic equality of two versions of the same program. Hence, instructions are inseparable from the analysis done by DiffKemp as it usually compares semantics on the *instruction-to-instruction* basis.

Values

Values are used in instructions at the place of operands and results. They can be mutable (variables) or immutable (constants). Variables can be further separated into three groups, depending on their location in memory and scope:

1. *Local variables* – These are accessible only from the function that they are defined in. Local variables are stored in so-called virtual registers, and have to comply to the *SSA* property mentioned earlier. An example of a local variable can be seen in Figure 2.3 on Line 5.

2. *Local stack-allocated variables* – Just as normal local variables, these are only accessible in a function scope but with the difference that they are allocated on the stack. Such variables are accessed via pointers and operable using load and store instructions, and often correspond to the local variables from the original program. In Figure 2.3, we can see an allocation of such a variable on Line 5 using the `alloca` instruction. On Lines 6 and 7, the variable is written to and read from using instructions `store` and `load`, respectively.

3. *Global variables* – In contrary to local variables, global variables are accessible from every point of the program. They work in the exact same way as stack-allocated variables, except that they do not have to be allocated with `alloca`. In the code, they can be recognized thanks to a different prefix (`@`). An example of a global variable can be seen in Figure 2.3 on Line 2.

Constants can occur in two forms: (1) either they can be defined in the global context with a `constant` property, or (2) they can be represented as inlined values (so-called literals). Thanks to compiler optimizations, we see can both of these forms for a single constant in Figure 2.3 on Lines 1 and 13.

Types

One of the most important features of LLVM IR is its strong type system. Thanks to this compilers can perform number of optimizations directly, without the need to do some extra analysis.

There are three main groups of types:

1. The void type – It does not represent any value and has no size.
2. The function type – It resembles function signature as it consists of return type and list of formal parameter types.
3. Single value types – This group is the only one that consist of more than one type, and it represents all types that can be assigned to virtual registers discussed earlier.

Single value types are the most important to this work as they are used in the custom change patterns discussed in Chapter 3. This class of types contains all the well-known types as integers, floats, etc. but also structure types, pointers and metadata.

A type that may need a further explanation is the pointer. Especially now as it has recently risen higher in the importance with LLVM 15 as there are now two versions of it (1) old *explicit pointer* which had many flaws and it is going to be deprecated in some new versions of LLVM, and (2) *opaque pointer* which is its replacement. The difference between these two is that explicit pointer contains information about the pointee type, this information is missing in the opaque pointer. The need for the latter sprang from the fact, that most operations in LLVM IR do not really care for an actual underlying type in memory (as these instructions usually take some arbitrary type and sometimes a size) and pointee type does not necessarily represent it either, hence it does not carry any real semantics. Existence of the pointee type only complicates optimizations. This is explained in further detail at [7].

Probably the best way how to get the idea of opaque pointers is through an example in some higher, better known language such as the C language:

```
1 int x = 42;
2 int *ptr = &x;
3 void *opaque_ptr = &x;
```

Figure 2.4: Example of opaque pointer in `C`

Here in Figure 2.4 we can see what it means for the pointer to be opaque. Its declaration does not carry any information about its type, hence it is *opaque*. An example of the difference between typed and opaque pointer can be seen in Figure 2.5.

```
1 %0 = load i64, i64* %p
2 %1 = load i64, ptr %p ;opaque pointer
```

Figure 2.5: Example of difference between typed and opaque pointer in LLVM IR

Metadata

LLVM IR allows to attach metadata to instructions and global objects. Such data can contain extra information about the code, that can be used in further language processing and analysis.

Probably the most notable usage of metadata is injection of debugging information to programs as can be seen on Line 4 in Figure 2.3 with the `!dbg` and `!22` metadata.

```
1 !0 = !{ !"and", i32 42 }
2 !1 = !{ !"eggs" }
3 !foo = !{ !0 }
4 %0 = alloca i32, align 4 !"spam" !foo !1
```

Figure 2.6: Example of different types of metadata and their combination in LLVM IR

In Figure 2.6 we present an example of metadata usage. As can be seen, metadata are denoted by „!“ at the beginning of an entity name or of a string, called in this context *the metadata string*.

A single metadata object can hold a list of other metadata objects, creating a hierarchical structure. In such a case, the holder object is called a metadata node and the list of contained objects is denoted with `{}`. Further, metadata nodes are divided into two groups:

1. Unnamed metadata – This metadata does not have a name (but it is assigned a number by the compiler or programmer). It can hold virtually any value. Example of unique metadata can be seen on Line 1 in Figure 2.6.

2. Named metadata – Unlike the unnamed metadata, it can hold only other metadata as its operands, but on the other hand its name is searchable in the module symbol table, which can be useful for identifiers and tagging (such as the mentioned `!dbg`). An example of a named metadata that holds the metadata from the previous list item is on Line 3 in Figure 2.6.

Usage of all the forementioned types of metadata can be seen in Figure 2.6 on Line 4, where we allocate a local variable and attach additional information to it using metadata. There are three distinct kinds of metadata on the line, but in the end it is going to be combined (by appending the latter ones to the first one) into a single metadata node, holding `!{!"spam", !"and", i32 42, !"eggs"}` as its operands.

Metadata are a very important resource for declaration of user-defined patterns, which are the central topic of this thesis. We describe their usage for this purpose in Section 3.2.

2.2 Function Equality Analysis

In this section, we are going to define the problem of semantic equality between two functions, and then we are going to show and explain the most important algorithms used by DiffKemp for checking semantic equivalence.

The approach, that Diffkemp takes circles around the idea of finding so-called *synchronization points* between the analyzed functions. Pairs of these points denote places in which the compared function should be in a semantically equivalent state. Typically, but *not always*, such points are placed at each instruction. In the following text, we denote I_1, I_2 the sets of instructions of the compared functions f_1 and f_2 respectively. Formally, the *problem of checking semantic equality* of functions f_1 and f_2 can be viewed as the problem of finding:

- two sets of synchronizations points $S_1 \subseteq I_1$ and $S_2 \subseteq I_2$
- two synchronization functions *smap*: $S_1 \leftrightarrow S_2$ and *varmap*: $V_1 \leftrightarrow V_2$

Here *smap* and *varmap* are bijections that represent mapping of synchronization points (S_1 and S_2) and variables (V_1 and V_2), respectively between f_1 and f_2.

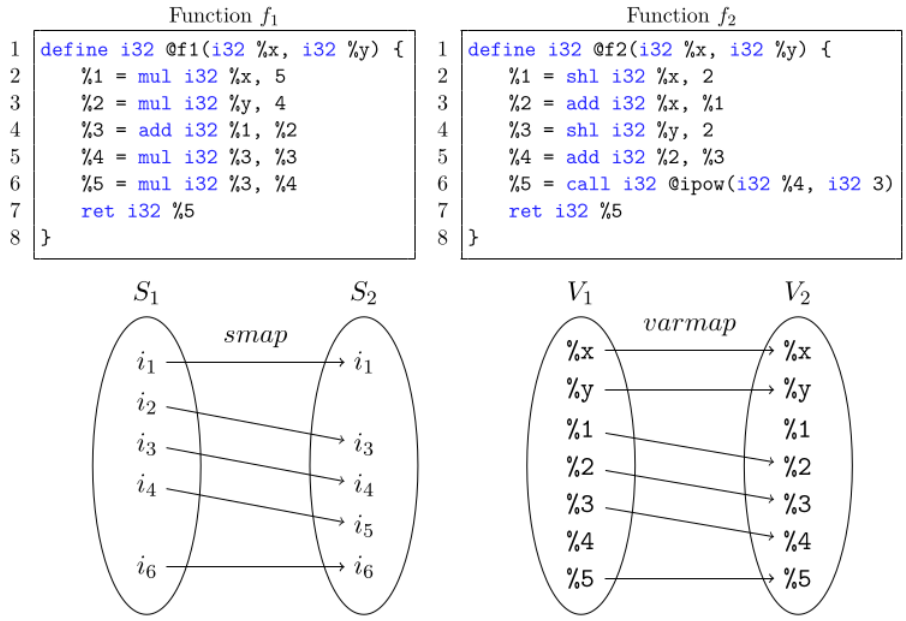


Figure 2.7: Example of *smap* and *varmap* for comparison of functions f_1 and f_2 and their corresponding representation in LLVM IR, which has been introduced in [20]

Proving semantic equality is a rather difficult task, especially for large blocks of code, and it can be very time-consuming, therefore DiffKemp applies *semantic preserving code transformations* in order to allow analysis *per-instruction* as often as possible. Application of these transformations should allow us to compare two programs by the *instruction-to-instruction* comparison. This approach is quite straight-forward as two instructions are semantically equal only if they perform the same operation on the same operands (or on operands, that are mapped via *varmap*). To put this in a perspective we can look at the example in the Figure 2.7 where two semantically equal functions (as they both calculate

mathematical expression $(5x + 4y)^2$) are mapped to each other with *smap* and *varmap*. As we can see some instruction and virtual registers are mapped to their counterpart with different order in the other function (e.g. $i_2 \rightarrow i_3$ between S_1 and S_2), which means, that we can expect the same result in corresponding register (for the previous example that would be %2 $\rightarrow$ %3 between V_1 and V_2).

All of these operations are handled by the main algorithm which we are now going to present as Algorithm 1 and explain it in further detail.

Input : Functions f_1 and f_2
Output: *true* if f_1 is semantically equal to f_2, *false* otherwise
1 run transformations of f_1 and f_2
2 **if** $|P_1| \neq |P_2|$ **then return** false
`// initialization of synchronization maps`
3 $S_1 = \{i_{in}^1\}, S_2 = \{i_{in}^2\}$
4 $smap(i_{in}^1) = i_{in}^2$
5 **for** $1 \leq i \leq |P_1|$ **do** $varmap(p_i^1) = p_i^2$
6 **for** $g_1 \in G_1$ **do**
7 $varmap(g_i) = g_2$ where $g_2 \in G_2$ has the same name as g_1
`// Main loop`
8 $Q = \{(i_{in}^1, i_{in}^2)\}$
9 **while** Q *is not empty* **do**
10 take any (s_1, s_2) from Q
11 $p = detectPattern(s_1, s_2)$ (1)
12 **foreach** $pair(s'_1, s'_2) \in succPair_p(s_1, s_2)$ **do** (2)
13 **if** $s'_1 \in S_1 \vee s'_2 \in S_2$ **then**
14 **if** $smap(s'_1) \neq s'_2$ **then return** false
15 **else continue**
16 **if** p *is none* **then** $equal = cmpInst(s_1, s_2)$
17 **else** $equal = compare_p((s_1, s'_1), (s_2, s'_2))$ (3)
18
19 **if** $\neg equal$ **then return** false
`// Update synchronization sets and maps`
20 $S_1 = S_2 \cup \{s'_1\}, S_2 = S_2 \cup \{s'_2\}, smap(s'_1) = s'_2$
21 update *varmap* according to p (4)
22 insert (s'_1, s'_2) to Q (5)
23 **return** true

Algorithm 1: Checking semantic equivalence of functions

At the beginning of the whole algorithm we have to perform the already mentioned transformations and compare lists of parameters P_1 and P_2, if the parameter count of these two functions does differ, then they are considered semantically different. Next the algorithm maps global variables g_1 and g_2 from their sets G_1 and G_2 defined for functions f_1 and f_2, respectively. This mapping is done by using the variable name comparison.

After that comes the main body of the algorithm, for increased brevity, we are going to explain the algorithm on its simplified version, which has been presented in [13]. At the beginning of the comparison function arguments and global variables are synchronized in *varmap* and one synchronization point from each function is placed at the beginning of those functions and is initially synchronized in *smap*. Then, we perform the following

steps for each yet unvisited synchronized pair (steps are for brevity marked by according numbers in circles in the Algorithm 1):

1. **Pattern detection** – In this step, the algorithm decides, whether a built-in SPCP is applicable at this point. This step has to be really fast, as it could be potentially performed for each pair of synchronization points for each supported SPCP. More details about supported SPCPs are mentioned in Section 2.3.1.

2. **Determine successor synchronization pairs** – In the second step of the algorithm, we are finding a following pair of synchronized points, i.e., a point from where the analysis will continue after the currently analyzed pieces of code are compared as semantically equal. If an SPCP is aplicable, then it must define the set of the following synchronization pairs. Otherwise, the new points are placed right after the following instructions using the succesor functions *succ*, or *succT* and *succF*, for non-branching/unconditional branching and conditional branching, respectively.

3. **Semantic equality detection** – The third step of the analyzing algorithm checks whether pieces of code between the current and each of the following pairs of synchronization points are semantically equal. This is done either by using a pattern-specific comparison or by comparing semantics of single instructions (if no SPCP was applied). If we are checking the semantics of an instruction, then we are checking whether it performs the same operations over the same operands, where some of them may be mapped via *varmap*.

4. **Update variable mapping** – Instructions may introduce new local variables or perform some other action, that could potentially produce some output. In such a case, those corresponding outputs have to be re-syncronized in the *varmap*.

5. **Schedule following comparison** – In the end, the new synchronization points from Step 2 are scheduled for a subsequent comparison and the algorithm repeats itself from Step 1, until all synchronization pairs are analyzed.

A more detailed description of the algorithm can be found in [14].

2.3 Code Change Patterns

In the introduction we have mentioned, that DiffKemp uses built-in *semantic preserving change patterns (SPCPs)* to check for equivalence of refactored pieces of code. In [13], the concept of SPCPs has been extended to generic *code change patterns* (CCPs), which enable the user to define their own patterns that allow them to declare some custom changes as semantic preserving.

A CCP is a pattern of software modification, that reoccurs across different software projects [15] and generally could be viewed as a pair of code fragments whose input and output can be mapped together as described in [20]. Hence, these fragments describe two different ways to calculate the same output from the same input. In the context of DiffKemp there are two additional properties that CCPs should have: (1) one of them is the transformation of the other and (2) both of them should come from different version of the same program.

In the next subsections (and chapters), we are going to discuss the SPCPs and CCPs even further.

2.3.1 Built-in Code Change Patterns

DiffKemp can handle many refactoring patterns known from literature by default, these patterns come mostly from [3], which is to our knowledge the most exhaustive collection of refactoring patterns applied to low-level languages, such as C. This collection is enriched with some patterns that were found by experimental study of refactoring patterns in the Linux Kernel by the team behind DiffKemp:

- **Changes in structure data types** – It covers a set of changes that can be done in user-defined structures and unions. Such as an addition, removal or renaming of a field.
- **Splitting code into functions** – This happens when programmer moves parts of the original functions to some stand-alone function, which is then called in place of the original code fragment.
- **Changes in source code location** – This is a special case for Linux Kernel, as it has macros and built-in functions that allow one to report the file name and the line number of current code location. If the location of the called macro/function should change, the semantics stays the same.
- **Changes in enumeration values** – It may happen when new value is added in the middle of an enumeration type, after that the rest of values is shifted and gets different numerical values.
- **Inverse branching conditions** – It covers when a branching condition is inversed and underlying branches swapped.
- **Relocated code** – This situation may happen when a usually independent piece of code is relocated to a different part of a function.

Another well-known collection is famous Fowlers catalog [2], but it is focused on object-oriented languages, which are not the main concern of DiffKemp.

By default, some of the patterns from [3] are handled by DiffKemp out-of-the-box, i.e., no additional modifications were needed to support them. Another are handled by SPCPs and some fraction is not handled at all.

We will now take a closer look on how are these SPCPs implemented. First, we have to mention that DiffKemp supports so-called *effective* SPCPs, which are specified by providing four functions:

1. A test whether the SPCP is applicable at a given pair of locations in the compared program versions. Again, they are usually a pair of potentially semantic different synchronization points. This test should be done by a quick and efficient analysis, as we want to know whether the SPCP is applicable as soon as possible, because other three functions may be more complex and thus expensive.
2. A function that computes next code locations, which succeed the given instance of SPCP, that is the location from which the Algorithm 1 continues the analysis from.
3. A specification of conditions under which the SPCP does indeed preserve semantics. This method can use a more complex algorithm, as it is focused only on small parts of the whole analyzed functions, that are determined by functions (1) and (2).

4. A way to compute mapping between program variables of two program versions after the SPCP is considered applicable.

2.3.2 Custom Code Change Patterns

As of version `v0.4.0` DiffKemp officialy supports *custom change patterns* (CCPs), which enables users to define their own patterns that would be used during analysis to eliminate some potential findings.

They are so important for this work, that we're going to dedicate the whole next section to explain them further in depth.

Chapter 3

Current State of Custom Code Change Patterns

There are times in software development when we actually want to introduce some semantic changes to functions of an otherwise stable API. Such changes could be bug fixes or could be related to security, which are desirable, and development teams usually know about them. But as we have already stated, they do introduce semantic changes, because of that they could be analyzed by the DiffKemp as semantically different, and pollute its output. Therefore, some unknown changes could be lost in a set of known changes (again, that are shown by DiffKemp as semantically different). Searching for these unexpected changes could be time-consuming and cumbersome.

What even further complicates this situation is a fact that there is no single generic solution for this problem, as many changes are often project-specific. Hence, what is considered safe for one project could not be viewed as safe for the other. It could even be viewed as a downright unsafe change in dependence on the project and the context of the change.

Therefore, the project needed some way of specification of filtration of changes on per-project basis and in an extension of DiffKemp introduced in [13] they came up with a simple, yet novel and very scalable approach of handling such situations, usage of so-called *custom change patterns*, denoted as CCPs in Section 2.3.2. With this approach developers have the ability to specify which changes they wish to ignore, as they view them as safe.

In this chapter we are going to formally define CCPs in Section 3.1. Then in Section 3.2 we are going to explain how are CCPs encoded and present DiffKemp-specific metadata. And lastly in Section 3.3 we are going explain the process of CCPs matching with corresponding algorithms.

Algorithms, definitions of CCPs and their regarding information come from [13] and a bachelors thesis this paper was originally based upon [20].

3.1 Formal Definition

In order to increase the brevity of algorithms and to provide context to some of the concepts mentioned in this chapter we are now going to present formalisms and notation from [13].

A Pattern is internally represented with the help of *parametrized control-flow graph*. A parametrized CFG c is a triple:

$$c = (in, CFG, out)$$

where CFG is a control-flow graph (known from Section 2.1) that can be parametrized using *undefined* structure types and local variables *in*, which represent the types and input values of the CFG, respectively. Last is the set of output variables denote as *out* (these are accessible outside c). With respect to this it is possible to define a code change pattern as a tuple:

$$p = (c_o, c_n, imap, omap)$$

Here c_o and c_n represent parametrized CFGs of old and new code change, respectively. With that in mind let:

$$c_o = (in_o, CFG_o, out_o)$$
$$c_n = (in_n, CFG_n, out_n)$$

Then, *imap* can be defined as a mapping between in_o and in_n as $imap : in_o \leftrightarrow in_n$, where in_o and in_n represent inputs of parametrized CFGs c_o, c_n, that is expressing which values have to semantically equal to successfully match the pattern. Analogically, $omap : out_o \leftrightarrow out_n$ is mapping between output variables of mentioned CFGs, which is in this case expressing which variables of the compared program will be mapped after successful matching of the pattern.

This definition of CCPs allowed incorporation to the current comparison algorithm with ease, as generic implementation of pattern-specific operations is being used in algorithm mentioned in Section 3.3, introduced in [13].

3.2 Patterns Encoding and DiffKemp Metadata

Now, that we have formally defined CCPs we need some mean of encoding them in a format that DiffKemp could recognize. As its whole codebase is build on top of LLVM libraries and CCPs are represented as parametrized CFGs, then LLVM IR comes as a natural choice, because it is designed to represent CFGs (as mentioned in 2.1) and it would not introduce any additional dependencies.

Then, if we take the formal definition of a pattern p mentioned in the previous section we can express different parts of the pattern using several LLVM IR facilities (such as prefixes, types, functions and metadata):

- Function bodies – In these we capture the nature of the change. That is, we keep a sequence of instructions that are related to the change from the old and new version of the function. To distinguish between the two we can use name prefixes `diffkemp.old` and `diffkemp.new`, that allow us such distinction.
- The sets of input values in_o and in_n – These are encoded without introduction of any custom facilities. They are simply encoded using LLVM function parameters and their mapping is determined based on the parameters order.
- Type parameters – If the pattern has need to parametrize by type, then it may use custom type prefixed with `diffkemp.type`, which is then used in both sides of the pattern (that is c_o and c_n). Therefore, it is not necessary to explicitly encode the mapping of this custom type.

- The sets of output variables (out_o and out_n) and their mapping *omap* – They are represented using a special function `diffkemp.mapping` which is called in each pattern function just before its exit. Such call contains a list of variables (again, representing out_o and out_n) and their mapping is determined automatically based on their order.
- Any additional information needed for pattern-matching algorithm (which can be found in Section 3.3.1) is encoded using custom LLVM metadata (more details about metadata can be found in Section 2.1 or in [6]), which are further explained in Table 3.2.

With these in mind we are now able to encode the change pattern (that is, a pair of new and old function CFG) to the LLVM IR. An example of such pattern can be seen in Figure 3.1. A summary of custom LLVM facilities can be seen in the Table 3.1.

LLVM Prefixes, types and functions	**Semantics**
`diffkemp.old`	Identifies that a function belongs to the old side of the pattern.
`diffkemp.new`	Identifies that a function belongs to the new side of the pattern.
`diffkemp.type`	Denotes a custom type, that removes the need for explicit encoding of the mapping, because it is used in both sides of the pattern.
`diffkemp.output_mapping`	Denotes an output of a pattern function

Table 3.1: Overview of custom LLVM IR prefixes used in pattern definition

As has been already mentioned, Table 3.1 summarizes custom LLVM IR facilities that can be used in process of encoding of the custom change pattern. Now we are going to show most of them in their usage on an example Figure 3.1. A usage of prefixes `diffkemp.old` and `diffkemp.new` can be seen on Lines 8 and 19, and on Lines 13 and 23. In the latter case their usage is slighty different, they still denote difference between the new and old version of a function, but they also associate different functions (declared on Lines 5-6) with their corresponding function contexts. That is, we want to make sure, that for example `@diffkemp.old.add` is going to be matched with some function `add()` from the context of `@diffkemp.old.twice` and not accidentally with some other function (with the same signature) from other context. Last example is the mapping `diffkemp.output_mapping`, which marks sets of output variables and their mapping *omap*, which is determined automatically based on their order. Its usage is explained in higher detail in 3.3.

The last piece of DiffKemp specific resources are custom metadata from Table 3.2. Their whole purpose is optimization and sometimes alteration of the CCP applicability. A usage of most of them is going to be explained in the next section.

In Figure 3.1 below, we can see an example of a CCP, where we have two versions of the same function called `twice`. The difference between these two is the call to a different function with different name and count of arguments. At the first glance we can see that many names are prefixed by `@diffkemp` and some instructions even have some metadata attached to them.

Metadata kind	Semantics
`pattern-start`	Marks the first pair of differing instruction.
`pattern-end`	Marks the end of the pattern main body. No other kind of markings are allowed after this metadata kind, but code fragment output and its mapping.
`group-start`	Marks start of grouped instructions, which have to be matched as a single block, hence no additional instructions are allowed between them.
`group-end`	Marks the end of a currently active instruction group.
`disable-name-comparison`	Disables name-based comparison of structures, replacing it with type-based equality comparison.

Table 3.2: Overview of custom LLVM IR Metadata used in pattern definition

3.3 Pattern Matching Algorithm

In this section we are going to present the pattern matching algorithm. To simplify its presentation (and also presentation of used supporting algorithms) we are going to assume the following:

- two versions of function f are denoted as f_o and f_n (representing the old and new version, respectively). We are also assuming that f_x means either f_o or f_n,
- functions f_x are represented using CFGs (which is described in Section 2.1) and it is referred to them as *compared function CFGs*,
- at some point, the comparison algorithm works with a pair of synchronization points s_o and s_n known from Algorithm 1.

With that taken into account in order to check, whether a pattern is applicable, we have to check if cfg_o and cfg_n are sub-graphs of f_o and f_n, respectively. This is a question of detection of sub-graph isomorphism, which is known to be generally very expensive operation, as such problem is labeled as NP-complete[1]. But fortunately we are not dealing in this case with some generic directed graph but with CFGs, which are a special kind of directed graph and we can, therefore, assume that:

1. Each CFG has a single entry point.
2. That we need to match pattern CFG (cfg_o and cfg_n) starting from the current synchronization point in the compared function CFG (f_o and f_n).

But that is only the first part of the problem, we also have to check that inputs and outputs defined for the pattern match corresponding inputs and outputs of compared functions. Therefore, we have to check applicability of the pattern on multiple levels.

We are going to use a top-down approach to explain the matching algorithms presented in [13] and [20]. First, we are going to explain the top-level algorithm in Subsection 3.3.1

[1] „A problem is assigned to the NP (nondeterministic polynomial time) class if it is solvable in polynomial time by a nondeterministic Turing machine.“[21]

```
!0 = !{ !"pattern-start" }
!1 = !{ !"pattern-end" }

declare void @diffkemp.output_mapping(...)
declare i32 @diffkemp.old.add(i32, i32)
declare i32 @diffkemp.new.dbl_it(i32)

define void @diffkemp.old.twice(i32 %0) {
  %2 = alloca i32, align 4
  store i32 %0, ptr %2, align 4
  %3 = load i32, ptr %2, align 4
  %4 = load i32, ptr %2, align 4 !diffkemp.pattern !0
  %5 = call i32 @diffkemp.old.add(i32 %3, i32 %4)
  %6 = add nsw i32 %5, 4
  call (...) @diffkemp.output_mapping(%6)
  ret void !diffkemp.pattern !1
}

define void @diffkemp.new.twice(i32 %0) {
  %1 = alloca i32, align 4
  store i32 %0, ptr %1, align 4
  %2 = load i32, ptr %1, align 4
  %4 = call i32 @diffkemp.new.dbl_it(i32 %2) !diffkemp.pattern !0
  %5 = add nsw i32 %4, 4
  call (...) @diffkemp.output_mapping(%5)
  ret void !diffkemp.pattern !1
}
```

Figure 3.1: Example of custom change pattern

and then in the subsequent subsections we are going to explain several functions used in the algorithm. Namely *matchCFG* for sub-graph matching in Subsection 3.3.2, *cmpInst* for instruction comparison in Subsection 3.3.3.

3.3.1 Top-Level Algorithm

In the case that two instructions become unsynchronized (i.e., they could potentially not be semantically equal) in Algorithm 1 then DiffKemp is checking for applicability of SPCPs and CCPs. In the latter case we have to check if any pattern P in set of predefined patterns is applicable. That is determined by the Algorithm 2, which we are now going to cover in further detail:

Input : (i_o, i_n) : a pair of differing instructions
smap and *varmap*: mapping functions from Algorithm 1
P: set of available instruction patterns
Output: A set of matched instructions if any pattern matched, $\emptyset$ otherwise

```
1  foreach (c_o, c_n, imap, omap) ∈ P do
2      match_o = matchCFG(i_o, c_o)
3      match_n = matchCFG(i_n, c_n)
       // Check whether any map is empty
4      if ¬match_o ∨ ¬match_n then continue
5      valid = true
6      foreach (i_o, i_n) ∈ imap do
7          if ValidityPredicate(match_o, match_n, i_o, i_n) then (1)
8              valid = false
9      if ¬valid then continue
10     foreach (o_o, o_n) ∈ omap do (3)
11         varmap(match_o(o_o)) = match_n(o_n)
12     return match_o ∪ match_n
13 return ∅
```

Algorithm 2: Definition of the *cmpInst* function from [20]

The first step is the mentioned sub-graph matching for old and new version of the program (Lines 2-3). The algorithm for sub-graph matching can be found in 3.3.2.

Then, if the matching was successful, we need to check whether values and types, that were matched to the inputs of the pattern CFGs, have the same semantics in both compared functions (marked as number 1). Hence, we need to check if they are semantically equal.

This can be done by using pattern matching functions, particurarly *imap*, which can be known from formal definition of patterns in Subsection 3.1, and maps of variable and types of pattern CFGs to variables and types of compared-function CFGs $match_o$ and $match_n$. With these mappings we then check if following predicate stands (note that we pressume that *varmap* is available together with function that returns a type name of an instruction called *typename*):

$$\begin{aligned} &ValidityPredicate(match_o, match_n, i_o, i_n) = \\ &\quad (match_o(i_o) = match_n(i_n)) \ \vee \\ &\quad (varmap(match_o(i_o)) = match_n(i_n)) \ \vee \\ &\quad (typename(match_o(i_o)) = typename(match_n(i_n)) \end{aligned}$$

Semantic equivalence differs in dependence to the kind of compared value: (1) constants are compared by value, (2) variables are compared using *varmap*, and (3) structure types are compared by the type name.

At last, it is needed to determined which variables created by the pattern have the same semantics for the following comparison (by the Algorithm 1). Therefore, we have to propagate that some functions, that had previously been marked as non-equal are now equal/ignored thanks to the CCPs. This is done by updating the *varmap* function with values from *omap* (again from 3.1 using mappings from mentioned $match_o$ and $match_n$. In the algorithm it is marked as number 3.

3.3.2 Sub-Graph Matching

An important step in the whole pattern applicability algorithm is the sub-graph matching. In this Subsection we are going to explain how it works in the case of DiffKemp CCPs with Algorithm 3 which is run separately for both versions of the compared program. If matching succeeds, then the pattern is deemed applicable.

Input : $c_x = (in_x, cfg_x, out_x)$: pattern CFG
f_x: compared function CFG
s_x: current synchronization point in f_x
Output: $match_x$: mapping between values and types of cfg_x and f_x

```
1  e_p = first differing instruction in cfg_x
2  e_f = instruction immediately following s_x in f_x
3  Q = {(e_p, e_x)}
4  match_x = {}
5  while Q is not empty do
6      take any (i_p, i_f) from Q
7      if ¬cmpInst(i_p, i_f, in_x) then // updates match_x
8
9          if succ (i_f) is defined then
10             add (i_p, succ(i_f)) to Q // instruction skipping
11             continue
12         else return ∅
13     if i_p is conditional branch then (1)
14         add (succT(i_p), succT(i_f)) to Q
15         add (succF(i_p), succF(i_f)) to Q
16     else add (succ(i_p), succ(i_f)) to Q
17 if ¬checkContext(match_x, ctx_x, in_x) then (2)
18     return ∅
19 return match_x
```

Algorithm 3: Matching pattern CFG to one of the compared functions from [13]

In its core, the algorithm simply traverses the control-flow of cfg_x and f_x (i.e. the pattern CFG and compared-function CFG) starting from the first differing instruction in cfg_x and from the instruction immediately following the current synchronization point s_x in f_x. This instruction pair is then compared using $cmpInst_p$ (from Al. 4) function and if they don't match, then it is allowed to skip instructions in f_x, but only in the case that such instructions have only single successor (skipping of conditional branching instructions is not allowed).

In the step marked as number 1 (Lines 13-16) algorithm determines where the analysis continues from, if the current change chunk comparison succeeded. In case of CCPs it should be the next instruction i of the compared functions, which has not yet been matched to the pattern CFG, but which is immediately following some instruction, that have already been matched. However, there may occur situations, when there is multiple of said instructions, due to two reasons:

1. As it has already been mentioned in Section 2.1, CFGs are not required to have only single exit point (as can be seen in Figure 2.2a), because of that the algorithm may be required to analyze multiple basic blocks.

2. It is allowed for CCPs to skip some instructions in the compared function CFG, which have to be compared, after the pattern is successfully matched.

Because of these two reasons, there could be numerous instructions to continue from. This is handled by a simple limitation. New synchronization point may be placed at each instruction i only if there is no other synchronization point placed at instruction i', from which i would be reachable. Such operation is safe, because otherwise i would eventually be analyzed as the default comparison method follows the natural control flow.

Then at the line marked by a circle number 2 we have to check if context of the pattern and compared function is still valid. More information about a context can be found in [13].

3.3.3 Instruction comparison

We need to check, whether two instructions are equal in the Algorithm 3, but this check is slightly different that the one presented in Algorithm 1. Functionality of this check is going to be further explained by Algorithm 4.

Input : $i_p : v_p = op_p(o_p^1, ..., o_p^m)$: pattern instruction
$i_f : v_f = op_f(o_f^1, ..., o_f^n)$: compared-function instruction
in_x: pattern inputs
ctx_x: the se of pattern context variables
Output: *true* if i_p matches i_f, *false* otherwise

```
1  if op_p ≠ op_f then return false  // ensures m = n
2
3  for 1 ≤ i ≤ n do
4      if typeof(o_p^i) ∈ in_x then (1)
5          add (typeof(o_p^i), typeof(o_f^i)) to match_x
6      else if typeof(o_p^i) ≠ typeof(o_f^i) then (2)
7          return false
8      if o_p^i ∈ in_x ∨ o_p^i ∈ ctx_x then (3)
9          add (o_p^i, o_f^i) to match_x
10     else if ¬(o_p^i = o_f^i ∨ name(o_p^i) ≈ name(o_f^i) ∨ match_x(o_p^i) = o_f^i) then (4)
11         return false
12 match_x(v_p) = v_f
13 return true
```

Algorithm 4: $cmpInst_p$: Comparison of pattern and compared-function instructions

To check whether instructions are semantically equal (i.e., they perform the same operation over semantically equivalent operands, which means that these operands have matching types and values). This is done in multiple checks, whose numbers correspond to the numbers in the above algorithm listing:

1. In the case that pattern instruction operation is a part of the pattern input, then new matching is created. This matching then contains types of the pattern and that of the compared function, respectively. Hence, compared function types are marked, if they are mapped to which input types of the pattern CFG.

2. If the types do not match and one of them is not an input, then operands are considered as semantically different.

3. Similar check to the first one, but in this case values now may be part of the pattern context.

4. Last check consists of several parts (denoted as individual disjuncts). First part is similar to the check number 2 for the operand values, but for values, the equality check is more complex than for types and it depends on the operand kind. Constants are checked for direct equality. Functions and global variables are checked for name match that is not made purely by name equality, but patterns are also allowed to specify *renaming rules*, which allow to specify different names between the versions (this is expressed by the $\approx$ in the algorithm). At last, local variables are checked if the values have already been mapped via $match_x$.

Chapter 4

Automatic Code Change Patterns Generation

The goal of this thesis is to provide a simple way of automatic inference and generation of CCPs. Even through their usage is quite easy, their creation can get difficult, error-prone and time-consuming. As it is written in the LLVM IR language, which is the form of encoding that DiffKemp is using for CCPs. But LLVM IR was not designed with programmers ergonomics in mind, because it is supposed to be generated with the help of LLVM libraries and not to be written by hand. From this sprang an idea, that CCPs could be automatically generated from selected diff chunks provided by the user. In the following text we are going to present our design of the proposed extension to DiffKemp that should provide automatic inference and generation of CCPs.

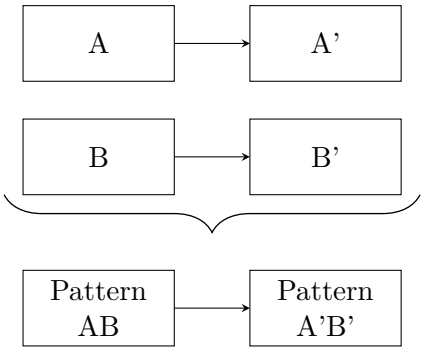


Figure 4.1: An example of pattern generation

Figure 4.1 summarizes our goal we want to take two different changes between two versions of some compare program, in this case $A \to A'$ and $B \to B'$, and create some pattern $AB \to A'B'$ which generalizes changes done in those two specific changes and allows us to ignore them (or other similar changes) in another run of DiffKemp.

Our proposed solution of automatic generation can be split into three phases:

1. **Base pattern inference** – The first phase tries to find the so-called *base pattern*, which is then going to be used to provide a foundation for further generation, this algorithm is explained in-depth in Section 4.3.

2. **Type and global variable variation** – Using the base pattern from the previous phase, we can achieve parametrization of various structure types and global values by

mutating selected instructions from the base pattern and creating new derivates from it. More about it can be found in Section 4.4.

3. **Pattern range determination** – After we have all variants available, we can determine the pattern range (i.e. the range of instructions that are different for the *new* and the *old* side of the pattern) and mark it with metadata from Table 3.2. This algorithm is introduced in Section 4.5.

Definitions of the concepts and their notation are going to be presented in Section 4.2. In the next section we are going to discuss previous work done in the field of automatic inference of semantic change patterns and patches together with the main differences with our proposed method and use-case.

4.1 Current State of Automatic Pattern and Patch Inference

One of the most common development tasks besides development of new features are refactoring and various fixes. In the former case we are speaking about changing the structure of a program without changing its behavior and in the latter case it may be a security fix, bug fix or some change that has been requested by an investors. There have been various works that research this part of computer science in which they are trying to find some generic method that could be used to infer and generate so-called *semantic patches*, which have been introduced in [18] and it has been described as a generalization of standard patches (which describe a transformation of software in terms of addition and removal on corresponding lines) and because they are generalized, then they should by definition cover more than one of these patches. Some of these works focus on automatic inference of code transforms that could then be applied to patch generation for some distributed code revision software examples of such project could be Genesis [10] or Par [4] There are even works that use *Genetic programming* for successful (albeit limited) inference of repairs or patches, e.g. GenProg [5]. However, most of these work do not really cover our needs for pattern inference because:

1. They usually concentrate on the problem of automatic patch inference, be it for from a bug fix or some common refactoring across a project.

2. Their goal is to find the most generic patch, that would semantically match as many patches as possible, in a contrary our goal is to find the most concrete pattern that would filter out all of its input *candidate differences* (which are explained in the next section).

3. Usage of their generated derivates is focused on automatic application of fixes and refactorings. Moreover, such semantic patches are mostly directed towards being project-independent (There are even works that specify in data mining with the goal of finding common refactoring and change patterns between projects, e.g. Common [16] or CPatMiner [17]). Again, in the contrary to our method, we are trying to provide support for project-specific patterns and all the data we would need should be provided by the user. But it is possible that DiffKemp could be extended with some system that could suggest difference candidates in the future.

At the moment of writing this thesis there is one not yet fully merged extension to DiffKemp, which introduces basic generalization features to CCPs. This, together with a

fact, that there is currently no support on how to correctly parametrize a CCP by instructions, or even whole basic blocks, limits our work to parametrization by type and a value without much generality. We are, for example, capable of expressing any integer value, but we are not capable of expressing any global value, that is partially solved by so-called *pattern variation*, which is going to be discussed further in Section 4.4.

4.2 Formal Definitions for Automatic Pattern Generation

Finding a so-called *base pattern* is a question of gradual generalization of a pattern using parametrized values. Each time a new, value with the same usage (which means that it is used in the same code locations in the same way) is encountered, then it is parametrized and hence the base pattern is changed. Intermediate product of this generalization is called *pattern candidate* which then becomes the base pattern after successful generalization of all given *candidate differences*.

Once we have this base pattern, we can try to create its variants, which represent usage of different types and global variables. Every time, instructions differ in their type of global usage, it is noted for the future variant generation.

First we have to find so-called *base pattern* which is the most generic representation of common parts of all the candidates differences. Formally, base pattern is a triple:

$$P_B = (cfg_o^P, cfg_n^P, V) \tag{4.1}$$

where cfg_o^P and cfg_n^P are parametrized CFGs representing functions from the new and the old version respectively. Then there is V which is a set of pattern variants. A pattern variant is a variation of several instructions in the base pattern, that are parametrized by a type or a global value. Hence, it is a set of instruction variants. Single instruction variant i_v could be denoted as substitution:

$$i_v = i_p[o_n/o_v]. \tag{4.2}$$

Here, i_p is the original instruction from the base pattern P_B and o_n is an operand of instruction i_p where n is its order. This operand o_n is to be substituted by the variant operand o_v (again, which should be some different type or global variable).

As have been already mentioned, candidate differences are the basic building block of inferred patterns. Formally, candidate difference c is a pair:

$$c = (cfg_o, cfg_n) \tag{4.3}$$

where cfg_o and cfg_n represent the difference between two versions. All of the above definitions allow us to better explain algorithms presented in the rest of the chapter. In the next section, we are going to introduce the top-level algorithm.

4.3 Top-Level Inference Algorithm

At the start of this chapter, we have presented a simple overview of the top-level algorithm. Now, we are going to present it as an Algorithm 5 and explain it more thoroughly.

Input : C_p : a set of candidate differences
Output: A set of generated patterns S_p if their inference was successful, $\emptyset$ otherwise

```
1  S_p = ∅
2  initialize pattern candidate P_c
3  foreach (cfg_o, cfg_n) ∈ C_p do
4      success = addFunctionPair(P_c, cfg_o, cfg_n)
5      if ¬success then
6          report failure
7          skip to the next pattern
8  P_B = P_c
9  get V out of P_B
10 foreach v ∈ V do
11     P_v = generateVariant(P_B, v)
12     determinePatternRange(P_v)
13     add P_v to S_p
14 return S_p
```

Algorithm 5: Top-level generation algorithm

First, we have to get the basic information about the pattern from the configuration file C_p (structure of this file is presented in Chapter 5). This configuration file contains all paths to all cfg_o and cfg_n pairs, that represent old and new version of a function from which we are going to try to generate the pattern later in the algorithm, respectively.

We also have to initialize pattern candidate P_c (Line 2), as was briefly mentioned in the previous section, and an empty generated pattern set S_p. Pattern candidate is the intermediate product of pattern inference, we are simply adding candidate differences to the P_c to make it even more generic.

It is done by function *addFunctionPair* which is going to be explained later in this section. If the inference wasn't successful, then we are reporting the user and moving to another pattern, as the differences the user provided were incompatible. In our case that means, that there was some conflicting instruction between cfg_o^P (from P_c) and cfg_o, or between cfg_n^P (again, from P_c) and cfg_n. Otherwise, we are adding each function pair defined for the pattern P to the P_c (Lines 3-7).

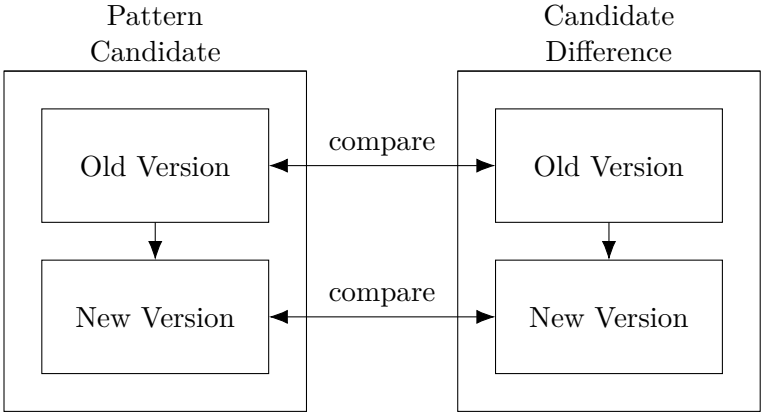


Figure 4.2: Comparison between pattern candidate and candidate difference

The concept of comparison of functions is better illustrated by the following Figure 4.2. The difference between this comparison and the one explained in Algorithm 1 is that we are not comparing new and old version of a function, but rather comparing two old versions and two new versions of two separate functions, and we are trying to find what they have in common and what could possibly be parametrized.

If everything have been successful, then the pattern candidate P_c becomes the pattern base P_B (Line 8).

After that we can take the set of variants V from the recently assigned P_B. Each variant $v \in V$ contains a set of instruction variants, which were generated from each candidate difference (although some of these sets may be empty, as it is not a rule that each difference should have some instruction variants). This set is then applied on P_B which creates new variant of the base pattern denoted as P_v. Hence, P_v consists of the instructions as P_B, but operands of some of those instruction use different global variables or types. Example of such variant, together with the algorithm behind it can be found in Section 4.4. After that we can take the set of pattern variants from the recently assigned P_B and apply them with function *applyVariant* from Section 4.4. For each variant we are also going to determine the pattern range (that is the *pattern-start* and *pattern-end* metadata pair, that allows pattern matching optimization) with function *determinePatternRange* from Section 4.5. Each applied variant is then added to the set of generated patterns S_p. Which is going to be returned at the end of the algorithm, as is mentioned in the algorithm header it may return an empty set which means that no pattern was successfully generated.

In the rest of this section we are going to explain supporting algorithms listed in the Algorithm 5. Starting with a function that adds a function pair to a pattern candidate in Algorithm 6.

Input : $P_c = (cfg_o^c, cfg_n^c, V)$: Pattern candidate
cfg_o : function on the old side of the change
cfg_n : function on the new side of the change
Output: *true* if the inference was a success, *false* otherwise

1 **if** P_c *is not yet assigned* **then**
2 $P_c = (cfg_o, cfg_n, \emptyset)$
3 **return** *true*
`// Both functions update` P_c
4 $(S_1, v_o) = addFunction(f_o, cfg_o)$
5 $(S_2, v_n) = addFunction(f_n, cfg_n)$
`// Reassignement of` P_c `values`
6 $P_c = (f_n, f_o, V \cup (v_o \cup v_n))$
7 **return** $(S_1 \wedge S_2)$

Algorithm 6: *addFunctionPair*: adds functions for generalization to a pattern

At the beginning of the algorithm the input pattern P_c may be in two states, when it is provided to this function:

1. The pattern is empty and this is the first function pair, that is going to be added to the candidate. In that case the functions represented as cfg_o and cfg_n are simply cloned to the pattern and we not assigning any variants (as there are not any). This is because our method is trying to find the most concrete pattern possible and if there would be a pattern in the configuration file specified simply as one old version and

one new version of a function, then this pair is the most concrete pattern that we can find, therefore, we do not have to generalize any of its parts (Lines 1-3).

2. This is not the first time this function has been called for this pattern P_c and therefore, we have to generalize cfg_o and cfg_n with cfg_o^c and cfg_n^c, respectively. A set of pattern variants associated with this pattern V is then merged with a new pattern variant created by the conjunction of sets of instruction variants v_o for the old pattern side and v_n for the new pattern side (Lines 4-6).

In the latter case we are calling the function *addFunction*, which is going to be explained in Algorithm 7 below. This function returns a tuple of boolean value (which indicates whether the inference was successful), and a set of variants, which have been found in during the inference (if the inference was unsuccessful or we haven't found any instruction variant, then it may return an empty set).

Input : cfg_1 : pattern candidate function CFG
cfg_2 : candidate difference CFG
Output: A tuple of boolean value, whether the inference was successful, and set of variant instruction, which may be empty

```
   // initialization of a set of instruction variants
1  V = ∅
   // Let i_1^1 and i_2^1 be the first instructions in cfg_1 and cfg_2,
   respectively.
2  Q = {(i_1^1, i_2^1)}
3  while Q ≠ ∅ do
4      take any pair (i_1, i_2) from Q
5      foreach (s_1, s_2) ∈ succPair(i_1, i_2)) do
6          insert (s_1, s_2) into Q
7      if cmpInst(i_1, i_2) then
8          continue
9      if i_1 and i_2 perform different operations then
10         return (false, ∅)
11     for o_n^1 ∈ i_1 and o_n^2 ∈ i_2, where n ∈ count of i_1 operands do
12         if o_n^1 ≠ o_n^2 then
13             if value(o_n^1) ∈ pattern input then
14                 continue
15             if value(o_n^1) is global ∨ value(o_n^2) is global then
16                 add i_1[o_n^1/o_n^2] to V
17             else
18                 parametrize the value in cfg_1
19         else if typeof(o_n^1) ≠ typeof(o_n^2) then
20             add i_1[typeof(o_n^1)/typeof(o_n^2)] to V
21 return (true, V)
```

Algorithm 7: *addFunction*: add function to generalization

In this algorithm we are simply traversing the control flow of two functions (one is denoted as cfg_1 and the other as cfg_2). We expect this CFGs to have the same control-flow (that is, the same composition of basic blocks) , otherwise the inference would be unsuccessful (as can be seen on Line 10), that is because of two things: (1) we are not

able to parametrize different instructions and (2) as showed in Figure 4.2 we should be comparing old version with old version and vice versa.

This comparison is done by the function *cmpInst* listed in Algorithm 1, if the instructions i_1 and i_2 are equal, then they are not interesting for the pattern inference and they may be skipped, as there is nothing to be parametrized (Lines 7-8).

We think that it is important to mention why our proposed solution checks if the instruction are the same (Line 9) and after that we are checking equivalence of instruction operands. That is because the *cmpInst* only informs us, that there is *some difference*, between the two instructions, but it does not tell us what is the nature of the difference. Possible differences range from conflicting values and types to conflicting instructions, all but the differing instructions are parameterizable . From now on, when we are mentioning difference in operation, we are speaking about difference in values or types and not differing operations, as such difference means unsuccessful inference.

Hence, in a case where two instructions were not the same then we have to check the nature of the difference between operands o_n^1 and o_n^2 from instructions i_1 and i_2, respectively. There are basically two possible states:

1. The difference is in a value, which means that we should be able to parametrize it. It may happen that the value is already parametrized in such a case the process may get quite complex, because of that it is going to be explained in more detail by an example after this listing. But there is a chance that the value happens to be a global value, then the operand is used to create a new instruction variant (Lines 15-16). Otherwise, if the value is nor parametrized nor global value, then it will be parametrized (Line 18).

2. The difference is in a type. Therefore, we are creating an instruction variant that should replace the type of the original operand o_1 with $typeof(o_2)$ (Lines 19-20).

As mentioned in the listing above, parametrizing may get complex, if one of the values is already parametrized, therefore we have decided to provide an example in form of Figure 4.3. In this example all functions are on the same side of the pattern (be it an old or a new side), and as with other examples of LLVM IR, they are heavily simplified to only contain instructions relevant to our example. At the beginning of the inference we have two function f_1 and f_2 from Figure 4.3a and Figure 4.3b, respectively. Their inference product would be our first pattern candidate in Figure 4.3c, which is now parametrized by parameter `%1` as both candidate differences use the same values (in that functions scope, they may differ between candidate differences) in the same way (that is `42` in the case of f_1 and `1` in the case of f_2). To this intermediate product we are going to add function f_3 from Figure 4.3d which uses different values, but they are of the same type (in this case it consists of two different values `2` and `3`). Therefore, we are going to add another parameter, that allows parametrization of these values, the algorithm then denotes this new parameter as `%2`. Even through we now have two parameters of the pattern, we are still able to express the same values, because both parameters have the same type.

```
define i32 @f1(i32 %0) {
  %4 = add nsw i32 %3, 42
  %6 = add nsw i32 %5, 42
}
```

(a) First candidate difference

```
define i32 @f2(i32 %0) {
  %4 = add nsw i32 %3, 1
  %6 = add nsw i32 %5, 1
}
```

(b) Second candidate difference

```
define i32 @pf1(i32 %0, i32 %1) {
  %5 = add nsw i32 %3, %1
  %7 = add nsw i32 %5, %1
}
```

(c) First pattern candidate from (a) and (b)

```
define i32 @f3(i32 %0) {
  %4 = add nsw i32 %3, 2
  %6 = add nsw i32 %5, 3
}
```

(d) Third candidate difference

```
define i32 @pf2(i32 %0, i32 %1, i32 %2) {
  %6 = add nsw i32 %3, %1
  %8 = add nsw i32 %5, %2
}
```

(e) Base pattern from the previous candidate and third difference

Figure 4.3: Illustrated process of pattern inference

4.4 Pattern Variation

Now, we can explain the second phase of the proposed method in detail. Pattern variance is a process of creating patterns that differ from the base pattern in types or usage of global values. That is done because there is no other simple generalization feature, that we could use (e.g., some substitution that could express any structure type or any global variable name). Because of this lacking feature we propose, that we generate a new pattern with different variants.

The Algorithm 8 consists of three main steps:

1. Copy the base pattern, as we are going to substitute some of its instruction operands in the copy P_v (Line 1).

2. Find the instruction, that should be substituted and substitute its operand o_p with o_v from instruction variant (Lines 3-7).

3. Map the return values of function in P_v. Patterns in DiffKemp are defined with output mapping function *omap*. In order to add the output of a function to the *omap* we have to wrap the original return value to a call to $@diffkemp.mapping$ function (Line 8), which are further explained in Chapter 3.

Input : $P_B = (cfg_o^P, cfg_n^P)$: pattern function
V : A set of instruction variants
Output: A variant of the base pattern P_v

P_v = deep copy of P_B
get (cfg_o^v, cfg_n^v) from P_v
for $f \in \{cfg_o^v, cfg_n^v\}$ **do**
    foreach $i_p[o_n/o_v] \in V$ **do**
        foreach $i \in f$ **do**
            if $i = i_p$ **then**
                substitute o_p with o_v
map return values of f_o^v and f_n^v
return P_v

Algorithm 8: *applyVariant*: apply pattern variant to base pattern

If all of the above steps are done, then the algorithm returns the newly created pattern variant.

```c
int glb1 = 1;
int f(int x) {
    x += glb1;
    return x + 42;
}
```

(a) First C code snippet

```c
int glb2 = 1;
int f(int x) {
    x += glb2;
    return x + 7;
}
```

(b) Second C code snippet

```
@glb1 = global i32 1
define i32 @f(i32 %0) {
  %3 = load i32, ptr @glb1
  %7 = add nsw i32 %6, 42
  ret i32 %7
}
```

(c) Simplified LLVM IR of (a)

```
@glb2 = global i32 1
define i32 @f(i32 %0) {
  %3 = load i32, ptr @glb2
  %7 = add nsw i32 %6, 7
  ret i32 %7
}
```

(d) Simplified LLVM IR of (b)

```
@glb1 = global i32 1
define i32 @f(i32 %0, i32 %1) {
  %4 = load i32, ptr @glb1
  %8 = add nsw i32 %6, %1

  ret i32 %8
}
```

(e) Base pattern from (c) and (d)

```
@glb2 = global i32 1
define void @f(i32 %0, i32 %1) {
  %4 = load i32, ptr @glb2
  %8 = add nsw i32 %6, %1
  call @diffkemp.mapping(i32 %8)
  ret void
}
```

(f) Variant of (e) for (d)

Figure 4.4: Illustrated process of base pattern inference and consequent pattern variation

In the Figure 4.4 we can see how pattern variation continues from the generation. We have two C code snippets (a) and (b) which for the sake of this example are on the same side of the change, be it old or new, and they are passed to the generator as candidate differences. Their generated LLVM IR can be seen in subfigures (c) and (d), these code fragments were greatly reduces for the sake of brevity and we have only included instructions, that are relevant for the inference and variation, this is not en example of valid LLVM IR. From these differences is then generated base pattern which can be seen in subfigure (e) by Algorithm 5. In the subfigure (f) we can see pattern variant which substituted global variable $glb1$ with $glb2$ in the `load` instruction. Thus, in the end we have the base pattern as one resulting generated pattern and one generated variant.

4.5 Pattern Range Determination

The last step of the top-level algorithm (Al. 5) is the determination of the pattern range. First, we have to explain what even pattern range is, it is a pair of metadata in both cfg_o^v and cfg_n^v (which are variants of functions from the base pattern P_B), that is used for optimization during pattern matching It has two parts and in both cases it is denoted by a metadata from Section 3.2:

1. **Start of the pattern** – It denotes the first differing instruction between the old and new side of the pattern. It allows faster matching as we only have to match from this denoted instruction. There can be only one such instruction in a pattern.

2. **End of the pattern** – This metadata denote an end of the differing instruction. For convenience, we denote the return instructions as the end of the pattern. In contrary to the start, there can be more than one instruction denoted as the end of a pattern. It is even desirable, as we have to provide an ending to a pattern in each of the control flows that end separately. If we take the CFG from Figure 2.2a, then both basic block (B) and (D) should have its own instance of ending metadata.

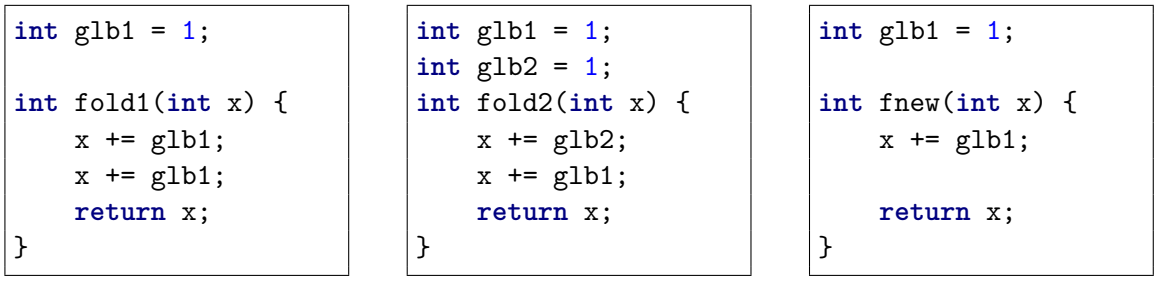
We think that the algorithm is simple enough that we do not have to provide it in form of pseudocode as that would probably be more confusing. Generally, the algorithm can be split into two phases:

1. Find the start of the pattern range, that is done by finding the first differing instruction between cfg_o^v and cfg_n^v. To this instruction we then attach the corresponding metadata and break from the nested loops.

2. Find all `ret` instructions in both cfg_o^v and cfg_n^v, and attach the *pattern-end* metadata to them.

Our proposed method determines this range after all other generation and variation is already done. This might raise some questions as to why we haven't decided to determine this range for the base pattern and then copy it to variants. This is because the start of a pattern variant may change in dependence to the instruction variants, consider this example in Figure 4.5.

Where we have three functions in language C, first two $fold1$ and $fold2$ are from the old version of a program and in the new version they both have changed to the function $fnew1$. As they both differ in the usage of global variables, this pattern is going to be a subject of



```c
int glb1 = 1;

int fold1(int x) {
    x += glb1;
    x += glb1;
    return x;
}
```

```c
int glb1 = 1;
int glb2 = 1;
int fold2(int x) {
    x += glb2;
    x += glb1;
    return x;
}
```

```c
int glb1 = 1;

int fnew(int x) {
    x += glb1;

    return x;
}
```

Figure 4.5: Code fragments in C from which we want to create a pattern.

pattern variation, and here comes the reason why we have to determine the ranges for each variant individually. Which can be seen in Figure 4.6 as there is a different first differing instruction between $fold1$ and $fnew1$, and between $fold2$ and $fnew1$. Therefore, if we would simply copy the precomputed range from the base pattern, then that range would be wrong.

```
@glb1 = global i32 1

define void @fold1(i32 %0) {
  %3 = load i32, ptr @glb1
  %6 = load i32, ptr @glb1 !0
  %11 = add nsw i32 %9, %10
  call @diffkemp.mapping(i32 %11)
  ret void !1
}
```

(a) Simplified LLVM IR for $fold1$ with determined ranges.

```
@glb1 = global i32 1
@glb2 = global i32 1
define void @fold2(i32 %0) {
  %3 = load i32, ptr @glb2 !0
  %6 = load i32, ptr @glb1
  %11 = add nsw i32 %9, %10
  call @diffkemp.mapping(i32 %11)
  ret void !1
}
```

(b) Simplified LLVM IR for $fold2$ with determined ranges.

```
@glb1 = global i32 1
define void @fnew(i32 %0) {
  %3 = load i32, ptr @glb1 !0
  %6 = add nsw i32 %9, %10
  call (...) @diffkemp.mapping(i32 %11)
  ret void !1
}
```

(c) Variant of (e) for (d)

Figure 4.6: An example of different pattern ranges in dependence to pattern variant

Chapter 5

Implementation of Pattern Generation Extension

In this chapter we are going to describe the most important implementation details of the proposed pattern generation extension from the previous chapter. The core functionality of the extension was implemented in programming language C++ as it is the language of choice used in the core of DiffKemp, together with a small part written in language python through FFI [1].

This chapter is organized as follows. First, we are going to discuss the architecture and implementation details of DiffKemp with relation to our extension in Section 5.1. Then we are going to describe the integration of pattern generation extension in Section 5.2 and in Section 5.3 we list all the limitation of the current implementation and obstacles we have encountered during the development process.

5.1 Architecture of DiffKemp

Architecture of DiffKemp was briefly discussed in the Chapter 2 but in this section we are going to highlight a few technical details, as for the purpose of this work we do not have to go too much in the depth about implementation of components. That is because our extension depends only on the outputs of *snapshot generator* and *snapshot comparator*. (Their implementation details can be found in [14]). With these two components we can split the comparison process into two phases which are handled by the forementioned components:

1. **Snapshot Generation** – In this phase DiffKemp compiles two versions of some project to LLVM IR language and generates so-called *snapshots*. This snapshot is an abstraction of one version of the compared project. It contains functions in LLVM IR together with a list of their names and locations in the original source code. To limit the scope of the analysis (and thus making it faster, as we then can analyze only the parts we need) the user may provide a function list, which contains names of functions that are going to be the subject of semantic comparison.

2. **Semantic Comparison** – Second phase is the compare phase, in which the DiffKemp compares two snapshots of the compared project. The algorithm behind can

[1] *Foreign Function Interface* allows interaction with commonly C functions in other, often scripting, language.[1]

be found in Section 2.2. Each function in both snapshots is analyzed and output of this phase is the decision whether said functions have the same semantics. Additionally, DiffKemp may provide information about the difference location, such differences are called *diff chunks*. Some of these differences may be ignored by using user-defined CCPs explained in Chapter 3.

Again, we only require the outputs of these two phases, that is because we need the information from the snapshot, and as the main goal of this thesis is to provide an automatic solution for generation of CCPs that should be able to filter out selected changes from the previous runtimes of DiffKemp, then we need the *diff chunks*. Because they capture the nature of the semantic-altering change. Unfortunately, getting these diff chunks automatically requires yet unmerged extension of DiffKemp, because of that we are handling whole LLVM IR functions.

As a result the proposed solution is loosely coupled with the rest of the codebase. Therefore, our implementation can be quite independent of the current architecture. We are going to look into it in more detail in the next section.

5.2 Integration of the Extension

In this section we are going to put our implementation of the proposed extension in the context of the architecture of DiffKemp. As mentioned in the previous section DiffKemp has two phases, and we have added a third optional phase to the process in a form of new sub-command. The knowledge required for implementation came from [8] and [12].

The integration can be illustrated as a slightly changed diagram of DiffKemp architecture known from Figure 5.1 (the new integrated component is colored in different color). In the diagram, we can see that the pattern generator takes previously made diff chunks as its input and then produces generated custom patterns as its output.

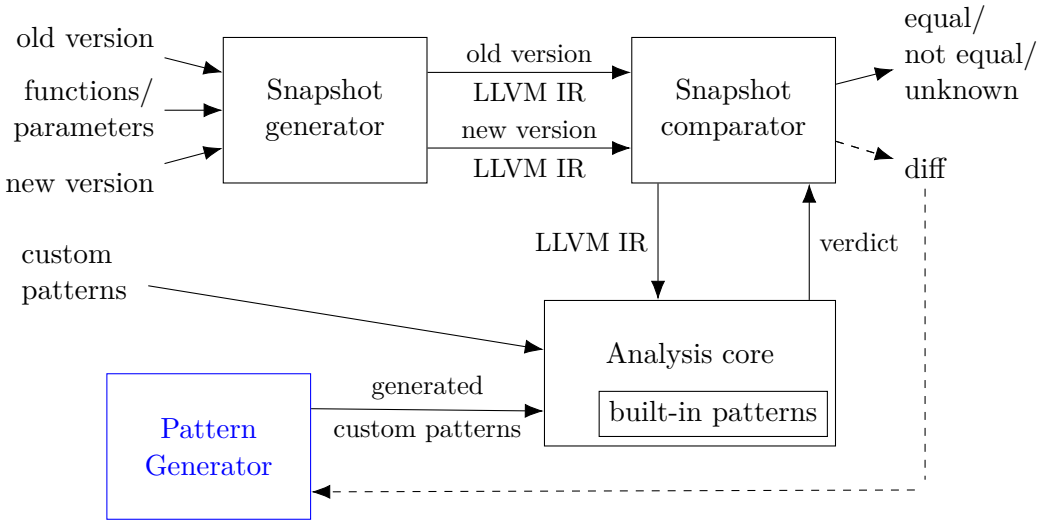


Figure 5.1: New architecture with integrated pattern generator. The original diagram comes from [11].

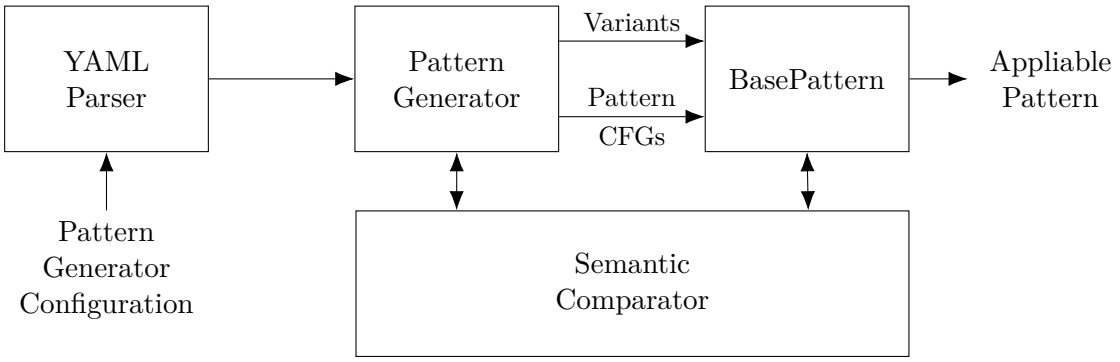


Figure 5.2: Components of the proposed extension

In the Figure 5.2 we can see the composition of components in the extensions implementation. In the rest of this section we are going to explain their relationship and what is their purpose.

- **YAML Parser** – We are using LLVM libraries which have build-in support for parsing files in YAML format which is based on C++ templates. More about our pattern generator configuration file and its format can be found in Subsection 5.2.3.
- **Pattern Generator** – This is the core component of the extensions as it handles most of the inference, creates the base pattern and assigns variants to it. It is implemented as the `PatternGenerator` class.
- **Base Pattern** – Contains all the necessary information to generate the pattern together with its variants. In the code it is represented by the `BasePattern` class.
- **Variants** – A set of instruction variants. All operations and information that are needed for their functionality are encaplusated in `InstructionVariant` class.
- **Semantic Comparator** – This is the implementation of Algorithm 1 by the DiffKemp development team, more information about it can be found in [12].

In the following subsection we are going to mention the biggest challenges encountered during the development and thus explain their impact on implementation.

5.2.1 Working With Structure Types

Generally, dealing with structure types in LLVM can get quite difficult. That is caused by LLVM maintainers approach to API regarding structures, as requested functions (by the community) are quite individual and greatly depend on the project. Therefore, LLVM maintainers decided that their approach to handling of structure types would be as minimal as possible, and they would let users of the LLVM libraries maintain their own implementations of utility functions.

Hence, we had to implement our own structure type remapping (so that it would be possible to change a type of a structure to a type of the other while cloning functions). This remapping was enabled by implementing the `ValueMapTypeRemapper` interface with our own class `StructTypeRemapper`.

5.2.2 Function Cloning

The only way to change LLVM IR function argument count (or types of those arguments) with LLVM API is through copying the function signature, changing it to our needs, then using it for initialization of an empty function and then copying the body of the original function to the new one with `llvm::cloneFunction`.

But that only covers the most basic needs as doing such an operation without some additional procedures would not yield a satisfactory result, because if that function is copied from different module (which is in our implementation the usual case), then it needs some additional declarations. Specifically, if there are any `call` instructions, then we have to check whether the called function actually exists in our (destination) module, otherwise we have to provide it. The same situation applies for global variables, if there were any instructions in the original function that used any global variable, that are not declared in the destination module, then they would be generated as undefined references.

The situation complicates even further with the appearance of structure types in the original function as we have to deal with type duplication which may happen when a structure type is copied from one module to the other. As if there already is some type in the destination module with the same name as the original structure type, then instead of implicit mapping to that type (because they do not necessary represent the same type) the LLVM creates a new type that receives a suffix.

But in our case we know, that the structure types do indeed represent the same type. The resolution of all the forementioned struggles is encapsulated in a function `cloneFunction`, which works as sofisticated wrapper around its LLVM counterpart.

5.2.3 Extension Input

Now we are going to describe the input of our extension. As mentioned in the Section 5.1 we need access to differences previously found by DiffKemp. At the moment of writing this thesis our only option was to simply take the whole differing functions. Therefore, we needed some way of expressing which functions should be used for generation by the user. We also needed some additional information about the functions such as path to their LLVM modules. Hence, we have decided to define a pattern generation configuration file in which the user can specify all forementioned details.

Fortunately, LLVM already has built-in support for parsing data in YAML format and therefore, we did not have to introduce any additional dependencies. All the information for implementing data serialization in YAML came from [9].

```
---
- name: <Pattern Name>:
    candidates:
      - name: <Function Name>,<Optional Alias>
        old_snapshot_path: <Path to the Old Snapshot>
        new_snapshot_path: <Path to the New Snapshot>
```

Figure 5.3: Structure of YAML configuration file

Figure 5.3 shows a generalized version of data format that we have created for the pattern generator configuration. We expect the user to define a series of desired patterns that are identified by name and each of these patterns contains a sequence of candidate

differences. Each difference is identified by the function name and an optional alias (as it is possible that the function name did change between versions), and contains paths to the old and the new snapshot, which contains all the necessary information.

5.3 Limitations

In this section we are going to mention limitations that our implementation has in comparison to the proposed method in the Chapter 4.

Order of candidate differences during inference

Parametrization should not be dependent on the order of candidate differences, but we overlooked this in the implementation of parametrization and due to the time pressure during finalization of this thesis we did not have enough time to fix it. Therefore, our implementation is not able to provide the same result as the example in Figure 4.3. With our implementation we would be left with the pattern showed in Figure 4.3c.

Chapter 6

Experimentation and Evaluation

This chapter contains a summary of various simple experiments. The goal of these experiments was to prove that the implementation of proposed method is able to generate patterns that are successfully appliable (with minimal interventions by the user) during comparison done by the DiffKemp.

During the development we have created a few regression tests that were then expanded with semantic comparison done by DiffKemp. Therefore, we were able to evaluate the applicability of our generated patterns. These test cases consisted of small sample program designed to demonstrate some particular change (i.e., in value, or in value with branching, etc.) and our expected output.

We have decided to split the experimentation into three sections. First, Section 6.1 where we summarize experiments without parametrization. Then Section 6.2 which is devoted to experiments with parametrization. And Finally, Section 6.3 where we discuss the achieved results,

6.1 Experiments without parametrization

In these experiments we have focused on a simple question, whether our method is able to generate patterns that would be appliable and successfully filter out changes, from which they have been inferred without parametrization. This is probably the simpler use-case of our proposed extension but nevertheless it does automate the process of creation of CCPs. All of these experiments were successful (although some of the generated patterns required small modification – i.e., adjusting the pattern range), therefore we are at least able to generate concrete patterns that should be able to filter out changes from which they come from.

6.2 Experiments with parametrization

Unfortunately, experiments that should prove that our extension is able to handle parametrization did not do so well. Although we were able to generate some patterns. We were not able to apply them during the comparison. Probable causes are discussed in the next section.

6.3 Evaluation of implemented extension

The unsatisfactory results of the experimentation with parametrization were probably caused by two main reasons: (1) our misunderstanding of nature of the patterns and changes they should capture and (2) the lack of a method, which would provide us with only the required differing instructions (but it is already implemented as an unmerged extension of DiffKemp, but we have decided to build this thesis on the stable release.) But on the other hand, we were able to generate patterns for situations that do not require parametrization and that is a foundation on which could some future work build upon.

Chapter 7

Conclusion

In this thesis we have proposed a method for automatic pattern generation that is based on parametrization of values in combination with pattern variant generation, which enables us to parametrize even further. Specifically by global variables and structure types. The extension is even capable of determination of pattern ranges.

Although the experimentation showed that we were not able to succesfully filter out changes with parametrized generated pattern, we were able to generate and successfully match patterns without parametrization. Therefore, we were not able to achieve generality, but our extension is usable and provides foundation for further improvement.

Future work could build on yet unmerged extensions of DiffKemp and use generalization features defined for CCPs, which could simplify (and potentially speed-up) the generation process. Additionally, merge of equivalence slicing extension to the upstream DiffKemp would open space for further research and experimentation. We think that the idea of base pattern together with pattern variation shows potential to enable limited parametrization of instructions without the need of extensive interference with functionality of DiffKemp core. Although there is the danger of so-called variant boom, where number of resulting variants may grow unpredictably fast.

Bibliography

[1] *CFFI Documentation — CFFI 1.15.1 Documentation* [online]. [cit. 2023-04-15]. Available at: https://cffi.readthedocs.io/en/latest/.

[2] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Second editionth ed. Addison-Wesley. Addison-Wesley Signature Series. ISBN 978-0-13-475759-9.

[3] Garrido, A. *Software Refactoring Applied to C Programming Language*. Urban-Champaign, USA, 2000. Master's thesis. University of Illinois.

[4] Kim, D., Nam, J., Song, J. and Kim, S. Automatic Patch Generation Learned from Human-Written Patches. In: *2013 35th International Conference on Software Engineering (ICSE)*. May 2013, p. 802–811. DOI: 10.1109/ICSE.2013.6606626. ISSN 1558-1225.

[5] Le Goues, C., Nguyen, T., Forrest, S. and Weimer, W. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*. january 2012, vol. 38, no. 1, p. 54–72. DOI: 10.1109/TSE.2011.104. ISSN 0098-5589.

[6] LLVM Project. *LLVM Language Reference Manual — LLVM 16 Documentation* [online]. [cit. 2022-12-07]. Available at: https://llvm.org/docs/LangRef.html.

[7] LLVM Project. *LLVM Opaque Pointers — LLVM 16 Documentation* [online]. [cit. 2022-12-07]. Available at: https://llvm.org/docs/OpaquePointers.html.

[8] LLVM Project. *LLVM Programmer's Manual — LLVM 16 Documentation* [online]. [cit. 2023-04-28]. Available at: https://www.llvm.org/docs/ProgrammersManual.html.

[9] LLVM Project. *YAML I/O — LLVM 16 Documentation* [online]. [cit. 2023-04-22]. Available at: https://www.llvm.org/docs/YamlIO.html.

[10] Long, F., Amidon, P. and Rinard, M. Automatic Inference of Code Transforms for Patch Generation. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, August 2017, p. 727–739. DOI: 10.1145/3106237.3106253. ISBN 978-1-4503-5105-8.

[11] Malík, V. *DiffKemp: Automatic Analysis of Semantic Differences in Kernel Options* [online]. [cit. 2023-04-14]. Available at: https://research.redhat.com/blog/research_project/diffkemp-automatic-analysis-of-semantic-differences-in-kernel-options/.

[12] Malík, V. *DiffKemp* [online]. April 2023 [cit. 2023-04-15]. Available at: https://github.com/viktormalik/diffkemp.

[13] Malík, V., Šilling, P. and Vojnar, T. Applying Custom Patterns in Semantic Equality Analysis. In: Koulali, M.-A. and Mezini, M., ed. *Networked Systems*. Cham: Springer International Publishing, 2022, p. 265–282. Lecture Notes in Computer Science. DOI: 10.1007/978-3-031-17436-0-18. ISBN 978-3-031-17436-0.

[14] Malík, V. and Vojnar, T. Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects. In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. P. 329–339. DOI: 10.1109/ICST49551.2021.00045. ISSN 2159-4848.

[15] Martinez, M., Duchien, L. and Monperrus, M. Automatically Extracting Instances of Code Change Patterns with AST Analysis. In: *2013 IEEE International Conference on Software Maintenance*. P. 388–391. DOI: 10.1109/ICSM.2013.54. Available at: http://arxiv.org/abs/1309.3730.

[16] Martinez, M. and Monperrus, M. Coming: A Tool for Mining Change Pattern Instances from Git Commits. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. May 2019, p. 79–82. DOI: 10.1109/ICSE-Companion.2019.00043. ISSN 2574-1934.

[17] Nguyen, H. A., Nguyen, T. N., Dig, D., Nguyen, S., Tran, H. et al. Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, p. 819–830. DOI: 10.1109/ICSE.2019.00089. ISSN 1558-1225.

[18] Padioleau, Y., Lawall, J., Hansen, R. R. and Muller, G. Documenting and Automating Collateral Evolutions in Linux Device Drivers. *ACM SIGOPS Operating Systems Review*. april 2008, vol. 42, no. 4, p. 247–260. DOI: 10.1145/1357010.1352618. ISSN 0163-5980.

[19] Vojnar, T. Static Analysis and Verification. [online]. [cit. 2023-05-04]. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-01.pdf.

[20] Šilling, P. *Applying Code Change Patterns during Analysis of Program Equivalence*. Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/24037/.

[21] Weisstein, E. W. *NP-Problem* [online]. Wolfram Research, Inc. [cit. 2023-04-14]. Available at: https://mathworld.wolfram.com/.

Appendix A

Contents of the Attached Medium

The attached SD Card contains following files and directories:

- `/diffkemp/` - Source codes of DiffKemp.
- `/tex/` - LaTeXsource codes, that is, the text of this thesis
- `/init.sh` - Project initialization script.
- `/README.md` - Readme for the implementation of proposed extension.
- `/xkrizd03_bp.pdf` - This PDF.

The source codes that implement the proposed pattern generation solution can be found in the `/diffkemp/diffkemp/simpll` directory. Specificall we are talking about:

- `PatternGenerator.cpp` and `PatternComparator.h` - Pattern inference and generation.
- `PatternAnalysis.cpp` and `PatternAnalysis.h` - Top-level access to the generator from FFI.
- `Snapshot.h` - Mapping of snapshots in YAML.
- `PatternBase.cpp` and `PatternBase.h` - Abstraction over base pattern.
- `InstructionVariant.cpp` and `InstructionVariant.h` - Abstraction over instruction variant.

The tests that were used for the validation and the evaluation of the extension can be found in the `/diffkemp/tests/regression` directory. Most notably:

- `pattern_generation/` - Source files and configuration required for runtime of both the semantic comparison and the pattern generation.
- `pattern_generation_test.py` - Test runner implemented in python.

Appendix B

How to Build and Run the Software

This appendix is devoted to the building and execution of DiffKemp together with our pattern generation extension. We assume that all the actions are going to be executed inside of development container image. But we have fixed a few issues with the upstream version and because of that we have to build the custom image (in this examples we are going to use `podman` but it can be used interchangeably with `docker` as they both follow the same standard):

```
podman build -t localhost/diffkemp ./docker/diffkemp-devel/
```

After the succesful build we can access the newly build container with this command:

```
./docker/diffkemp-devel/run-container.py -image localhost/diffkemp
```

There is a small chance that the build would not be succesful, but that is not real compilation error but rather some mistmatch due to the usage of container. If that would happen the simply in the container run these three commands:

```
cmake -GNinja -Bbuild -DCMAKE_BUILD_TYPE=Debug -DSIMPLL_REBUILD_BINDINGS=1
cmake -build build
pip install -e .
```

At that moment you should be able to access the DiffKemps command line interface located in `./bin/diffkemp`.