



HAL
open science

Semantic Patch Inference

Denis Merigoux

► **To cite this version:**

| Denis Merigoux. Semantic Patch Inference. Programming Languages [cs.PL]. 2016. hal-02936287

HAL Id: hal-02936287

<https://inria.hal.science/hal-02936287v1>

Submitted on 11 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

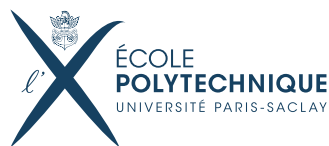


SEMANTIC PATCH INFERENCE

Rapport de stage de recherche (non confidentiel)

Option Département d'informatique
Champ Ingénierie logicielle
Enseignant référent Francesco ZAPPA NARDELLI
Tutrice de stage Julia LAWALL
Dates 21 mars - 29 juillet 2016
Organisme LIP6 - 4 place Jussieu - 75015 Paris

Denis MERIGOUX X2013



Déclaration d'intégrité relative au plagiat

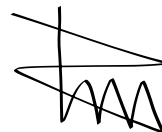
Je soussigné MERIGOUX Denis certifie sur l'honneur :

- Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.
- Que je suis l'auteur de ce rapport.
- Que je n'ai pas utilisé des sources ou résultats tiers sans clairement les citer et les référencer selon les règles bibliographiques préconisées.

Mention à recopier : *Je déclare que ce travail ne peut être suspecté de plagiat.*

Date : 05/07/2016

Signature :





Résumé

Afin d'améliorer la qualité du code C du noyau Linux, les développeurs se retrouvent souvent à répéter des transformations qui peuvent être spécifiées dans un langage de plus haut niveau, le *Semantic Patch Language* (SmPL, [13]). Néanmoins, l'écriture de la spécification de transformation est non-triviale et oblige le développeur à réfléchir au contexte et au niveau d'abstraction de sa transformation. Pour rendre cette tâche plus facile, nous proposons un nouvel outil appelé Spinfer, inspiré des approches d'Andersen et al. [2] et de Kim et al. [10] qui travaille à la fois au niveau des nœuds et du graphe d'un « graphe de flot de contrôle unifié » pour identifier les modèles de modifications et leurs contextes, liant les variables abstraites pour inférer un patch sémantique complet à partir d'exemples de code modifié.

Abstract

When improving the C code of the Linux kernel, developers often tend to repeat transformations that can be specified using a higher-level language, the Semantic Patch Language (SmPL, [13]). Nevertheless, writing the transformation specification is not trivial and requires the developer to determine the context and the level of abstraction of the transformation. To solve this, we propose a new tool named Spinfer, inspired by the approaches of Andersen et al. [2] and Kim et al. [10], working at both node-level and graph-level of a “merged CFG” to identify common patterns of modification and their context, linking abstract metavariables to infer a complete semantic patch from examples of modified code.

Contents

1	Context and related work	5
1.1	The Linux kernel: large code base and collateral evolution	5
1.2	Related work: semantic patch inference	6
1.2.1	Safety and conciseness	6
1.2.2	Spdiff	7
1.2.3	LASE	8
1.2.4	Review	9
2	A mixed node-graph tree-based semantic patch inference	10
2.1	Data structures	10
2.1.1	Motivations	10
2.1.2	Merged CFG	10
2.1.3	Skimming	12
2.2	Extracting common patterns	15
2.2.1	Abstraction of patterns	15
2.2.2	Pattern mining	16
2.2.3	Abstraction rules	19
2.2.4	Usage	20
2.3	Building sequences of patterns	20
2.3.1	Ordering on CFG nodes	21
2.3.2	Growing pattern sequences	21
2.4	Producing semantic patches	22
3	The Spinfer program	23
3.1	Implementation	23
3.2	Current results	23
3.3	Performance analysis	25
4	Conclusion	25



1 Context and related work

1.1 The Linux kernel: large code base and collateral evolution

Improving a large code base such as the Linux kernel (more than 16 million LOC) requires developers to perform systematic code modifications whenever an API is modified. For instance, adding an argument to a function or changing from a structure to a pointer implies changing all the calls to the function or to the data structure, sometimes with hundreds of occurrences. This category of code modification, called a collateral evolution, is described by Padioleau et al. [12]. Padioleau et al. state that they represent a third of the modifications in the Linux kernel, especially concerning the files in the `drivers` folder.

Listing 1 – A transformation specified in SmPL

```

1 @@ expression X0,X1; @@
2 X1 = pinctrl_register(...)
3 ... when != X1
4 - if(!X1)
5 + if(IS_ERR(X1))
6 {
7     ...
8 -     return X0;
9 +     return PTR_ERR(X1);
10 }
```

Coccinelle specifically address this issue by allowing developers to specify and perform transformations of C code using a higher-level language, the Semantic Patch Language (SmPL). A code transformation specified in SmPL is called a semantic patch. The “patch” terminology and the SmPL syntax are inspired from the UNIX `patch` tool. Listing 1 is an example of semantic patch. When we give to Coccinelle the SmPL semantic patch and a collection of source files, Coccinelle will analyse the semantic patch to determine in what context the transformation should be applied. For instance, we can see that lines 6, 7 and 9 of Listing 2 match the context of the transformation specified by Listing 1. Then Coccinelle applies the semantic patch for every match of the context of the transformation in the source code files. We can see the result of the application of the semantic patch of Listing 1 on the source code of Listing 2 in Listing 3.

Listing 2 – `drivers/pinctrl/bcm/pinctrl-bcm2835.c` : excerpt of the code before application of commit `323de9ef` of the Linux kernel.

```

1 @@ -1029,23 +1029,23 @@
2 if (err) {
3     dev_err(dev, "could not add GPIO chip\n");
4     return err;
5 }
6 pc->pctl_dev = pinctrl_register(&bcm2835_pinctrl_desc, dev, pc);
7 if (!pc->pctl_dev) {
8     gpiochip_remove(&pc->gpio_chip);
9     return -EINVAL;
10 }
11 pc->gpio_range = bcm2835_pinctrl_gpio_range;
12 pc->gpio_range.base = pc->gpio_chip.base;
```

Listing 3 – `drivers/pinctrl/bcm/pinctrl-bcm2835.c`: excerpt of diff output of commit [323de9ef](#) of the Linux kernel.

```
1 @@ -1036,9 +1036,9 @@
2 pc->pctl_dev = pinctrl_register(&bcm2835_pinctrl_desc, dev, pc);
3 - if (!pc->pctl_dev) {
4 + if (IS_ERR(pc->pctl_dev)) {
5 gpiochip_remove(&pc->gpio_chip);
6 - return -EINVAL;
7 + return PTR_ERR(pc->pctl_dev);
8 }
```

The syntax of SmPL is similar to the unified diff syntax with lines prefixed by + and -, but offers powerful features such as the `...` operator which stands for “for all execution paths” or any number of arguments in a function, depending on the context, and metavariables (*e.g.* `X0` and `X1`) which stand for any expression, type, statement, etc. `@@` introduces a semantic rule; a semantic patch consists in one or several semantic rules.

Coccinelle has been adopted by the Linux kernel community and has been used in more than 2,800 commits to the Linux kernel since 2007. Nevertheless, designing a semantic patch is a time-consuming task and requires the developer to learn SmPL, thus limiting the ease of use of such a tool for matching and transforming code.

This issue provides a first use case for a tool that would produce semantic patches from a collection of code modification examples: semi-automate the refactoring operations. The developer would just need to create a small but significant set of modification examples, from which the general rule of transformation can be inferred.

Another problem related to collateral evolutions for the developers is to understand these evolutions, when confronted with a legacy code base. Information about the history of modifications is available thanks to Version Control Systems such as git but searching in the history of commits is cumbersome with the current features of these VCS. Lawall et al. [8] address this problem with the Prequel tool by allowing to search for commits that match with a code transformation specified in SmPL. A semantic patch inference tool could then allow to search for commits similar to a specified commit: a semantic patch could be inferred from the original commit and then given to Prequel which could then search for other commits implementing similar transformations. A semantic inference patch tool could also provide a higher-level description of the contents of a commit, allowing the developer to understand faster and better the transformation implemented by a specific commit.

1.2 Related work: semantic patch inference

Many attempts of semantic patch inference have been made, and are summarized into Kim’s work [7]; but the two most relevant tools are LASE and Spdiff, which are discussed in Sections 1.2.2 and 1.2.3. To better understand semantic patch inference, Section 1.2.1 first defines two important notions.

1.2.1 Safety and conciseness

Andersen’s theory of atomic patching [1] defines two very important notions to evaluate the results of a semantic patch: safety and conciseness. We consider a simplified language that only has functions and integers. Here is a set of “before” and “after” code fragments.

1. $f(1, g(1)) \rightarrow f(7, g(1))$



2. $f(1, 2) \rightarrow f(7, 2)$
3. $g(f(3, 2)) \rightarrow g(f(7, 2))$
4. $g(3) \rightarrow g(3)$ (this term is not modified)
5. $g(2) \rightarrow g(9)$

If we consider the semantic patch $f(x, y) \Rightarrow f(7, y)$ where x and y are metavariables, it applies to situations 1, 2 and 3 and produces correct results, and does not apply to situations 4 and 5: this patch is safe. On the other hand, the semantic patch $g(x) \Rightarrow g(9)$ is not safe because it would provide wrong results on situation 4. Thus safety is always defined relative to a set of changed (or unchanged code). The precise definition proposed by Andersen is more complicated, but the global idea is that a patch is safe, it will not perform unwanted modifications.

The semantic patches $f(x, y) \Rightarrow f(7, y)$ and $g(2) \Rightarrow g(9)$ are concise because they implement all the modifications of the modified code base. The conciseness is thus related to the level of abstraction of a patch and the proportion of changes covered by the patches. For instance, $f(1, y) \Rightarrow f(7, y)$ is not concise since it does not cover situation 3; to implement all the modification, we would need another patch $f(3, y) \Rightarrow f(7, y)$. Having two rules instead of one to describe the same modifications is less concise. The conciseness of a patch is also relative to the modified code base.

Safety and conciseness are distinct properties and have no logical implications between them. The ideal goal of semantic inference is to infer patches that are both safe and concise. But there exists a tradeoff between safety and conciseness when inferring semantic patches. Indeed, the more metavariables in the semantic rules, the more concise it typically becomes but also the more risk there is that it becomes unsafe. On the other hand, a trivial solution to semantic patch inference is to list all the modifications without using any metavariables, but that would not help to understand the transformation at a higher-level and would not be robust, in the sense that the inferred semantic patch could not be applied to code different from the examples it has been inferred from.

1.2.2 Spdiff

Andersen and Lawall have already developed a semantic patch inference tool called Spdiff [2]. The semantic patches it produces feature both `...` and metavariables. Spdiff's workflow is made to try to satisfy safety and conciseness, using algorithms derived from the above theory of atomic term patching [1]. It is suggested that Spdiff be used with small amounts of code chosen by the programmer; Spdiff is good at finding context to modifications (context that `diff` does not provide) from a small set of code but does not scale up well when confronted with thousands of lines of code, as the algorithm that determines the right level of abstraction (to be the most concise while remaining safe) enumerates all the possibilities of creating metavariables in order to choose a good one.

Spdiff works by first identifying maximal semantic patterns, *i.e.* semantic patches with no added or removed nodes, which match the code before the modifications, then trying to pair pieces of modifications to this maximal semantic pattern. A side-effect of this technique is the output of irrelevant, too abstract context in the semantic patch (for instance `*X6 X5 ...`) consisting only of metavariables not related to those in the modified code. We can see this effect in Listing 4, as well as the apparition of `devm_ioremap_resource` in the semantic patch; it was added like context of the modification but it is too specific and might undermine the robustness of the semantic patch.

Listing 4 – Output of Spdiff on the same commit [323de9ef](#) of the Linux kernel described by the semantic patch of listing 1.

```

1 @@
2 expression X21; expression X26; X21; struct pinctrl_desc X22; identifier X12;
3 X24 *X25; expression X23; expression X14; X7 X8; expression X9;
4 expression X5; X6;
5 @@
6 X6 *X5;
7 ...
8 X8=X9;
9 ...
10 X25->X12=devm_ioremap_resource(X23, X14);
11 ...
12 X21=pinctrl_register(&X22, X23, X25);
13 ...
14 - return -[X26];
15 + return PTR_ERR(X21);

```

Nevertheless, SmPL has many strong points as a result language for inference: metavariables can stand for any-type expressions (expression `X14`) or have a specific type (struct `pinctrl_desc X22`). Metavariables can occur multiple times in the semantic patch, and this is very important for added pieces of code since Spdiff uses what was in the code before modifications to describe the added code: the `X21` in `X21=pinctrl_register(&X22,X23,X25);` is the same `X21` as in `+ return PTR_ERR(X21);`.

While the underlying theory of Spdiff is correct, the implementation chosen for the tool leads to some drawbacks in its practical usage. Indeed, some code structures like `if(!X1)` cannot be inferred by Spdiff at the moment. Moreover, because of its internal workflow of first identifying the patterns in the “before” code then adding the modifications, we have observed empirically that in the semantic patches produced by Spdiff, the context tend to include irrelevant patterns of context code and the modified patterns tend to cover only a part of the modifications.

1.2.3 LASE

Another interesting approach for semantic patch inference is LASE (Kim et al., [10]). LASE operates on Java code and the authors have also chosen to design their own dedicated rule application tools to apply the semantic patches that they infer. Listing 5 is an example of a semantic patch produced by LASE. The metavariables in this semantic patch are `u$0:FieldAccessOrMethodInvocation` (meaning that this metavariable is either a field access or a method invocation) and `v$0`. There is no `...` in LASE semantic patches; all the statements in the semantic patches have to be contiguous.

Listing 5 – Example of inferred edit script cited from [10] written using a SmPL-like syntax.

```

1 Iterator v$0 = u$0:FieldAccessOrMethodInvocation.values().iterator();
2 while(v$0.hasNext())
3 - MVACTION action = (MVACTION)v$0.next();
4 - if(action.m$0())
5 + Object next = v$0.next();
6 + if(next instanceof MVACTION)
7 + MVACTION action = (MVACTION)next;

```

The major phases of LASE’s patch inference are:

- perform a tree-differencing on the ASTs of the code before and after to find tree edit scripts, which are lists of atomic tree operations such as “remove a node” to transform the AST of the



code “before” to the AST “after”;

- determine the longest common edit script while abstracting some identifiers (names of variables, methods, etc.);
- determine the context of the modification using first code clone detection [6] on the code of the methods impacted by the modifications, then filtering the results of the clone detection by extracting the largest subtree common to the ASTs of these methods.

Code clones detectors use a large variety of techniques based on tokens, Abstract Syntax Trees (AST) or Control Flow Graphs (CFG) to identify contiguous blocks of code that are repeated over a code base, the meaning of “repeating” being either in the sense of text equality or a more abstract semantic equality. Code clone detection is largely used in software engineering and has been extensively studied; a general overview of the topic is provided by Roy et al. [14].

The common subtree extraction of LASE operates on a AST whose nodes are incomplete statements in Java code; in fact this AST can be seen as a non-cyclic CFG since for example, an `if` node contains the condition of the `if` and has two children (the condition can be true or false), a variable assignment in a sequence is a node containing the variable assignment and has one child.

The method used in LASE considers first the changes, and then tries to add the context. Thus, in the output, the modified patterns would tend to be all taken into account but the context would tend to be minimized. In a sense, LASE’s inference is less sophisticated than Spdiff’s:

- LASE only abstracts names of variables (`u$0`, `v$0`) whereas Spdiff’s can abstract expressions;
- LASE only produces patches that capture contiguous pieces of code and happen at most once per method declaration.

Kim et al. do not provide any theoretical insight comparable to Spdiff’s preliminary work [1], but include an empirical study of LASE’s performance based on 24 examples of semantic patches which suggests that this workflow is effective. Particularly, LASE’s semantic patches are not safe because the precision of the tool is not 100%. Nevertheless, the conciseness seems to be quite good when looking at the recall scores that tend to 100%.

1.2.4 Review

Any attempt on producing an inference tool depends on a higher-level transformation specification language. The inference features are limited by the features of the specification language. That is why SmPL is a good choice as a transformation specification language since it already features a vast choice of constructions to fine-tune the transformation specification. Indeed, SmPL and Coccinelle have already been extensively used in the Linux kernel to perform collateral evolutions, as shown by Lawall et al. [11] and by the significant collection of semantic patches available on the [Coccinellery website](#). Using SmPL as the output language for our tool Spinfer is thus a logical choice.

The strong points of Spdiff are its context detection and the concept of sequentiality of patterns thanks to `...` in the semantic patches it produces, which has no equivalent in LASE. The strong points of LASE are its efficiency since it uses sequence and token-based matching to identify the common patterns. The goal of the new tool we propose, Spinfer, is to combine the strengths of these two approaches: using an efficient workflow inspired by LASE to produce semantic patches in SmPL having the same expressiveness as Spdiff’s. Rather than focusing only on safety or conciseness, Spinfer’s approach would rather be result-oriented in the sense that its primary objective would be to always

produce a semantic patch to account for the modifications in its input. If it cannot produce a safe semantic patch, it would rather output an unsafe patch along with informations on why it is unsafe.

2 A mixed node-graph tree-based semantic patch inference

The title of this section sums up the characteristics of Spinfer’s semantic patch inference: it works with the CFG and considers the CFG nodes as ASTs, and the main inference is tree-based since it is performed on the CFG nodes. During the reading of this section, one can refer to Figure 1 for an overview of the different phases of the workflow of Spinfer.

2.1 Data structures

2.1.1 Motivations

When analyzing the input code, we have to choose whether to represent it as an Abstract Syntax Tree (AST), a Control Flow Graph (CFG) or another data structure. This choice has a great impact on what is possible to analyse. Indeed, because LASE works exclusively with the AST, it would not be able to offer the `...` feature of Spdiff requires the use of a CFG, since the AST does not offer the semantics of execution paths. Because of this, Spinfer uses the CFG as its primary data structure to represent the code it analyses. The nodes of the CFG are roughly the instructions of C code; for instance an `if` instruction would have its node (the condition of the `if` is stored in the node) and two successors in the graph, one for the case where the condition is true, and the other when it is false. A `for` instruction would have one node for it self, one successor which is the first instruction of the body of the loop, and two predecessors: the instruction before the `for` in the code and also the last instruction of the body of the loop.

Because we analyse modified code, we actually need to build two CFGs: one for the code “before” and one for the code “after”. The problem is that with keeping two CFGs to represent modified code, the modifications in the code are not aligned: we do not know which nodes are present in the two CFGs and which nodes have been modified. This is the same problem with a modified source code file: we want to know which lines have been modified from the two versions of the file. Our solution described in Section 2.1.2 is to use the information of `diff` executed on the two source files used to make the two CFGs to merge the CFGs of the “before” and “after” versions of the code.

Furthermore, we want to make our data structure more compact to reduce as much as possible the future cost of the pattern identifying algorithm, to avoid the pitfall of Spdiff not terminating in a reasonable amount of time. Indeed the source code files Linux drivers files passed as input of the program would have lengths around 1,000 LOC. Our solution described in Section 2.1.3 is to remove all the nodes of the CFG that we suppose will not be part of the context of any semantic patch using a code clone detector.

The input of our program is a set of pair of source code files. Each pair of files are the two versions of a same source code file, “before” and “after” modification. In the next sections, we will describe how to handle one of these pairs of files; but the same operations are repeated for every pair of files given as input to Spinfer.

2.1.2 Merged CFG

Identifying and aligning the differences between two source code file is non-trivial. We preferred to leave this task to the traditional UNIX `diff`, which indicates the line numbers which have been

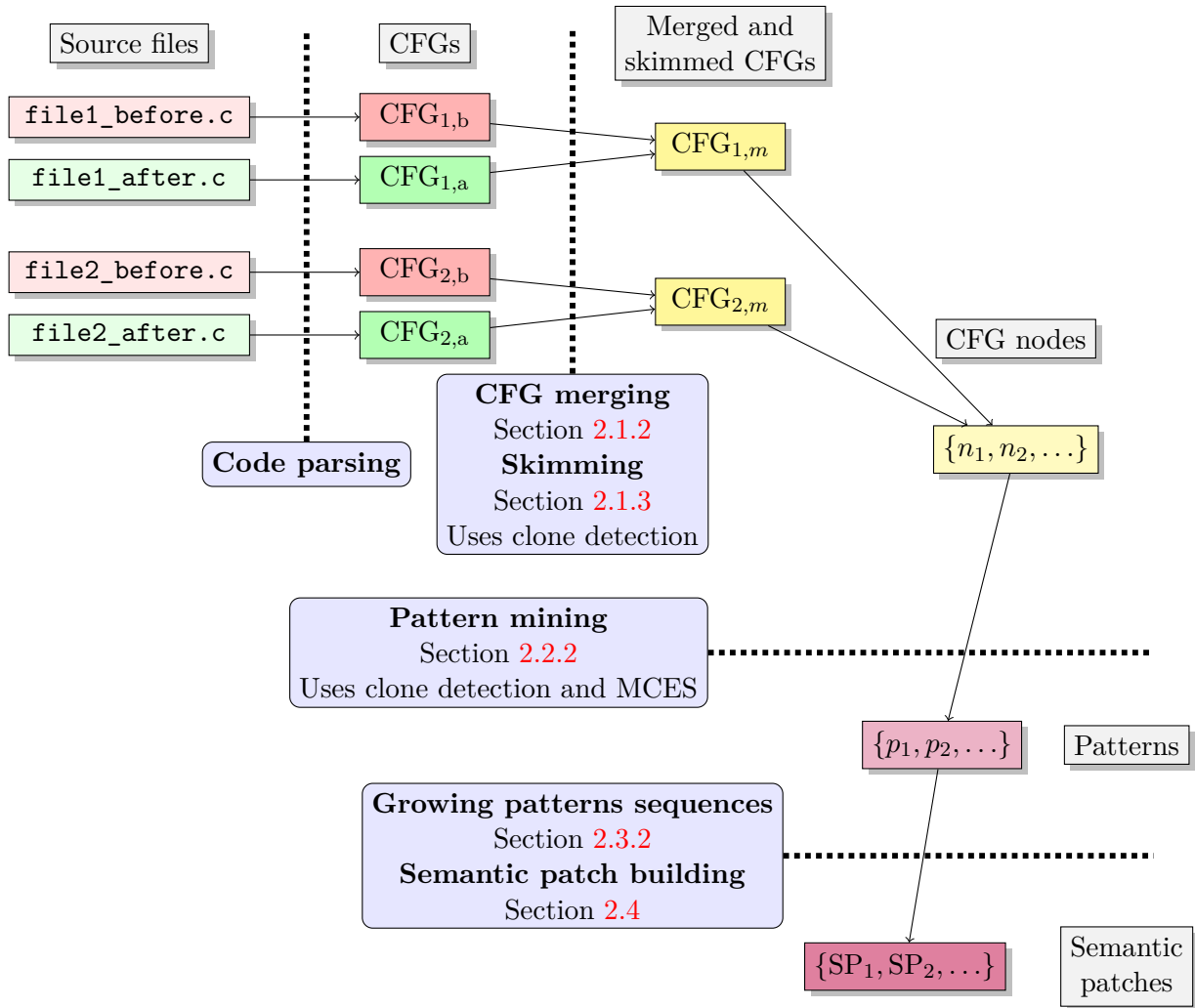


Figure 1 – General overview of Spinfer’s workflow. The shadowed boxes with sharp corners represents the data passed along, the non-shadowed boxes with rounded corners represent the different algorithms used. When an arrow between two data boxes crossed a dotted line, it means that the data is transformed by the algorithm at the end of the dotted line.

modified in the code “before” and “after”. Using the information gathered during the parsing of both versions of the code and the result of the `diff`, we are then able to tag the nodes of both CFG with the following rules:

- if a node in the “before” CFG is associated with a line that `diff` reports as being removed, we tag the node with `-`;
- if a node in the “after” CFG is associated with a line that `diff` reports as being added, we tag the node with `+`;
- if a node is not associated with any line that `diff` reports as being removed or added, we tag the node with `0`.

Next, we merge the two CFGs into a single one, based on this tag information. The underlying assumption of this algorithm is that since nodes tagged with `0` are not modified, they appear in both CFGs but they will appear only once in the merged CFG. On the other hand, if a line of code is modified, it will be tagged by `-` in the “before” CFG and tagged by `+` in the “after” CFG. Then, the two versions of the node will appear in the merged CFG, each version linked to its correct predecessors and successor who also appear in the merged CFG, whether they are modified or not. Figure 2b illustrates the behavior of this algorithm.

The algorithm is implemented in a single parallel traversal of both the “before” and “after” CFGs (see Algorithm 1). The general principle is the following: as long as we traverse nodes that are not modified, the parallel traversal is synchronized between the two graphs. But when for instance we encounter a removed node on the “before” CFG that is not present (by definition) in the “after” CFG, we stop our traversal of the “after” CFG while continuing in the “before” CFG until we find in the “before” CFG the node at which we stopped at in the “after” CFG. We are sure to reach this point because if the node we stopped at in the “after” CFG is not modified, then we are sure it will also appear in the “before” CFG. Then, we resume our synchronized parallel traversal.

Another solution for aligning the differences of the CFGs could have been to use a tree-differencing algorithm such as GumTree by Falleri et al. [3] to align the differences of the AST of the code “before” and “after” and then transformed this merged AST into a merged CFG.

2.1.3 Skimming

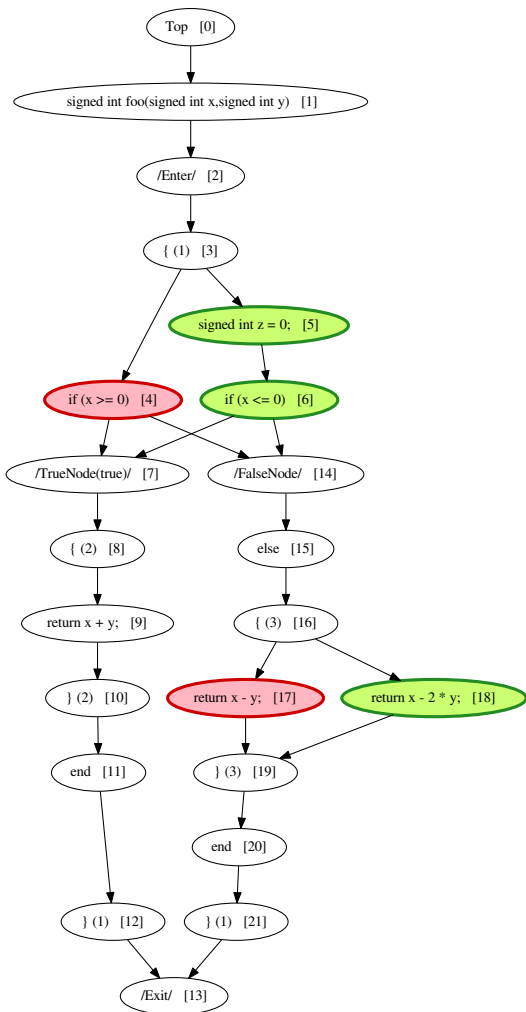
We now dispose of a merged CFG for every modified source code file. But in the worst-case scenario when each file features only one occurrence of the modification (for instance two modified LOC out of a thousand), we would pass onto our later analysis a significant amount of code that is not related at all to the semantic patch we want to produce. Our goal is to get rid of this irrelevant code while keeping, in addition to the modified code, all the code that might be part of the context of a semantic patch. However, we suppose that the semantic patch we want to produce will match several occurrences in the code we have; to achieve this, the relevant context code should be repeated several times throughout our codebase. Thus this context code we want to keep will appear in the output of a code clone detector. The same idea is used by Kim et al. [10] to produce an over-approximation of the context of a modification.

Spinfer runs a code clone detector, Deckard [5] by Jiang et al., over all the nodes of the CFG to detect groups of similar nodes. In this operation, all the identifiers and constants are abstracted, so the similarity is purely structural: for instance, `if(x==0)` is similar to `if(y==1)` but is not similar to `while(x==0)`. Deckard works by generating characteristic vectors of a real vector space of finite

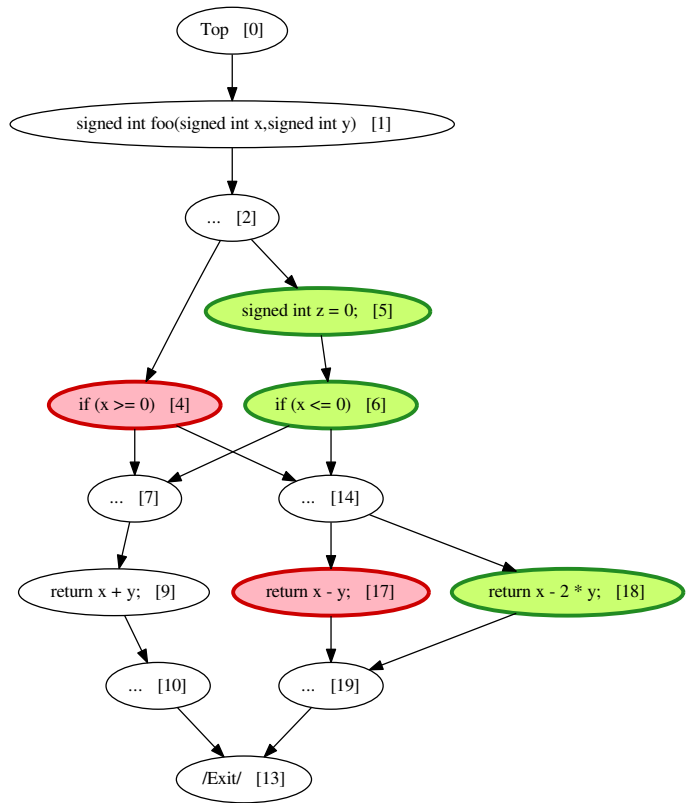
```

1 static int foo(int x, int y)
2 {
3 +   int z=0;
4 -   if (x >= 0) {
5 +   if (x <= 0) {
6         return x+y;
7       } else {
8 -       return x-y;
9 +       return x-2*y;
10      }
11 }
    
```

(a) Snippet of modified code



(b) Original merged CFG



(c) Skimmed merged CFG

Figure 2 – Control flow graphs produced by Spinfer; red nodes represent removed code and green nodes represent added code. The function definition node and a return node have been kept in the skimmed graph because they were signaled as clones of code. The additional nodes such as /FalseNode/ are produced by the C parsing library and are not significant.

Algorithm 1 CFG merging

Input: The recursive algorithm passes along a data structure D that contains: G_b , G_a and G_m , *i.e.* the CFG of the code before modification, after modification and the merged CFG being built, M_a and M_b the mappings between the nodes G_m and respectively G_a and G_b ($M_a(n_{p,a})$ is the node of G_m mapped to $n_{p,a}$ for instance), $n_{p,b}$ and $n_{p,a}$ the nodes of G_b and G_a previously visited by the algorithm and $n_{c,b}$ and $n_{c,a}$ the nodes being visited in the current call.

Output: The algorithm returns the same data structure containing G_m updated in a functional style.

```

1: function CFGMERGING( $D$ )
2:   if  $D.n_{c,b}$  and  $D.n_{c,a}$  already visited then ▷ The visited nodes are in the mappings.
3:     Add arcs  $D.M_b(D.n_{p,b}) \rightarrow D.M_b(D.n_{c,b})$  and  $D.M_a(D.n_{p,a}) \rightarrow D.M_a(D.n_{c,a})$  on  $D.G_m$ 
4:     return  $D$ 
5:   else if  $D.n_{c,x}$  already visited with  $x \in \{a, b\}$  then
6:     Add arc  $D.M_x(D.n_{p,x}) \rightarrow D.M_x(D.n_{c,x})$  on  $D.G_m$ 
7:     return  $D$ 
8:   else
9:      $D' \leftarrow D$ 
10:    if  $D.n_{c,a} = D.n_{c,b}$  (including modification prefix equality) then
11:      Add node  $n_m$  to  $D'.G_m$  and add  $D.n_{c,b} \rightarrow n_m$  to  $D'.M_b$  and  $D.n_{c,a} \rightarrow n_m$  to  $D'.M_a$ 
12:      Add arcs  $D'.M_b(D.n_{p,b}) \rightarrow n_m$  and  $D'.M_a(D.n_{p,a}) \rightarrow n_m$  to  $D'.G_m$ 
13:      for all  $(n_{s,b}, n_{s,a})$  pair of respective successors of  $(D.n_{c,b}, D.n_{c,a})$  do
14:        If  $D.n_{c,b}$  and  $D.n_{c,b}$ 's lists of successors don't have the same length we complete the
last pairs by repeating the last successor of the shorter list.
15:         $D' \leftarrow (D'.G_a, D'.G_b, D'.G_m, D'.M_a, D'.M_b, D.n_{c,a}, D.n_{c,b}, n_{s,b}, n_{s,a})$ 
16:         $D' \leftarrow$  CFGMERGING( $D'$ )
17:      end for
18:      return  $D'$ 
19:    else if  $D.n_{c,a}$  and  $D.n_{c,b}$  are modified then
20:      Add nodes  $D.n_{m,b}$  and  $D.n_{m,a}$  to  $D'.G_m$  and
21:      Add  $D.n_{c,b} \rightarrow D.n_{m,b}$  to  $D'.M_b$  and  $D.n_{c,a} \rightarrow D.n_{m,a}$  to  $D'.M_a$ 
22:      Add arcs  $D'.M_b(D.n_{p,b}) \rightarrow D.n_{m,b}$  and  $D'.M_a(D.n_{p,a}) \rightarrow D.n_{m,a}$  to  $D'.G_m$ 
23:      for all  $(n_{s,b}, n_{s,a})$  pair of respective successors of  $(D.n_{c,b}, D.n_{c,a})$  do
24:         $D' \leftarrow (D'.G_a, D'.G_b, D'.G_m, D'.M_a, D'.M_b, D.n_{c,a}, D.n_{c,b}, n_{s,b}, n_{s,a})$ 
25:         $D' \leftarrow$  CFGMERGING( $D'$ )
26:      end for
27:      return  $D'$ 
28:    else if  $D.n_{c,x}$  is modified and  $D.n_{c,y}$  is not, where  $x \neq y$ ,  $x, y \in \{a, b\}$  then
29:      Add node  $n_m$  to  $D'.G_m$ , add  $D.n_{c,x} \rightarrow n_m$  to  $D'.M_x$ 
30:      Add arc  $D'.M_x(D.n_{p,x}) \rightarrow n_m$  to  $D'.G_m$ 
31:      for all  $n_{s,x}$  successor of  $D.n_{c,x}$  do
32:         $D' \leftarrow (D'.G_a, D'.G_b, D'.G_m, D'.M_a, D'.M_b, D.n_{p,y}, D.n_{c,x}, n_{c,y}, n_{s,x})$ 
33:         $D' \leftarrow$  CFGMERGING( $D'$ )
34:      end for
35:      return  $D$ 
36:    else if  $n_{c,b} \neq n_{c,a}$  but neither are modified then
37:      return  $D$ 
38:    end if
39:  end if
40: end function

```



dimension from the ASTs, then by clustering these vectors in the vector space. The vectors inside a cluster correspond to a group of similar ASTs.

As described in Jiang et al.’s work, the vector generation works by giving an index to each tree pattern encountered during the post-order traversal of the AST. As there is a finite number of binary tree patterns having a maximum given height, it is possible to give each of them a specific index. Each time a pattern is encountered, the value in the vector at its index is incremented by 1.

Spinfer reimplements the vector generation module of Deckard to adapt it to its own data structures for code representation. Indeed, Deckard’s vector capture only tree patterns of height 1, which corresponds to capturing the type of the nodes encountered. Thus, Deckard would produce the same characteristic vector for $(x.y)\rightarrow z$ and $(x\rightarrow y).z$ since they have the same number of tree nodes for each type of tree nodes. Spinfer’s vector generation differentiates the two cases by capturing tree patterns of height more than 1: the Spinfer vector of $(x.y)\rightarrow z$ would have a 1 at the index of the tree pattern corresponding to a record followed by a record pointer, which has a different tree pattern than a record pointer followed by a record. The current implementation’s maximum height is 6, value fixed empirically to match the maximum height of most AST nodes. The higher the maximum height of patterns captured, the more differentiation Deckard can have, but also the costlier the vector generation. We can also tune Deckard’s vector clustering with a similarity parameter ranging in $[0, 1]$, which expresses the amount of strictness we require for the similarity of vectors inside a same cluster. If the parameter is 0, then all vectors are similar, and if it is 1, then only identical vectors are similar. We fixed empirically this parameter to 0.95 (which is also the default value chosen by Jiang et al. in Deckard sample configuration files).

Spinfer then flags the CFG nodes corresponding to any characteristic vector signaled by Deckard as being part of any cluster as interesting, along with all of the modified nodes (even if they do not occur in clones). Each group of contiguous non-interesting nodes is then transformed in one \dots node that have the same meaning as in SmPL: all the nodes contained in all the execution paths between each predecessor and the successor of the \dots nodes are represented by the \dots . We call this removal of nodes the *skimming* of a CFG; the result is illustrated in Figure 2c.

2.2 Extracting common patterns

The semantic patches we want to produce are made of a sequence of lines of code separated by \dots , similar to Spdiff’s output illustrated in Listing 4. These lines of codes, which possibly contain metavariables, are called *patterns*. We say that a pattern matches a CFG node if the AST of the pattern and the AST of the CFG node are the same; a metavariable is the same as any subtree. For instance, $\mathbf{x0=f00(x1\rightarrow x2)}$ is a pattern, as well as $\mathbf{x0=x1(x2)}$. Both may match $x = f00(y\rightarrow z)$. The second pattern is said to be more general than the first one because all the nodes matched by the first one are matched by the second one. We can also say that the second pattern matches the first pattern.

2.2.1 Abstraction of patterns

We want Spinfer to produce safe semantic patches: it means that if we apply the semantic patch output by Spinfer on the same modified code it has been inferred from, it should not perform any modifications that are not part of the original set of modifications. The difficulty when building patterns is to get to the right level of abstraction to represent multiple modifications in a single pattern while ensuring the safety property. Andersen et al. [1] addresses this issue by computing all the different possibilities of abstraction of a semantic patch and choosing the safe pattern that is the most concise. Even with the

pruning strategies implemented by Andersen et al., we found empirically that this algorithm does not terminate in many cases in a reasonable amount of time. But in this section, we deal with patterns and not whole semantic patches. Since the safety of the semantic patch depends on the whole semantic patch and not on each of its patterns individually, the pattern mining algorithm presented in the next section to generate patterns does not consider the safety of the patterns it produces. For instance, the pattern `x0=x1(x2)` alone may be unsafe because it would match a too large number of nodes in the original code; but if we add the pattern `foo(x0)` as a context, then the whole semantic patch may become safe.

The abstraction strategy we take targets conciseness; the pattern mining algorithm presented in the next section can be seen as a provider of raw pattern material to the next phase of semantic patch inference which will actually produce semantic patches that are safe.

2.2.2 Pattern mining

We have a forest of ASTs, each corresponding to a single CFG node, and want to extract patterns from it. The details are in Algorithm 2. We first run the same clone detector as in Section 2.1.3 on the forest with all identifiers and constants abstracted, to determine the groups of similar ASTs. Then we compare each pair of ASTs in each group to find the Maximum Common Embedded Subtree (MCES) using an algorithm by Lozanno et al. [9]. A tree t_1 is said to be embedded in another tree t_2 if we can transform t_2 in t_1 by a series of edge contractions. An edge contraction is the action of merging together two nodes of the tree separated by one edge. An example of edge contraction is provided in Figure 3. The MCES is the embedded subtree that has the maximum number of edges.

Lozanno et al.’s MCES algorithm is based on edge contractions in the tree: the algorithm performs a parallel pre-order traversal of the AST and contracts edges that are different until only the common edges remain.¹ In the implementation of the algorithm, the labels are carried by the edge whose child is the node rather than the nodes themselves. That is why contracting an edge means destroying the child node of this edge.

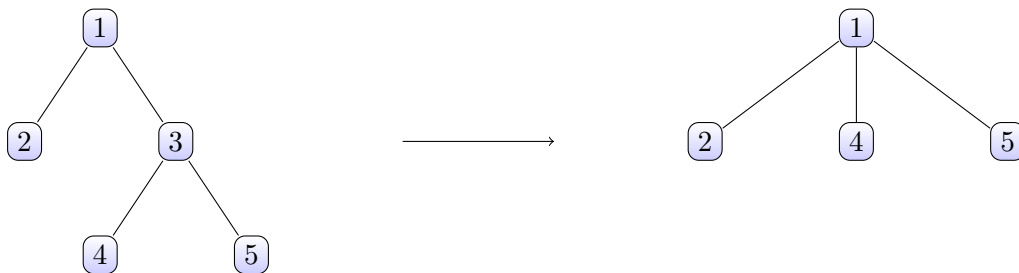


Figure 3 – Examples of trees before and after edge contraction. Here, the edge between node 1 and 3 has been contracted.

An advantage of the MCES algorithm is that it can accurately detect all kinds of similarity between trees: trees that share a common root part but have a different subtree near the leaves, or trees that share common subtrees near the leaves but have different root parts. Indeed, the difference between the two scenarios is the location of the edges contracted by the MCES algorithm: for instance, if edges near the root are contracted, the common subtree found will correspond to a common part near the leaves. Thanks to this, Spinfer is able to identify patterns of different granularities: one pattern can

¹See [9] for an accurate description of the algorithm.

Algorithm 2 The pattern mining algorithm

Input: $\{t_1, \dots, t_n\}$ is a set of ASTs.

Output: A set of patterns matching trees of the input.

```

1: function MINEPATTERNS( $\{t_1, \dots, t_n\}$ )
2:    $p, \{t'_1, \dots, t'_p\} \leftarrow$  EXTRACTMAXIMALPATTERN( $\{t_1, \dots, t_n\}$ )
3:    $p \leftarrow$  REFINEPATTERN( $p$ )
4:   if  $p=0$  then
5:     return  $\emptyset$ 
6:   else
7:      $F \leftarrow \{t_1, \dots, t_n\} \setminus \{t'_1, \dots, t'_p\}$ 
8:     return  $\{p\} \cup$  MINEPATTERNS( $F$ )
9:   end if
10: end function
11: function EXTRACTMAXIMALPATTERN( $\{t_1, \dots, t_n\}$ )
12:    $\{g_1, \dots, g_r\} \leftarrow$  GROUPTREESBYCLONES( $\{t_1, \dots, t_n\}$ )
13:    $P \leftarrow \emptyset$ 
14:   for all  $g$  in  $\{g_1, \dots, g_r\}$  do
15:     for all  $(t'_1, t'_2)$  pair of elements of  $g$  do
16:        $p \leftarrow$  FINDMCES( $t'_1, t'_2$ )
17:        $P \leftarrow P \cup p$ 
18:     end for
19:   end for
20:    $p', \{t''_1, \dots, t''_s\} \leftarrow$  MAXIMALMATCHESANDMATCHEDNODES( $P$ )
21:   return  $p', \{t''_1, \dots, t''_s\}$ 
22: end function

```

match a the variable x that can appear in any kind of CFG node, but another pattern can match only the CFG nodes that are if statements. The granularity can also be mixed, for instance with a pattern matching all the if statement which condition contains the variable x . This last pattern is noted `if (<+...x...+>)` in SmPL and this construction is called a *nested pattern*.

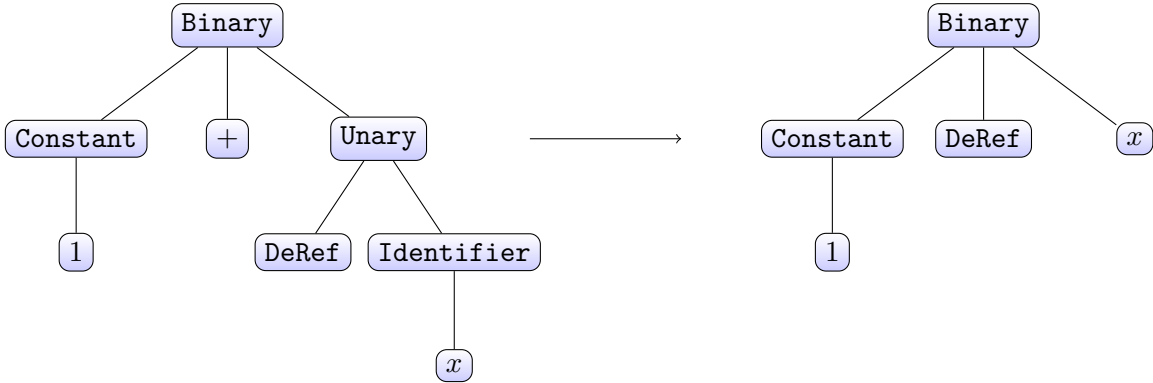


Figure 4 – `Identifier` and `Unary` have been contracted. The right tree is not syntactically correct and cannot be serialized into proper C code.

Nevertheless, edge contraction in the AST does not preserve syntactic correctness as illustrated in Figure 4. Thus, not all the MCES found can become patterns. That is why we match all the MCES found against the trees to select the MCES that matches the greatest number ASTs among our inputs. Our definition of matching is the following: a pattern matches an AST if the pattern is a strict subtree of the AST it is matched against; a metavariable node can represent any subtree rooted at its position, and nested patterns are matched against the sub-AST of the parent AST. Edge contractions might result in splitting a single tree into a forest. We discard the results of MCES that are forests. This definition of matching preserves all the granularities of patterns provided by the MCES algorithm, including the nested patterns.

It is important to understand what is the impact of grouping the nodes using a clone detector before running the MCES algorithm. Indeed, as we run the MCES algorithm on every distinct pair of trees, the cost of the operations is $O(n^2)$ where n is the number of trees (and CFG nodes since the trees are the CFG nodes). The grouping phase aims at reducing the cost of the pattern mining since we run the MCES algorithm on the pairs of each separate group. If there are \sqrt{n} groups of size \sqrt{n} for instance, the new cost is $O((\sqrt{n})^2 \times \sqrt{n}) = O(n\sqrt{n})$ instead of $O(n^2)$. But grouping the nodes according to their similarity decided by the clone detector has a complicated effect on the output patterns. Indeed, the clone detector will tend to group together CFG nodes of the same type: variable assignments with variable assignments, if conditions with if conditions, etc. But if the code modification is for instance renaming a variable, this variable could occur in any type of CFG node and the grouping will not be relevant. Instead of capturing only the variable to rename, the output pattern will also capture the different common types of nodes in which the variable occurs. Because we think that handling complex modifications involving large parts of the individual CFG nodes that are modified is the primary purpose of Spinfer, rather than small modifications like renaming a variable, we group the nodes by default. But the implementation leaves an option for not grouping the nodes, at the price of a longer execution time and perhaps too general patterns if the nodes given as input are very different. One can also change the value of Deckard’s similarity parameter mentioned in Section 2.1.3, that has an effect on the similarity of nodes inside the groups formed by Deckard.



The patterns yielded by MCES have all their identifiers abstracted. For instance, if we have the following CFG nodes as an input of the MCES algorithm:

```
1 var2 = foo(other_var1->y,x&another_var1);
2 var2 = foo(other_var2->y,x&another_var2);
```

then the pattern output will be $X0=X1(X2->X3,X4\&X5);$.

Afer the MCES algorithm, we proceed to match the patterns found against the CFG nodes given as input to the pattern mining algorithm. During this process, the metavariables in the patterns are filled with all the concrete subtrees they match. When a metavariable always matches the same subtree every time, we replace it by the subtree: for instance, our pattern will become $X0=foo(X1->y,x\&X2);$. The function `REFINEPATTERNS` of Algorithm 2 performs this process.

The same MCES algorithm is used in by Kim et al. in [10]; nonetheless our approach is different. Since Kim et al. use MCES on an AST whose nodes are statements of code, forming some kind of non-cyclic CFG (see Figure 5 for an example). We use the MCES on a fine-grained AST to detect similar code (see the left tree of Figure 4 for an example), whereas Kim et al. perform this operation by comparing tree edit operation sequences.

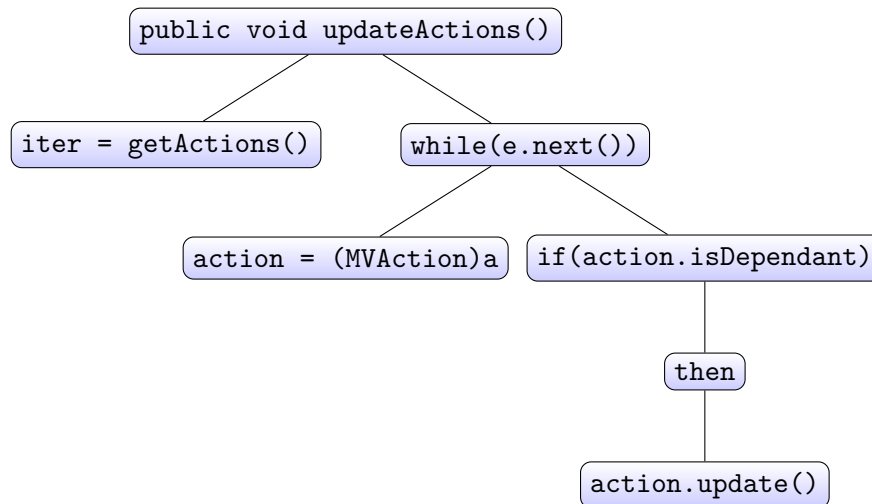


Figure 5 – Example of an AST used by LASE [10]

2.2.3 Abstraction rules

In addition to the “make concrete metavariables that always match the same subtree”, the `REFINEPATTERNS` function also implements custom abstraction rules. Indeed, if we know beforehand that our modification will affect only a certain identifier, we can tell Spinfer never to abstract this identifier in the patterns. This mechanism is meant to compensate for possible insufficient amount of sample code to feed to Spinfer, in order to get more accurate results even if the code contains too little information to make the right inference. The rules are expressed by specifying a sequence of nodes types traversed in the AST before arriving at the node that should not be abstracted. For instance, the current custom abstraction rules currently implemented in the `REFINEPATTERNS` function are to keep concrete all the function names and the names of record fields and global variables. Indeed, these identifiers correspond to global structures that are likely to be the context of code modifications.

When a custom abstraction rule indicates to keep concrete a node that otherwise should have been abstracted (were the custom abstraction rules not present), one solution would be to create a new pattern where this node is concrete. Instead, we prefer to turn the metavariable containing the identifier to be kept concrete into a disjunction of cases. Indeed, disjunction of cases is supported by SmPL and makes it possible to make semantic patches more concise. To illustrate this effect, a pattern $\mathbf{X0}(y,0,\mathbf{X1})$ is a call to the function whose name is abstracted by $\mathbf{X0}$. As we want to keep function names concrete, we look what is matched by $\mathbf{X0}$ and find that it can be either the function `foo` or the function `bar`. So the new pattern becomes Listing 6. The notation in this example means that $\mathbf{X0}$ is a disjunction of cases whereas $\mathbf{X1}$ can match anything.

Listing 6 – Example of pattern output by Spinfer.

```

1 Pattern matching 9 nodes:  $\mathbf{X0}(y,0,\mathbf{X1})$ 
2 Disjunctions:
3  $\mathbf{X0}$  =
4     | foo (3 matches)
5     | bar (6 matches)

```

There is potentially an issue with multiples variables having disjunctions in the same pattern: if $\mathbf{X0}$ can be a or b and $\mathbf{X1}$ can be c or d can we have all the elements of $\{a, b\} \times \{c, d\}$ for the disjunction? The answer is provided by the matching information in the metavariables. Since we know which case of the disjunction in the metavariables match which node of the CFG, we can only include in the global pattern disjunction combination that actually occur in our input code.

As of now, the custom abstraction rules cannot be modified by the user (except enabling and disabling them) but the goal is to read them from a configuration file (for instance the `.cocciconfig` already used by Coccinelle), which would be useful for custom rules that don't change but are specific for a project.

2.2.4 Usage

Note that the pattern mining technique we have developed applies to any forest of ASTs given as input. The pattern mining will be used in several occasions on the next phases. Moreover, the level of abstraction in the output patterns can easily be changed as the metavariables record all the occurrences of concrete node they match. For instance, in Listing 6, the numbers of nodes matches are computed as the cardinality of the set of portions of AST matched by each metavariable (or the whole pattern). In the example, $\mathbf{X1}$ would for instance match 3 times `z->w`, 2 times `z->f` and 4 times `NULL`. Since we keep track of what matches what, we will be able to use this information to refine the patterns inside of the semantic patches. For instance, if the pattern $\mathbf{X0}(y,0,\mathbf{X1})$ is placed inside of a sequence of patterns that matches only portions of code where the metavariable $\mathbf{X1}$ stands for `NULL`, we will be able to replace $\mathbf{X1}$ by `NULL`.

2.3 Building sequences of patterns

As explained in Section 3.1, all the algorithms described below are not yet implemented (Spinfer is a work in progress). We will describe in the next sections the ideas that will guide the future implementation, but the conditional used in certain sentences indicates choices that could be changed at implementation time.

We now dispose of a function to determine the pattern that matches the maximum number of CFG nodes inside a given set of CFG nodes. As the semantic patches we want to produce are sequences



of patterns separated by \dots , we have to build sequences of patterns (including patterns of modified nodes) that occur a maximum number of times in the initial code base. The algorithm proposed here is inspired from Andersen’s work [2] in the sense that we will grow the sequences of patterns iteratively, but we will adopt a more comprehensive approach: indeed, Andersen’s Spdiff searches for sequences of patterns inside the “before” code, then tries to pair the modifications with these patterns of “after” code; whereas the merged CFG data structure allows Spinfer to directly build patterns that include removed and added nodes. In this section, we work with the skimmed CFG described in Section 2.1.3.

2.3.1 Ordering on CFG nodes

Since we want to build ordered sequences of patterns separated by the SmPL \dots which means “for all execution paths”, we associate with each pair (n_1, n_2) of CFG nodes a boolean $D(n_1, n_2)$ which is true n_2 dominates n_1 , *i.e.* if every path in the graph from n_1 must go to n_2 ; and false otherwise. Dominance is a well-known notion used in compilation and a variety of algorithms to compute the dominance of all the nodes in a flow-graph are described by Goergidis et al. [4]. We could use the simple iterative algorithm despite its $O(n^2)$ cost if we find empirically that it does not cause a performance bottleneck.

2.3.2 Growing pattern sequences

The growing of pattern sequences is done recursively and described by Algorithm 3. We initialize the sequence with a pattern extracted from the modified nodes.

Suppose now that we have a sequence of patterns S and a new pattern that we want to add to the sequence. For instance, we will take $(P_1, P_2) = \mathbf{x} = \text{foo}(\mathbf{x0} \rightarrow \mathbf{y}); \dots \text{bar}(\mathbf{x}, \mathbf{x1});$ as our pattern sequence and $P_3 = \mathbf{x2} = \text{baz}(\mathbf{x});$ as the new pattern to add to the sequence. We try to determine if this new pattern can be inserted to the existing sequence; to check this we look at each of our matching occurrences, which are sequences of nodes. If each node n that is matched by P_3 (for instance $n = \text{var} = \text{baz}(\mathbf{x});$) is such that for every (n_1, n_2) match of the sequence of patterns (P_1, P_2) (for instance $(n_1, n_2) = \mathbf{x} = \text{foo}(\mathbf{a} \rightarrow \mathbf{y}); \dots \text{bar}(\mathbf{x}, \mathbf{0});$), $D(n_1, n) = \text{true}$ and $D(n, n_2) = \text{true}$, then we say that we can add the new pattern between P_1 and P_2 in S . For instance, if P_3 matches nodes like $\text{var} = \text{baz}(\mathbf{x});$ that are always between nodes matched by P_1 and nodes matched by P_2 , then we can add P_3 to the sequence which becomes (P_1, P_3, P_2) . If we want to insert the new pattern at the beginning or at the end of the sequence, then we would verify a similar condition with only one test instead of two.

If it is possible to insert the new pattern in the sequence of patterns, then we do it. Then we proceed to search for a new pattern to insert in the sequence. To obtain a new pattern to add, we match the sequence of patterns on our set of nodes, then run a pattern extraction on all the nodes not matched by the pattern (*i.e.* not in any matching occurrence) and start again recursively the algorithm with the new sequence (which is strictly longer than before) and the new pattern to add.

If it is not possible to insert the new pattern in the sequence of patterns, then we try the insertion process again with new patterns formed from the pattern by turning one of its metavariables into its concrete components. For instance P_3 would become $\mathbf{z} = \text{baz}(\mathbf{x})$. If one or more of these new patterns fits, then we take the one that matches the more occurrences in the code base, and look for another pattern to add the same way than before. If none of the more concrete version of the original pattern to add matches, then we cannot grow the sequence more and we stop the algorithm.

As the growing sequence algorithm does not backtrack (it only grows sequences of patterns), the first pattern given as initialization to the algorithm must be at the right level of abstraction to trigger the correct growing of the sequence. For instance, if we start with the pattern $\mathbf{x0} = \mathbf{x1}$ which is too general, then the whole sequence is going to be too general. That is why the sequence should be

initialized with a pattern inferred from the modified nodes : the pattern extraction algorithm works better at finding the right level of abstraction for the pattern it output if its input data has only one identifiable pattern inside them. We assume that the semantic patch we want to find contains some patterns of modified nodes.

Algorithm 3 Growing pattern sequences algorithm

Input: A set \mathcal{N} of CFG nodes, modified or not, and a function $E : (n_1, n_2) \rightarrow \{\text{true}, \text{false}\}$ indicating the presence of an execution path between two nodes.

Output: A sequence of patterns which matches the maximum of nodes.

```

1: function GROWINGPATTERNSEQUENCE( $S$  : sequence of patterns,  $p$  : pattern to add)
2:   if NEWPATTERNFITSINTHESEQUENCE( $p, S, \mathcal{N}$ ) then
3:      $S' \leftarrow$  ADDNEWPATTERNTOSEQUENCE( $p, S$ )
4:      $p' \leftarrow$  FINDMAXIMALPATTERN( $\mathcal{N} \setminus \{\text{nodes matched by } S\}$ )
5:     return GROWINGPATTERNSEQUENCE( $S', p'$ )
6:   else
7:     for all patterns  $p'$  more concrete than  $p$  do
8:       if NEWPATTERNFITSINTHESEQUENCE( $p', S, \mathcal{N}$ ) then
9:         if  $p'$  matches more than  $p'_m$  then
10:           $p'_m \leftarrow p'$ 
11:        end if
12:      end if
13:    end for
14:    if  $p'_m$  exists then
15:      return GROWINGPATTERNSEQUENCE( $S, p'$ )
16:    else
17:      return  $S$ 
18:    end if
19:  end if
20: end function

```

2.4 Producing semantic patches

We extract the sequence of patterns that matches the maximum number of occurrences thanks to the previous algorithm; then we remove from the CFG node set the nodes matched by the sequence of patterns and can run the algorithm again to find another sequence of pattern, until the sequences found don't match any occurrences (because the CFG node set does not contain any repeating pattern any more). Note that the growing sequence algorithm produces a sequence of patterns that can be either patterns of modified nodes or patterns or context nodes. With this approach, added code patterns, removed code patterns and context code patterns are considered the same in the sequence of patterns.

We ensure that there is always a modified code pattern in the sequence by initializing it with a modified code pattern. But our algorithm could as well be used produce semantic patches with no modified code, *i.e.* the semantic patterns computed by Andersen et al. in [2]. The safety of the sequences of patterns can be ensured by the growing pattern sequences algorithm: a pattern can be added to the sequence only if the new sequence of patterns is safe relatively to the code not modified in the input codebase.



Then we proceed to turn the sequences of patterns into proper semantic patches. This is done by looking at all the metavariables in the sequences of patterns and merging together the metavariables that always match the same thing when inside an occurrence of a pattern sequence match. This process links the patterns in the sequence. We also give the metavariables a certain type depending on their contents: if a metavariable matches only expressions of a certain type, then the metavariable will be assigned this type.

Then the last phase is the serialization of the sequences of patterns into SmPL.

3 The Spinfer program

3.1 Implementation

Spinfer is implemented in OCaml and uses the C parsing library of Coccinelle. As of now, with 6,000 LOC, the tool is able to parse the code coming from specific files or git commits, build the merged and skimmed CFG² and extract patterns from any given set of CFG nodes. The program is fully documented with `ocamldoc`.

In order to be fully functional, the following features need to be added to the implementation:

- support of nested patterns;
- the growing pattern sequences algorithm;
- a semantic patch safety checker;
- the semantic patch serialization in SmPL format.

3.2 Current results

We tested the program with several commits from the Linux kernel git that were implemented by a manually written Coccinelle semantic patch, as to compare the results of Spinfer with the semantic patches.

For instance for the commit [323de9ef](#), implemented (for the majority of its modifications) by the semantic patch of Figure 1, Spinfer identifies the following patterns among the modified nodes of the CFG:

```

1 ===== Removed code patterns =====
2
3 Pattern matching 41 nodes: if (!X0->X1)
4 Disjunctions:
5 X1 =
6   | pmx (1 match)
7   | pinctrl (1 match)
8   | pctrl_dev (1 match)
9   | pctrl (4 matches)
10  | pctldev (5 matches)
11  | pctl_dev (6 matches)
12  | pctl (20 matches)
13  | pcdev (1 match)

```

²As of now, the part of the skimming after the nodes have been tagged as interesting need to be reimplemented.


```

14     | ctrl (2 matches)
15
16 Pattern matching 37 nodes: return -X0
17
18 Pattern matching 7 nodes: X0 = -X1
19
20 ===== Added code patterns =====
21
22 Pattern matching 42 nodes: if (IS_ERR(X0->X1))
23 Disjunctions:
24 X1 =
25     | pmx (1 match)
26     | pintrl (1 match)
27     | pctrl_dev (1 match)
28     | pctrl (4 matches)
29     | pctldev (5 matches)
30     | pctl_dev (6 matches)
31     | pctl (21 matches)
32     | pcdev (1 match)
33     | ctrl (2 matches)
34
35 Pattern matching 35 nodes: return PTR_ERR(X0->X1)
36 Disjunctions:
37 X1 =
38     | pctrl_dev (1 match)
39     | pctrl (4 matches)
40     | pctldev (4 matches)
41     | pctl_dev (6 matches)
42     | pctl (17 matches)
43     | pcdev (1 match)
44     | ctrl (2 matches)
45
46 Pattern matching 7 nodes: X0 = PTR_ERR(X1->X2)
47 Disjunctions:
48 X2 =
49     | pmx (1 match)
50     | pintrl (1 match)
51     | pctldev (1 match)
52     | pctl (4 matches)
53
54 Pattern matching 3 nodes: return ERR_PTR(-X0)
55
56 Pattern matching 3 nodes: return X0(X1)
57 Disjunctions:
58 X0 =
59     | PTR_ERR (2 matches)
60     | ERR_PTR (1 match)
61
62 Pattern matching 2 nodes: if (IS_ERR(X0))

```

The abstraction rules have been set to keep concrete all identifiers of field records and function name. We can observe that some patterns are more general than others, even if it is written that they



match fewer nodes: that is because the pattern mining algorithm removes from the input the nodes matched by one pattern before searching for another. These results obviously have to be refined to form semantic patches but their resemblance with the original semantic patch is encouraging.

3.3 Performance analysis

We profiled Spinfer with the contents of commit [323de9ef](#) of the Linux kernel as input. The real execution time is 1 m 30 s, and the percentages of time spent in notable functions are:

- 50% for Deckard's vector generation and clustering;
- 10% for parsing the code;
- 20% for the MCES algorithm;
- 2% for building the merged CFG and skimming it.

The input code is a unfavorable scenario for time performance since the commit modifies 45 files of the `drivers` folder and features more than a hundred modified CFG nodes. Nevertheless, the total execution time is reasonable and the major bottleneck is the clone detector, which was expected because we use it to filter the data before using finer-grained algorithms.

4 Conclusion

Spinfer is a semantic patch inference tool designed to provide the most useful results for the C developers to quickly perform collateral evolution or understand the contents of a commit. Its workflow, inspired by LASE [10] and Spdiff [2], uses a merged control flow graph of the input code and searches first for patterns among the nodes, then grows sequences of patterns with the `... SmPL` operator as separator. The algorithms works at the level of the individual CFG nodes or the whole graph, and target a reasonable execution time. The level of abstraction of inferred semantic patch is decided part in the pattern extraction process and part in the sequence growing process. Spinfer is currently half-implemented, amounting to already 6,000 lines of OCaml code.

References

- [1] Jesper Andersen and Julia L. Lawall. Generic patch inference. *Automated Software Engineering*, 17(2):119–148, 2010.
- [2] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 382–385, New York, NY, USA, 2012. ACM.
- [3] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.
- [4] Loukas Georgiadis, Robert E. Tarjan, and Renato F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications*, 10(1):69–94, 2006.
- [5] Lingxiao Jiang, Ghassan Mishserghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.
- [6] T. Kamiya, S. Kusumoto, and K. Inoue. Cfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.
- [7] Miryung Kim. *Analyzing and inferring the structure of code change*. ProQuest, 2008.
- [8] Julia Lawall, Quentin Lambert, and Gilles Muller. Prequel: A Patch-Like Query Language for Commit History Search. Research Report RR-8918, Inria Paris, June 2016.
- [9] Antoni Lozano and Gabriel Valiente. On the maximum common embedded subtree problem for ordered trees. *String Algorithmics*, pages 155–170, 2004.
- [10] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.
- [11] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/S/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 247–260, New York, NY, USA, 2008. ACM.
- [12] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 59–71, New York, NY, USA, 2006. ACM.
- [13] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Smpl: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electron. Notes Theor. Comput. Sci.*, 166:47–62, January 2007.



- [14] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.