Automatic inference of system software transformation rules from examples

Lucas Serrano

Automatic Inference of System Software Transformation Rules from Examples

---

Lucas Serrano

Jury members:

Tegawendé F. Bissyandé, Senior Research Scientist, University of Luxembourg	*Reviewer*
Martin Monperrus, Full Professor, KTH Royal Institute of Technology	*Reviewer*
Antoine Miné, Full Professor, Sorbonne University	*Examiner*
Valérie Issarny, Senior Research Scientist, Inria	*Examiner*
Jean-Rémi Falleri, Associate Professor, University of Bordeaux	*Examiner*
Julia Lawall, Senior Research Scientist, Inria	*Supervisor*

Résumé

Le noyau Linux est aujourd'hui présent dans tous les types de systèmes informatiques, des smartphones aux supercalculateurs, comprenant à la fois le matériel le plus récent et les systèmes "anciens". Cette diversité d'environnement a pour conséquence une base de code importante, d'une dizaine de millions de lignes de code, pour les pilotes matériels. Cependant par souci d'introduction de nouvelles fonctionnalités, ou pour des raisons de performance ou de sécurité, certaines interfaces de programmation (APIs) internes doivent être parfois revues, ce qui peut impliquer des changements pour des milliers de pilotes les utilisant.

Cette thèse propose une nouvelle approche, Spinfer, permettant d'effectuer ces migrations d'utilisation d'APIs de manière automatique. Cette nouvelle approche, basée sur l'assemblage de motifs en suivant des contraintes de flot de contrôle, est capable d'apprendre à partir d'exemples, même imparfaits, des règles de transformations adaptées aux enjeux des migrations d'utilisations d'APIs dans le noyau Linux.

Abstract

The Linux kernel is present today in all kinds of computing environments, from smartphones to supercomputers, including both the latest hardware and "ancient" systems. This multiplicity of environments has come at the expense of a large code size, of approximately ten million lines of code, dedicated to device drivers. However, to add new functionalities, or for performance or security reasons, some internal Application Programming Interfaces (APIs) can be redesigned, triggering the need for changes of potentially thousands of drivers using them.

This thesis proposes a novel approach, Spinfer, that can automatically perform these API usage updates. This new approach, based on pattern assembly constrained by control-flow relationships, can learn transformation rules from even imperfect examples. Learned rules are suitable for the challenges found in Linux kernel API usage updates.

Remerciements

La thèse est une longue aventure de 3 ans, parfois plus, remplie de travail mais surtout d'émotions, d'échecs et de réussites, de hauts et de bas. Pour mener à bien une telle aventure, sans baisser les bras, il faut être bien entouré. C'est pour cette raison que je souhaite tout d'abord remercier *Marion* qui a été à mes côtés et m'a soutenu pendant ces trois années. C'est aujourd'hui grâce à toi que j'arrive à terminer ce périple. Je remercie aussi mes collègues de bureau avec lesquels j'ai passé de très bons moments. Tout d'abord en commençant par les anciens: *Antoine*, *Damien* et *Cédric*, qui nous montrent aujourd'hui qu'il existe bien une vie après la thèse. Ensuite, *Redha* et *Darius*, avec lesquels j'ai partagé les meilleurs moments d'une thèse que sont, bien entendu, la rédaction et les procédures administratives, mais aussi beaucoup de rires et de moments surréalistes. Et enfin *Pierre*, le petit dernier, à qui je souhaite bon courage pour les deux ans à venir. Merci également à mes collègues et amis du labo: *Hakan*, *Ilyas*, *Saalik*, *Jonathan*, *Francis* et *Arnaud* avec qui j'ai passé de très bon moments et surtout de très bonnes pauses café. Je remercie également *Julien*, pour ses analyses et ses conseils aux moments où j'en avais le plus besoin. Merci à ma famille, qui m'a apporté un soutien sans faille, et merci à mes amis, dont beaucoup sont aussi passés par là, pour leur soutien et leurs conseils, mais aussi pour tous ces moments de joie partagés qui m'ont aidé à tenir durant cette thèse.

Enfin une thèse n'en serait pas une sans un travail scientifique, et en cela je souhaite remercier mon encadrante *Julia*. Merci à toi de m'avoir accompagné et suivi toutes ces années, d'avoir toujours répondu rapidement à toutes mes questions, même lorsque que je les posais à des horaires improbables. Enfin merci de m'avoir appris qu'un refus en science ce n'est pas le rejet de nos travaux mais simplement une invitation à les améliorer. Je souhaite remercier également *Gilles*, *Lingxiao*, *David*, *Ferdian* et *Stefanus* avec lesquels j'ai eu le loisir d'échanger et de collaborer dans le cadre de mes travaux, et dont les conseils se sont avérés très pertinents.

Contents

1 Introduction

From the largest supercomputers to the tiniest embedded devices, the Linux kernel is present on an overwhelmingly diverse range of platforms. This diversity of targets has come at the expense of a code size that has grown steadily over the years, reaching 12 million lines of code dedicated solely to device drivers in early 2020.

Driver code interacts with the rest of the Linux kernel using the kernel internal Application Programming Interfaces (APIs), which defines the list of kernel functions that are publicly available. The complexity of designing an operating system kernel and the need to continually take into account new requirements has implied that the design of internal interfaces must often be revisited, triggering the need for repetitive changes among the users of these interfaces that may affect the entire source tree. As drivers maintained in the source tree must be kept functional and are seldom removed, the size and number of the needed changes to update or replace an API can discourage Linux kernel developers from doing it. Developers willing to perform those changes can introduce new bugs or only perform changes on a subset of affected files, requiring out of date APIs to be only deprecated instead of removed.

To help developers perform these large-scale changes, automation is needed. On one hand, semi-automatic tools targeting specifically the Linux kernel have been developed. Since 2008, the automatic C code transformation tool Coccinelle [LM18] has been part of the Linux kernel developer toolbox for automating large-scale changes in kernel code. Coccinelle provides a notion of semantic patch allowing kernel developers to write transformation rules using a patch-like [MES03] (i.e., diff-like) syntax. Given a semantic patch, Coccinelle applies the rules automatically across the code base. Today, even if Coccinelle is widely adopted by the Linux community there are still large-scale changes in the Linux kernel commit history where it has not been used, as writing semantic patches is not necessarily easy. On the other hand, many tools performing fully automatic API usage updates by learning transformations from examples have been developed by the software engineering community. However the large majority of these tools targets Java code, and none of them can target the transformation challenges of the Linux kernel.

In this thesis, we address the challenge of automatically learning Linux kernel API usage updates from a set of update examples. The contributions of this research are:

- A taxonomy of transformation inference difficulties, organized into five categories of challenges.
- A control-flow representation of code changes.
- A code clustering approach, optimized to cluster similar changes in a transformation.
- A novel approach for learning code transformations that focuses on control-flow relationships and generates readable transformation rules.
- An open-source implementation of our approach called SPINFER.[1]
- An evaluation of the taxonomy of transformation challenges in the Linux kernel.
- An evaluation of our novel approach for learning real code transformations in the Linux kernel.

The rest of this document is organized as follows: Chapter 2 characterizes API usage update transformations and present a taxonomy of transformation inference challenges; Chapter 3 describes the state of the art of transformation learning from examples; Chapter 4 presents our approach in this domain; Chapter 5 evaluates our approach on Linux kernel transformations and finally Chapter 6 concludes this thesis. More precisely:

- Chapter 2 first defines what are API-usage update transformations and how to describe them accurately using the semantic patch language (SmPL). It then presents the first contribution of this thesis, a taxonomy of inference challenges organized into multiple categories of inference difficulty.
- Chapter 3 first presents existing automatic approaches for learning transformations from examples. For each approach we analyze its inference capabilities in terms of the taxonomy. Then, this chapter presents other related work in semi-automatic API usage updates inference and related work for learning other kinds of transformations.

[1] Sources available at: `https://gitlab.inria.fr/spinfer`

- Chapter 4 details our approach for learning transformation from multiple examples. We start by describing how to represent input examples as modifications in a control-flow graph. Then, we will see how to cluster terms in these modifications, to form abstractions representing multiple concrete terms. Next we will detail how to assemble abstractions using control-flow properties found in the examples. Finally, we describe how to get from an assembly of abstractions to a patch in the semantic patch language.
- Chapter 5 evaluates of our approach to infer real Linux kernel changes. We analyze multiple metrics to describe how well our approach covers the changes to be made and analyze some of the failures of our approach. We also compare semantic patches inferred by our approach to human-written semantic patches and analyze the efficiency of our approach.
- Chapter 6 concludes this thesis with a summary of our work and findings. We discuss the strengths and weaknesses of our approach and future work complementing this approach. Then, we will present some perspectives for the future of automatic software maintenance.

2 Understanding API usage update transformations

In this chapter we will try to understand what are API usage updates transformation and what challenges these transformations bring. To do so, we will first see what is an API usage update transformation, then how to precisely describe code transformations without using a natural language description and finally we will see what challenges these API usage updates arise.

2.1 What is an API usage update transformation?

The transformations we want to infer from are often called collateral evolutions, systematic edits, API migrations or API usage updates. Multiple definitions are found in the software engineering literature for each of these terms:

- **Collateral evolutions** are evolutions that occur "when an evolution that affects the interface of a generic library entails modifications [...] in all library clients" [PLM07].
- **Systematic edits** are modifications that "consist of similar, but not identical, program changes to multiple contexts" [MKM11].
- **API migration** [LSC18] and **API usage update** [FXO19] are changes that appears in client code due to used API having their methods[1] changed, removed or deprecated.

All of these terms have some differences but in general they all refer to the same notion: a set of similar transformations that must happen in many places because of a change in a common interface. In this thesis we will mostly use the term *API usage update*, as it is, in my opinion, the most easily comprehensible one.

Repetitive changes, such as API-usage update, can represent a significant portion of the changes made within a project, or of the changes made in a group of software

[1] The two cited papers target Java in which APIs come from object methods, but this definition applies also to APIs in form of functions for non object-oriented programming language as well.

using the same APIs. For instance, Ray *et al.* [Ray+15] found that at least 12% of changes in the Linux kernel are non-unique. Nguyen *et al.* [Ngu+13] found that most small changes in Java project are highly repetitive and approaches that can automatically perform or recommend transformation are beneficial to handle these changes.

Moreover, for the case of API-usage updates, omitting to perform these transformations can lead to a large fragmentation of API used and usages of deprecated APIs that can potentially represent a security risk. A recent study on Android API deprecation in Android apps [MRK13] found that, in average and at any point in time, more than one fourth of Android API usages were deprecated in the studied apps. Another study on the same subject [Li+18] found that some Android APIs were not correctly marked as deprecated in the documentation or correctly annotated in the source code, preventing code analyzers for outputting deprecation warning.

Now that we know more about the kind of transformations we want to learn, we will see how to describe them accurately.

2.2 How to describe transformations?

2.2.1 When Diff is not enough

Software developers are already used to describing transformations using GNU-Diff reports, also called simply diffs. GNU-Diff "compares two files line by line, finds groups of lines that differ, and reports each group of differing lines" [MES03].

Diffs are useful to report a change between two versions of the same file, and are commonly used in conjunction with the GNU-patch tool which "takes comparison output produced by `diff` and applies the differences to a copy of the original file, producing a patched version" [MES03]. Diffs can accurately describe transformation instances such as the following API usage update that happened in the Linux kernel in 2017 [Coo]:

```
  if (!error) {
-   init_timer(&ns_timer);
+   setup_timer(&ns_timer, ns_poll, 0UL);
    ns_timer.expires = jiffies + NS_POLL_PERIOD;
-   ns_timer.data = 0UL;
-   ns_timer.function = ns_poll;
    add_timer(&ns_timer);
  }
```

```
  #ifdef FS_POLL_FREQ
-   init_timer (&dev->timer);
-   dev->timer.data = (unsigned long) dev;
-   dev->timer.function = fs_poll;
+   setup_timer (&dev->timer, fs_poll, (unsigned long)dev);
    dev->timer.expires = jiffies + FS_POLL_FREQ;
    add_timer (&dev->timer);
  #endif
```

These two diffs describe two instances of a transformation that changes the usage of the deprecated `init_timer` in favor of the more recent `setup_timer`, two functions that initialize a low-resolution timer struct, written in the C language. Since `setup_timer` already takes care of the initialization of two fields of this struct, namely `function` and `data`, the previous initialization are removed and their value are used as the second and third argument of `setup_timer`.

The diffs we just saw are a description of instances of the same transformation, with lines removed that start with a minus, and lines added that start with a plus symbol. However it is not sufficient to describe all transformation instances of the same API usage update. Indeed, a diff only describe operations on whole lines, and it is very unlikely that a single diff can describe other instances of the same transformation because they will need to have the exact same lines in the exact same order. As each instance has different variable names, spacing and so on, the two presented diffs will only match two specific instances.

On the contrary, our explanation of the changes describes the transformation in general and not only a transformation instance, but since it uses natural language, English, it can have some ambiguity and cannot be understood easily by a computer.

What we are missing here is a description that could accurately describe a whole API usage update for both humans and computers.

2.2.2 A transformation language: SmPL

In 2007 Padioleau *et al.*introduced the semantic patch language (SmPL) [PLM07], a declarative transformation language for specifying collateral evolution in the Linux kernel. [2]

[2]SmPL was built to target Linux kernel transformations but can describe transformations of any C code.

The idea behind SmPL is to describe transformations in a generic way so that the same transformation description, also called a semantic patch, applies to multiple transformation instances.

Moreover SmPL was made to be both easily comprehensible by humans and easily parsable by automated tools, by using a syntax close to a traditional diff. This language comes along with the Coccinelle engine [Pad+08; Bru+09] which takes a semantic patch and C source code as input and then searches the code for suitable locations for the transformation described by the semantic patch. If locations are found, Coccinelle can either generates diffs for all transformation instances or modify the code in place.

We will now see some core concepts of the SmPL language, illustrated by the following semantic patch performing the same transformation on timers that we have seen previously:

```
1  @@
2  expression T, D, F;
3  @@
4  - init_timer(
5  + setup_timer(
6    &T,
7  + F, D
8    );
9    ...
10 - T.data = D;
11 - T.function = F;
```

Listing 2.1: Semantic patch for the `init_timer` to `setup_timer` change

Structural matching and replacement

The first core concept of SmPL is a matching and replacement syntax that focuses on the structure of the code instead of its syntax, ignoring changes in line breaks, spacing and comments.

This is illustrated by the following portion of the previous semantic patch:

```
4  - init_timer(
5  + setup_timer(
6    &T,
7  + F, D
8    );
```

This excerpt describes a transformation that changes any `init_timer` function call to `setup_timer`, keeping the first argument the same while adding two new arguments. When passed to the Coccinelle engine, both the semantic patch and the code to match are converted to *Abstract Syntax Trees (ASTs)*, enabling to match and replace without taking into account spacing and style differences. This allows our example semantic patch to match both of our transformation instances even though each instance has different spacing.

We can see that the structural matching allows the semantic patch to focus on what changed at a fine-grained level. In our semantic patch it is clear to the reader that even if the function name of the function call changed the first argument `&T` stayed the same.

Another consequence of this structural matching is that, contrary to diff, all code described in the semantic patch must be valid C, with the exception of some SmPL special syntax that we are going to see.

Metavariables

Spacing and coding style were one source of differences between transformation instances, another major source of difference come from variable names and expression values. For example, each of our transformation instances used different terms for the timer structure: the first one used the variable `ns_timer` and the second one the expression `dev->timer`.

To abstract terms that are likely to be local to only one instance, SmPL introduces another concept: *metavariables*. Metavariables are typed abstractions that can represent a variety of things such as constants, expressions, identifiers, statements and so on.

Metavariables are declared at the start of the semantic patch, called the header, to distinguish them from real terms. The following excerpt of our semantic patch header declares 3 metavariables of type expression:

```
1 @@
2 expression T, D, F;
3 @@
```

The two consecutive at-signs `@@` mark the beginning and the end of the header part of the semantic patch.

The metavariables are then reused during the matching and replacement like in the following portion.

```
10 - T.data = D;
11 - T.function = F;
```

This will match and delete any two consecutive statements that assign a value to `data` and `function` fields of any expression. However the expression `T` matched for `data` assignment must be syntactically identical to the one matched for `function` assignment because they use the same metavariable name.

Sequence operators

The third core concept of SmPL is the sequence operators, that abstract away arbitrary code between matched term.

One example of a sequence operator is found in our semantic patch for the timer transformation, marked by the ... symbol, also call an ellipse:

```
8    );
9    ...
10 - T.data = D;
```

This symbol means that they can exist any arbitrary number of statements (even zero) between the previous term (the `init_timer` call) and the next term (the `data` field assignment). In layman's terms, the sequence operator indicates that anything (or nothing) can be present between the terms matching the patterns before and after the ... symbol. This sequence operator also adds the constraint that the term after ... must come after the term before ... in all control flow paths.

It is possible to prefix the ellipse symbol with a minus sign, to remove all statements between two terms:

```
1 @@
2 @@
3   oneFunCall();
4 - ...
5   anotherFunCall();
```

The ellipse symbol can also be used to abstract other sequences, such as function arguments as shown below:

```
1 @@
2 @@
3 - foo(...);
```

This small semantic patch removes all calls to a function named `foo` regardless of the number of arguments.

Finally SmPL also defines the `<... ...>` and `<+... ...+>` sequence operators. The first one "indicates that the pattern in between the ellipses is to be matched 0 or more times" and the second one "indicates that the pattern must be matched at least once" [Smp]. This allows a semantic patch to describe a transformation that need to capture or remove a pattern an arbitrary number of time.

Isomorphism

The last concept is isomorphism, which define terms that are semantically equivalent so matching one term is the same as matching the other one. For instance, if `x` is a pointer then `x == NULL` and `!x` are two equivalent expressions.

This concept enables to group together terms with the same semantics but with a different structure that depends only on the developer coding style. SmPL defines a list of isomorphisms which groups together terms with the same semantics. A semantic patch using a term in a group of isomorphisms is able to match other terms within the group.

Summary

With a matching independent from spacing and style, usage of metavariables to abstract specific terms, sequence operators to abstract arbitrary sequences and isomorphisms, semantic patches are able to capture transformations instead of only capturing transformation instances. SmPL patches can be used to describe transformations for both humans and computers.

2.3 Taxonomy of transformation challenges

Now that we can describe code transformation accurately with the SmPL language we will look at the challenges for inferring such transformations. When we look at existing techniques later, we will see that they can infer some API usage update transformation but not all. By assessing the challenges of transformations we will able to predict on which transformations the existing techniques will work or not.

This highlights the importance of classifying the difficulties in transformation inference in order to discriminate existing techniques capabilities. Still, to the best of our knowledge, there was no reference to the level of difficulty for the inference task in existing work, making existing techniques very hard to compare.

As the first contribution of this thesis, we suggest a taxonomy of transformation inference difficulty. This taxonomy will allow us to discuss the existing work, which we will see next, from a higher-level point of view.

2.3.1 What kinds of challenges?

Difficulties in automatic transformation inference from examples can come from various aspects. When faced with examples of a transformation one has to wonder:

- What is the control flow associated with the transformation?
- What is the data flow associated with the transformation?
- How many transformations the examples contain?
- How many transformation variants exist in the example?
- Are all the examples correct and relevant?

From these questions we classify inference difficulties in the following five categories:

1. Control flow dependencies
2. Data flow dependencies
3. Number of change instances per function
4. Number of change variants
5. Presence of changes not possible to infer

These categories and their levels of difficulty are described in detail in the following sections.

This taxonomy describes the difficulty of a set of examples from which a tool is supposed to infer a transformation. In a set of examples not all examples have the same level of difficulty. To define the difficulty of the set we will take the maximum difficulty encountered in the set. We will illustrate each level of difficulty in each category with some semantic patches or transformation instances, when possible.

Although it is not necessary to use semantic patches to perform a transformation, they offer a comprehensible description of the semantics and the constraints of the transformation.

2.3.2 Control-flow dependencies

The first category of our taxonomy is control-flow dependencies. Understanding control-flow dependencies is essential, as control flow explains in what order the different part of a program are executed. As a consequence, understanding control flow enables to understand how a transformation will affect the execution of a program.

Examples can exhibit control-flow constraints for performing transformations, such as requiring that a change take place only after another, requiring the change to be in an unmodified `if` branch and so on. The level of difficulty varies in terms of the quantity of control-flow information to retrieve in order to perform the correct transformation. The required quantity of information ranges from no control-flow information to control-flow information spread across multiple functions.

No control-flow dependencies

The first level of difficulty is when there is no control-flow dependency in the changes observed in the examples. In that case there is no control-flow information to retrieve in order to perform the required changes.

This case happens when all changes consist in either:

- Removal of independent terms.
- Replacement of a term by another one.

We will illustrate the second case with a semantic patch example:

```
1 @@ @@
2 - old_function
3 + new_function
4   (...)
```

In this example we are simply replacing all calls to `old_function` with calls to `new_function`, without any control-flow constraint on when this change should happen.

Control-flow dependencies between changed terms only

The next level of difficulty is when changed terms have control-flow dependencies between them. For this case we have to retrieve some control-flow information but it is very limited because it is bound to changed terms.

The case is very frequent because it appears every time there are at least two removed connected terms in the change. An example is illustrated by the following semantic patch:

```
1 @@ @@
2 - function1(...);
3 - function2(...);
```

In this semantic patch, both calls to `function1` and `function2` get removed at the condition that the call to `function2` immediately follow the call to `function1`. Consequently there is a control-flow dependency between the removal of `function2` and the removal of `function1`.

> **!** Note that, in the previous example, the control-flow dependency is only involved if we consider the granularity of a term to be at most one statement. If we consider the two consecutive statements as one changed term, the control-flow dependency relationship disappears.

Another example is the following:

```
1 @@ @@
2 - function1(...);
3   ...
4 - function2(...);
```

In this case the control-flow dependency between the removed calls is explicit and given by SmPL ... sequence operator.

Control-flow dependencies between changed and unchanged terms

Changes get harder to infer when they require retrieving some information about unchanged terms in order to perform the correct transformation. The next level of difficulty is when there exist some control flow dependencies between changed and unchanged terms.

The following listing is an example of such a transformation:

```
@@ @@
  unmodified_function(...);
  ...
- removed_function(...);
```

In this transformation the function call to `removed_function` is removed, if and only if, it is preceded by a function call to `unmodified_function`.

This is more difficult because examples of transformations are often presented at function or file level and so they contain much more unmodified terms than modified terms. As a consequence knowing if some modified term must precede or follow one or multiple unmodified terms is hard, and finding the correct unmodified terms is harder.

Inferring transformations requiring control-flow dependencies to unmodified terms can often lead to learn incorrect transformations that have either wrong constraints or not enough constraints. To understand why this happen let us suppose that we want to perform the transformation given as an example. In this case the inference tool has several choices.

First the tool can consider that the transformation does not require any unmodified terms, leading to the following transformation, which has not enough constraints: the examples:

```
@@ @@
- removed_function(...);
```

Or the inference tool can look for common unmodified terms in all examples, but it has to carefully select a relevant subset. If it selects the wrong subset it can end up inferring the following transformation, which have the wrong constraint:

```
@@ @@
  unrelated_function(...);
  ...
- removed_function(...);
```

Negative control flow dependencies

The next challenge level is when the applicability of the transformation depends on negative information. This means that the transformation must be performed, if and only if certain terms are not present in the control flow. This is much more difficult than the previous level, because it ideally requires to have examples where the term is absent and the change is made, and similar situations where the term is

present but the change is not made. If the training set contains only examples where the change is made then it can be impossible to infer what must be absent for the transformation to be valid.

This case happens regularly when dealing with assignment. For instance, the following patch has negative control flow dependencies:

```
@@
expression ValueFromImportantCall, OtherValue;
@@
 ValueFromImportantCall = important_call(...);
  ... when != ValueFromImportantCall = OtherValue;
- old_function(ValueFromImportantCall);
```

In this example we want to delete all `old_function` calls which use a stored result of a `ValueFromImportantCall` call. As a consequence, we want to make sure the expression used to store the result is never reassigned.

Interprocedural control-flow dependencies

The last level of difficulty for control flow is when some transformation terms have control-flow dependencies to terms in other functions, namely interprocedural control flow-dependencies.

For this case the number of unchanged terms to consider explode, and it is very hard to know what are the relevant unchanged terms to check when performing the transformation.

> **!** It is not possible to give a semantic patch example for this case because SmPL does not have a syntax for interprocedural control-flow dependencies within a rule.

2.3.3 Data-flow dependencies

Our second category of inference challenges focuses on data-flow dependencies. The data flow refers to the way in which variables and expressions are related to each other, which is important to understand a transformation. In this category, the difficulty levels range from no data-flow information to retrieve to data-flow information to retrieve from both modified and unmodified terms.

No data-flow dependencies

The easiest case is when no data-flow dependencies need to be considered in order to perform a transformation.

No data-flow dependencies happens naturally when there is only one piece of data in the transformation, like in the following example:

```
@@
expression E0;
@@
- old_function(E0);
```

This also happens when the changed terms are not required to share any data as seen in this example:

```
@@
expression E0, E1;
@@
- one_function(E0);
- another_function(E1);
```

> **!** In the above semantic patch, `E0` and `E1` can be equal in some examples but they are not required to.

Finally this can happen when there is simply no data at all to consider:

```
@@ @@
- old_function();
+ new_function();
```

Similarly to what we saw for control flow, some cases can have either no data-flow dependencies or have some data-flow dependencies depending on the granularity of what we call a term.

For instance, for the same change, the following patch contains a function call having data-flow dependencies with its replacement:

```
@@
expression E0, E1, E2;
@@
- old_function(E0, E1, E2)
+ new_function(E0, E2)
```

This only happens because the granularity of the replacement is at least one function call. If we express the transformation at a more fine-grained level, as in the following semantic patch, then the data-flow dependencies are not needed:

```
@@
expression E0, E1, E2;
@@
- old_function(
+ new_function(
  E0,
- E1,
  E2)
```

Data-flow dependencies between changed terms only

The second level of difficulty for data-flow relationships is the presence of data-flow dependencies between changed terms. Being restrained to changed terms limits the set of relationships that have to be considered.

The following semantic patch is an example of data flow relationship between changed terms only:

```
@@
expression E0;
@@
- old_function(E0);
- another_old_function(E0);
```

In this semantic patch the removal of `old_function` and `another_old_function` calls only takes place if their arguments are the same, which is a data flow constraint.

Data-flow dependencies between changed and unchanged terms

The third level is data flow dependencies between changed and unchanged terms. Similarly to control flow, the amount of data flow dependency to track becomes much higher when considering unchanged terms.

One example of this level is the following semantic patch:

```
@@
expression E0;
@@
  important_function(E0);
- function_to_remove(E0);
```

This level of difficulty can happen very often if added terms require arguments not appearing in removed terms. Indeed the following semantic patch is not valid:

```
@@
expression E0;
@@
  unmodified_function();
+ new_function(E0);
```

In this case we must find a value for the new argument `E0` that does not appear in the removed code.

Let us detail each case for terms needing new arguments:

The first case is when the argument can come from parent function parameters:

```
@@
type T;
identifier I0;
@@
  void parent_function(T I0, ...) {
  ...
      unmodified_function();
+     new_function(I0);
  ...
  }
```

The second case is when the argument is a brand new identifier:

```
@@
@@
+ int new_identifier;
  ...
  unmodified_function();
+ new_function(new_identifier);
```

> **!** In that case, the human or the automatic tool performing the transformation is responsible for choosing the name of the identifier.

The last case is when the argument is used by another unmodified term:

```
@@
expression E0;
@@
  function_using_argument(E0);
  ...
  unmodified_function();
+ new_function(E0);
```

This case causes a problem similar to the one seen for control flow dependency on unmodified terms: “how to choose the relevant terms?”

> ! For this last case, the transformation requires control-flow dependencies to unmodified terms as well.

2.3.4 Change instances per function

Another challenging category is the number of transformation instances per function. Indeed knowing the number of transformation instances is essential to knowing the size of the transformation. It is not necessarily easy to deduce the number of instances as developers often copy-paste code requiring several instances of the same transformation at the same place. Consequently the inference task gets harder with the number of instances of the transformation at the same place. For this category we will define "at the same place" by being in the same function but other choices, more coarse or fine-grained, are possible.

> ! In this category most of the examples will be illustrated by both diffs and semantic patches.

One instance

The easiest level in this category is when only one change instance is present per function. In this case there is no ambiguity in the transformation boundaries.

Multiple non-overlapping instances

Inferring transformations gets more difficult when examples have multiple instances of the same change in the same location, as in the following diff:

```
- api_usage(var1);
- api_usage(var2);
```

In this case, the transformation could be carried out by either of these semantic patches:

```
@@
expression E0, E1;
@@
- api_usage(E0);
- api_usage(E1);
```

```
@@
expression E0;
@@
- api_usage(E0);
```

In these two semantic patches, the first one remove all double instances of the `api_usage` function call, whereas the second one remove all instances. If there is no need to check explicitly for double instances then the second semantic patch is better than the first one, because it is more concise and more general.

Inferring the correct transformation can be difficult if all examples in the learning set are double instances but the underlying transformation only requires one instance to be performed.

Overlapping instances

The highest level of difficulty is when transformation instances overlap, like in the following diff:

```
- first_fun(var1);
- first_fun(var2);
- second_fun(var1);
- second_fun(var2);
```

In this case it is hard to decide whether the transformation is the following semantic patch:

```
@@
expression E0, E1;
@@
- first_fun(E0);
- first_fun(E1);
- second_fun(E0);
- second_fun(E1);
```

Or this one, assuming this example is illustrating two overlapping instances of the same transformation:

```
@@
expression E0;
@@
- first_fun(E0);
...
- second_fun(E0);
```

2.3.5 Change variants

It is not necessarily the case that all transformation instances for the same API usage update can be captured by a single transformation rule, as multiple small variations can appear in each instance. These variations can differ in both control flow and data flow, the number of terms used and the kinds of terms used. We will call each of these variation a transformation variant.

One variant

The easiest transformations to infer are those with only one transformation variant, meaning they can be generated with only one transformation rule. That was the case for all transformations we have shown previously in this taxonomy.

Multiple variants

The second level of difficulty in this category is when a transformation has multiple variants.

Here is an example of a transformation composed a two variants, updating different function to the same one:

```
@rule1@
expression E0;
@@
- old_function1(E0);
+ bar(E0);

@@rule2@
expression E0;
@@
- old_function2(E0);
+ bar(E0);
```

Variants can differ by data flow, as in the following example:

```
@rule1@
expression E0;
@@
- old(E0, 0);
+ new_when_zero(E0)

@rule2@
expression E0;
@@
- old(E0, 1);
+ new_when_one(E0)
```

Order dependent variants

Sometimes transformation rules need to be put in a specific order to be able to perform the correct overall transformation.

```
@rule1@
expression E0;
@@
- old(E0, 0);
+ new_when_zero(E0)

@rule2@
expression E0, E1;
@@
- old(E0, E1);
+ new_when_nonzero(E0)
```

In this semantic patch, rule1 and rule2 cannot be reordered without changing the semantics of the transformation. This happens here because every time rule1 is applicable, rule2 is also applicable.

2.3.6 Noise

The final category in our taxonomy of challenges is the presence of noise in the example to learn from. We defined noise as any of these four items:

- Unrelated changes: changes that are not part of the overall transformation.

- Example specific changes: changes that are part of the overall transformation but that make sense only in their specific example. Those kinds of changes are often found when deleting whole blocks of code regardless of their content.
- Incorrect changes: changes that have been performed incorrectly.
- Missed changes: changes that should have been performed but that were not.

Noise in the transformation examples means that there exist some transformation instances that are counter-example to the change semantic.

No noise

The first level of difficulty is when there is no noise, so all changes illustrate the overall transformation and the overall transformation only.

Contains noise

The second level is when transformation instances contain noise, in the four categories seen above. Handling examples at this level of difficulty is important as human developers often make mistakes or omissions, or group multiple unrelated transformations together.

We will illustrate the first two items we listed previously.

Unrelated changes are present in the examples when some examples have terms that are changed but have nothing to do with the transformation. Unrelated changes appear mostly when the modified portion of the code contains some cleaning such as in the following diffs:

```
- old_fun();
- printf("DEBUG: %s\n", unrelated_string);
+ new_fun();
```

```
- old_fun();
+ new_fun();
```

The removal of `printf` is considered unrelated if the overall transformation has the following semantics:

```
@@ @@
- old_fun()
+ new_fun()
```

Example specific changes appear when arbitrary blocks of code are removed, such as in the following diffs:

```
- if (value_ok())
-     var = 0;
```

```
- if (value_ok()) {
-     specific_fun1();
-     specific_fun2();
- }
```

In this case the overall change is described by the following semantic patch, which does not care about the contents of the then branch of the removed `if`:

```
@@
statement S;
@@
- if (value_ok())
- S
```

2.3.7 Summary

The Table 2.1 on page 26 gives a summary of every level in every category of our taxonomy. At the left side of the table are some indexes that we will use later when talking about the taxonomy. The letters indicate categories in the taxonomy and the numbers the levels in these categories.

We also introduce a notation to refer to category level concisely: Instead of saying that a transformation need *intraprocedural dependencies between changed and unchanged terms* we can simply say that a transformation is C_2. With this notation we can also discuss about tool capabilities. For instance saying that a tool targets V_1 is the same as saying that this tool targets transformations that are either composed of one or multiple variants, but it does not target variants requiring to be applied in a specific order.

C.	Control-flow dependencies
0.	*No* control-flow dependencies between changed terms
1.	*Intraprocedural dependencies* between changed terms
2.	Intraprocedural dependencies between changed and *unchanged terms*
3.	Negative control-flow dependencies
4.	*Interprocedural* control-flow dependencies
D.	Data-flow dependencies
0.	*No* data-flow dependencies between changed terms
1.	Data-flow dependencies between changed terms
2.	Data-flow dependencies between changed and and unchanged terms
I.	Change instances per function
0.	One instance
1.	Multiple instances
2.	Overlapping instances
V.	Change variants
0.	One variant
1.	Multiple variants
2.	Multiple variants with a specific order
N.	Noise
0.	*No* noise
1.	Contains noise

Tab. 2.1: Summary of the transformation challenges taxonomy

3 State of the art

We have previously seen the challenges of transformation learning. We will now study the state of the art for learning transformations from a set of examples. We will mainly focus on fully automatic approaches learning API usage updates from examples, and assess the strengths and weaknesses of these approaches in terms of the taxonomy challenges we introduced earlier. We will also see more distant related work, with semi-automatic approaches and approaches learning transformations other than API usage updates.

3.1 Automatic inference of API usage updates

The problem of API-usage update learning can be decomposed into multiple sub-problems that need to be addressed:

1. **Finding relevant examples:** The first step of every approach to learning API usage updates from examples is to find one or more examples of the transformation we want to learn.

2. **Extracting transformation information from examples:** Once examples of a transformation are obtained, the approach needs to determine what the transformation is doing. To do so, the approach needs to extract different kinds of information from examples by converting the textual input to representations that can be analyzed.

3. **Creating and applying transformation rules:** From several pieces of information about the API usage update and its constraints, an approach must create transformation rules that can replicate the same API usage update at other locations and apply the transformation to a set of target code.

We will now study how the tools in the literature solve each of these sub-problems.

3.1.1 Finding relevant examples

The first step for API usage update inference is to find one or more relevant examples that will serve as input for the transformation inference algorithm. These examples are coarse-grained code extracts such as

- Pairs of files before and after the transformation.
- Pairs of functions before and after the transformation.
- Diff extracts.
- Sets of commits [Git] or patches.

This step is necessary to give an input to any approach that learns API usage updates from examples, but it is not part of the learning algorithm itself and it is often considered to be out of scope of the transformation inference problem. Consequently, it is in general only briefly mentioned in the related work, as many solutions have been found for this kind of problem in the field of code mining. For instance, SysEdMiner [MSDR17] can find systematic edits in Java commits. It represents edits as itemsets and uses CHARM's [ZH02] itemset mining algorithm to find similar edits that appear many times in the commits. C3 [Kre+16] encodes code changes into a similarity matrix and uses clustering algorithm to find similar changes. Patchparse [PLM06] is a tool that can report changes belonging to the same transformation, called *collateral evolution* in the original paper. It analyzes a set of diff patches, to find recurring occurrences of the same change in different files. Other approaches analyze commits to find those whose changes match a user-provided pattern, in the form of a semantic patch [Law+17], or in the form of an XML file [MM19].

Given the diversity of existing approaches to find systematic changes authors often suppose that the user has some ways of providing the inputs for their tools. However some tools provide their own automatic ways to collect API usage update examples.

For instance Meditor [XDM19] mines Github project commits that change API versions in build files for Java building systems such as Maven and Gradle. If a commit changes the version number of a used library in the build file, then all changes in that commit are considered to be migration examples for that library, from the old version to the new version.

A4 [LSC18] looks at the official Android documentation to find the "lists of all public methods available as API", then it mines online repositories such as Github

or F-Droid [Fdr] (an open source application store for Android) to find usages of the API and their associated commits. A4 first mines API usages in a two-steps process: the first step is matching strings corresponding to API names, the second step is using ASTs to remove false positives from the first step. Once API usages are found, A4 looks for commits that change the API usage. Similarly to the previous procedure, this is done in two steps, the first one using diff to find changes in strings corresponding to API names and the second one using the AST. Commits identified by this last phase are considered to be the input examples for the rest of A4 algorithm.

AppEvolve [FXO19] uses a user-provided name mapping, from old function names to new functions names, to know what API functions to look for. Then, it scans its target input code in order to find relevant keywords associated with the old functions in the mapping. Using these keywords it searches online an code base indicated by the user to find examples that changes the old API functions to new API functions. These examples are used as input for the AppEvolve API usage update learning algorithm.

3.1.2 Extracting information from transformation instances

Once the coarse-grained code extracts that contain examples of a transformation are found, tools need to extract meaningful information to generate transformation rules.

Text changes

As examples of transformations are often textual format, the first way to extract information is to compare what changed in the source code between before and after transformations. We already saw the most used tool to extract text change information, Diff [MES03], which can report blocks of deleted or added lines between two files. Some approaches in the literature improve the results of Diff by reporting more fine-grained textual information. In addition to removed and deleted lines, *LDiff* [CCDP08; CCDP09] and *LHDiff* [Asa+13] are able to report updated lines as well as moved lines using similarity metrics and their own line differencing algorithms.

It is often difficult to extract meaningful information directly from textual format. In order to learn about the transformation, most transformation learning tools convert

their textual input examples into a more structured format such as abstract syntax trees, data flow graphs and control flow graphs.

Abstract Syntax Tree

An abstract syntax tree (AST) is a structure used by nearly all existing tools in tranformation learning (Sydit [MKM11], Spdiff [And+12], Lase [MKM13], A4 [LSC18], Meditor [XDM19], AppEvolve [FXO19], and REFAZER [Rol+17]) to extract transformation information. An AST gives a tree representation of source code that keeps only the structure and throw away uninformative details such as spacing and comments. Such a tree allows tools to learn how the source code of the input example is organized.

We have already seen ASTs usage in A4 to refine matches of API usages during its example search phase. However, most tools use ASTs as a structure to compare code from before and after a transformation. For instance, REFAZER uses the tree edit distance defined by Zhang *et al.* [ZS89] on ASTs before and after changes to find groups of similar transformations. Sydit, Lase, Meditor and AppEvolve use ChangeDistiller [Flu+07], a tree differencing algorithm that generates AST edit operations, to understand fine-grained changes in the transformations. ChangeDistiller produces four kinds of AST-edit operations, *additions, deletions, updates* and *moves*, that describe how to transform the AST from before the change into the AST from after the change.

Besides ChangeDistiller, other work has been done to accurately report changes in tree-like structure like ASTs. Some approaches, like RTED [PA11], focus on finding optimal solutions with the low algorithmic complexity for AST-edits operations without the *move* operation. Other approaches focus on quickly reporting AST-edit operations that include *move* operations at a fine-grained level. Recent tools in this category include Diff/TS [HM08], GumTree [Fal+14] and MTDIFF [DP16].

Data-flow graph and data dependency

Data flow is the way variables relate to each other, which can help to understand a transformation.

One common form of learning from data flow is by constructing the data flow graph of the program, in which nodes are terms that create or use variables and edges connect nodes representing terms that create variables to nodes representing terms

that use them. A4 retrieves information from the data flow graph of all parts of the transformation instances while Sydit, Lase and Meditor only retrieve information from unmodified nodes that have data dependencies to modified nodes.

Control-flow graph and execution dependencies

Control flow gives the order in which instructions of the program will be executed, which can help to understand how the different functions or methods of an API are organized.

A common form to exploit control-flow information is by using a control-flow graph in which nodes represent instructions or blocks of instructions (often called basic blocks) and edges represent jumps [All70]. Surprisingly, to the best of our knowledge, no existing tools fully use Control-flow Graphs (CFGs) to learn properties of a transformation. However Sydit, Lase and Meditor use a subset of control-flow graph properties: execution dependency to modified nodes. "A node y is control dependent on x if y may or may not execute depending on a decision made by x." [MKM11]

Clone Detection

For tools learning transformations from multiple examples, it can be useful to detect similar unmodified portions of code between examples. As unmodified code is often much larger than modified code in the transformation examples, tools must rely on code clone detection techniques, optimized to detect similar portion of code in large code bases. For instance, LASE uses CCFinder [KKI02], a token-based code clone detector, to find common unmodified context linked to the transformation it is learning. Numerous other approaches exist in the domain of code clone detection [SK16]. Similarly to what we have seen for extracting information from change examples, code clone detectors can use different kinds of code representation to find clones. These representation can be text-based [Joh94], token-based [KKI02; YG12], tree-based [Jia+07; Whi+16] or graph-based [Kri01; CLZ14].

3.1.3 Creating and applying transformation rules

After processing examples to extract information about API usage updates, a tool must create transformation rules and apply them to suitable code to transform. We

can divide tools roughly into two main categories. The first one is for tools that can create a transformation rule from only one example and the second one for tools that need multiple examples. One important point to keep in mind is that creating a transformation rule from a single example does not necessarily mean using only one example as an input, tools using such process can, for instance, use multiple examples to find the best one to create a rule from.

We will now see the different strategies used by existing approaches to generate transformation rules from transformation instance examples. We will also comment on what inference difficulties can be managed for each strategy with respect to the taxonomy we defined earlier.

Creating rules from a single example

The majority of tools made to infer API usage update are able to infer a transformation from only one transformation instance example. When creating rules from a single example, the main challenge is to find what expression, variable or method name must be abstracted in order to generalize the single transformation instance into a generally applicable transformation rule.

Sydit [MKM11] first transforms its single example into a sequence of AST-edit operations, using ChangeDistiller [Flu+07]. Then, it identifies the edit context by retrieving the list of unmodified nodes that are connected by data dependencies or execution dependencies or that are AST parents of elements in the sequence of AST-edit operations. Next, it abstracts the variables, types and method names of elements in both AST-edit operations and in the context found at the previous step. The lists of abstracted edit context and AST-edit operations form the transformation rule. To apply the transformation, Sydit first abstracts variables, types and method names in its target code, then it tries to find a location, in a set of user-provided locations, that matches the context of its transformation rules. If such a location is found, Sydit applies the AST-edit operations and maps the abstracted identifiers and types of these operations to concrete names found in the target code.

In terms of the taxonomy, the strategy used by Sydit to generate transformation rules does not enable it to handle transformations that are sensitive to control-flow dependencies (C_0). Its algorithm uses execution dependency which only indicates whether a node affects the execution of another node, but the algorithm does not collect information about the execution order of the different nodes. Since it processes a single example at a time, it cannot handle transformations whose

variants require a specific ordering. However, provided that the user runs the tool multiple times on different examples, it can handle transformations composed of multiple variants (V_1). Sydit cannot deal with examples composed of multiple instances (I_0) since its learning algorithm will consider all instances as part of a unique transformation instance. Since all modifications in the transformation instance example are kept in the final transformation rule, Sydit cannot learn a correct transformation from an example containing noise (N_0).

AppEvolve [FXO19] starts with an Android target app and a user-provided mapping from old API method names to new API method names. AppEvolve then looks for all APIs in the mapping that are used by its target app, and for each of them looks for online API usage update examples as described in the previous section.

For each found example, AppEvolve uses AST-edit operations to represent changes, similarly to Sydit, but restrict edits to operations on AST statement nodes instead of any AST node. Restricting operations to statements is more coarse-grained than allowing operations on all types of AST node, which in theory should lead to a less accurate change representation than that used by Sydit. Our reproducibility paper [Thu+20] shown that AppEvolve mitigates this issue with code normalization to put most expressions used in API usages into variables. As a consequence, all changes in expressions are transformed into changes on statements which makes the AppEvolve's AST-edit operations equivalent to the ones at smaller granularity.[1] AppEvolve does not retain all AST-edit operations to construct a transformation rule. Instead it first keep only operations affecting the methods in its mapping and then uses data-flow dependencies to add connected AST-edit operations.

AppEvolve then abstracts all variables in all collected AST-edit operations to create a transformation rule. When provided with several API usage update examples, AppEvolve generates a transformation rule for each of them, then computes the common part of the generated rules and ranks the generated rules according to their proximity to the common part. Then using the user provided location, and if provided a test suite, AppEvolve tries to apply the transformation rules in rank order until one matches the code at the location provided by the user and passes the test suite.

In terms of the taxonomy, AppEvolve cannot handle examples sensitive to control-flow dependencies because, like Sydit, it uses only AST-edit operations and does not collect control flow information connecting the positions of these edit operations

[1] Normalizing source code has drawbacks such as not preserving the coding style, creating new variables with unclear names and possibly changing the semantics of the program, if done incorrectly.

together (C_0). Computing the common core between a set of transformation rules to rank them allows AppEvolve to deal with examples containing multiple transformation instances, even interleaved (I_2). However this strategy prevents AppEvolve from dealing with transformations of multiple variants in a specific order (V_1), as there is no guarantee that the order follows the proximity to the common core. As AppEvolve does not use all edits but only those connected to the API to change, it is capable of handling API usage update examples containing noise (N_1).

Meditor [XDM19] mines online commits related to changes in library usages and processes each of them individually. The strategy used to get from a commit to a transformation rule is similar to that of AppEvolve: it starts from AST-edit operations on statements that directly change the API call, then adds other AST-edit operations that are connected by either data dependencies or execution dependencies to the former operations and finally abstracts variable names and user-specific methods to create a transformation rule. Contrary to AppEvolve, Meditor does not rank transformation rules.

In terms of the taxonomy, Meditor can handle the same transformation challenges as AppEvolve, since these two tools share a very similar approach.

A4 [LSC18] mines online example of Android APIs usage using the procedure described in the previous section. Then it analyzes its target application to find usage of Android APIs. For each API used, it tries to find a migration candidate by comparing the data flow for the API usage to the data-flow graph of online examples mined previously. If a migration candidate is found then A4 constructs a migration mapping that contains the changes that must be made to existing statements in order to get from the old data-flow graph to the new data-flow graph. This migration mapping also contains the new data-flow graph nodes that are created by the migration. Similarly to AppEvolve and Meditor, A4 does not use all the modified nodes to construct its mapping but only nodes with data dependencies to the API call. At last, A4 proceeds with the migration of the target app, one API at the time, by using the migration mapping it adds the new data-flow nodes to the target app data-flow graph and migrates the existing nodes.

Although A4 uses a different strategy than AppEvolve and Meditor, it can handle the same set of transformation challenges.

GenPat [Jia+19] first represents its single transformation example as a hypergraph, whose nodes are AST nodes that contain a list of attributes. These attributes can be the type of AST node or the string representation of the subtree at the node. Each edge represents either direct-parent relations, indirect-parent relations (transitive closure of the direct-parent relations), or data-flow relations. Changes are represented by the list of AST edit operations, similarly to Sydit, AppEvolve and Meditor. The core idea of GenPat is to abstract attributes of nodes depending on their popularity in online examples. To do so, GenPat mines online repositories to find the popularity of each attribute. When they are not popular enough according to a user threshold, the attributes are abstracted, creating an abstracted hypergraph. Once this is done, GenPat takes its abstracted hypergraph and tries to match it to code base, by constructing an hypergraph for each piece of code and matching it to the previously constructed hypergraph. If a match is found, GenPat proceeds with the changes, using the list of AST-edit operations.

Like its predecessors, GenPat is not able to learn transformations that require taking into account control-flow dependencies (C_0). Moreover since GenPat considers all edits to be part of the inferred transformation rule it cannot learn correct transformations from examples with noise (N_0).

Creating rules from multiple examples

We have seen that it is not possible for approaches learning from a single example to handle transformations with multiple variants requiring a specific order. It is also difficult for these approaches to deal with examples containing multiple instances of a transformation or examples containing unrelated changes.

We are now going to review how approaches learning API usage updates from multiple examples can solve some of the previously mentioned problems.

Lase [MKM13] first transforms each of its transformation examples to a sequence of AST-edit operations (similarly to Sydit), and then searches for the common subsequence of all these examples. The core idea is that multiple instances of the same transformation must share the same AST-edit operations. Identifiers such as methods or variable names in this subsequence are then abstracted if they differ between the original AST-edit sequences. Then, Lase searches for transformation context: first it uses the CCFinder [KKI02] code clone detector to find similar context in the AST subtree, and then, for each tree, it abstracts all identifiers and extracts the common subtree for all these trees. Next, nodes in the common context subtree that

are not linked to modified nodes by data flow dependencies are removed. After that, Lase takes the common subsequence of AST-edit operations and the common context subtree, and looks for a matching location in its target code. If one is found, it uses the subsequence of AST-edit operations to migrate code at the found location.

In terms of the taxonomy, Lase's strategy allows it to learn from examples with noise (N_1), as noise will not be part of the common AST-edit subsequence. However this is done at the expense of learning transformations with multiple variants (V_0), because variants do not share a common sequence of AST-edit operations. In terms of control flow, as Lase only uses AST-edit operations to describe changes, it cannot deal with transformations requiring control-flow dependencies between the changed terms (C_0).

REFAZER [Rol+17] uses inductive programming to generate transformation rules. REFAZER first identifies AST-edit operations for each example, then finds connected edit examples by using tree-edit distance [ZS89]. The core idea behind REFAZER is that transformations found in the examples can have been generated by multiple rules, so edits must be grouped by similarity before inferring a transformation rule for each group. Each group of similar edits is found by using DBSCAN [Est+96] clustering algorithm. Then, for each group, REFAZER's inductive programming framework infers a transformation rule. For this, it first finds constraints that must be checked by a transformation rule in order to generate the observed AST-edit operations, then generats transformation rules that match those constraints and finally ranks the generated transformation rules to find the best one. Each REFAZER transformation rule is composed of a *location expression*, which indicates how to find the location where a change is needed, and an *operation*, which is an AST-edit operation on a subtree.

REFAZER's approach is different from the other approaches that we saw earlier, in the sense that it does not construct a single transformation rule from a set of examples but generates multiple transformation rules that match the examples and ranks them afterward. We have seen that REFAZER first clusters examples before inferring a transformation rule which enables it to deal with transformation with multiple variants (V_1) and transformation examples with multiple instances or even interleaved examples (I_2). In terms of control flow, since the rules are inferred from AST-edit operations constraints REFAZER cannot learn transformation requiring control flow dependencies (C_0). Since all edits are considered part of the transformation, it also cannot learn from examples containing noise (N_0).

Spdiff [AL10; And+12] processes user-provided change sets, which are pairs of AST extracts from before and after a change, to generate a Coccinelle semantic patch. It starts by generating semantic patch patterns that must match at least a certain number of terms on the part of the change set before the change. The number of terms to match is a parameter provided by the user. Then it iteratively creates a semantic patch using the previously found patterns by putting the SmPL sequence operator (`...`) in between each pattern, while checking if the resulting semantic patch still matches examples from before the change. Once this is done, Spdiff obtains a semantic patch composed only of removed parts. It then obtains the added parts, by computing the difference of the control-flow graphs of the examples from before and after the change and pair added parts to removed parts, while generating metavariables. In the end, Spdiff obtains a complete semantic patch that the user can use with the Coccinelle engine.

Because it uses the SmPL sequence operator `...` to connect patterns, Spdiff can learn transformations with control-flow dependencies between changed terms (C_1). However, since it tries to match as many terms as possible in its change set, it cannot learn correctly from examples with noise (N_0), and it must learn from examples from the same variant of the transformation (V_0).

3.2 Semi-automatic inference and recommendation

We have seen approaches that can autonomously infer and apply an API usage update transformations learned from a set of examples. However, existing work has also been done to infer API usage update transformations in a semi-automatic fashion, either by requiring user input to infer transformations or by presenting multiple inferred transformations to the user.

In this category the closest related work is ARES [Dot+17], a recommendation system that learns systematic edits. ARES learns from a list of method pairs from before and after a transformation. Using the MTDIFF [DP16] AST-differencing algorithm it compares all the methods before the changes and all the methods after the changes to create an edit pattern. When faced with different variants for the added parts, it lets the user choose which one to use.

Another approach by Thung *et al.* [Thu+16] recommends code changes for Linux drivers backporting. Due to API evolution, a target driver introduced in recent Linux kernel versions often does not compile in older versions, and needs to be backported. The approach uses the fact that commits that change APIs and prevent a target

driver from compiling often contain related API-usage changes in other drivers, that give information about how to backport a driver. Using this information, Thung *et al.*'s approach finds the most recent Linux kernel commit that prevents compilation of a target driver. Once such commit is found, using GumTree the approach analyzes commit AST changes and compares them to the AST in the line indicated by the compilation error, deducing a set of transformations that could fix the error. Inferred transformations are then ranked and presented to the user, who must choose the most appropriate one.

LibSync [Ngu+10] is another approach to recommend locations and edits for API-usage updates. This approach constructs a database of API migration patterns, first by extracting API usage templates in client code, then by inferring migration patterns from the old version to the new using a graph alignment technique. From this pattern database and a target app, LibSync can suggest locations and edits in the target app to migrate outdated API usages.

3.3 Inference of other transformations from examples

The work we have seen so far focused on inferring API-usage updates. More distant related work targets the inference of other kinds of transformations, by learning from examples.

Rase [Men+15] is a direct extension of Lase that targets code clone refactoring. Indeed, systematic edits targeted by Lase may indicates that the fragments of code that have to be changed many times in a similar fashion can also benefit from a refactor. Using the edit script generated by Lase, Rase identifies the portion of code that needs to be refactored, extract common code into new methods and adapt code clone to use these newly created methods.

REVISAR [Rol+18] learns short fix templates that can help code analyzers detect and repair dubious code fragments. From revisions in a code repository, REVISAR extracts AST-edit operations using GumTree, then clusters similar edits and infers a short fix template for each cluster.

VuRLE [Ma+17] can detect and repair code vulnerabilities by learning vulnerability patterns from repaired examples. It first uses ChangeDistiller to extract AST-edit operations that transformed vulnerable examples into repaired ones. It then creates edit groups by clustering AST-edit operations with DBSCAN and then, for each group,

it generates a repair template. Given a target app, VuRLE uses these repair templates to detect and fix vulnerabilities in the app.

More distant approaches in the field of automatic software repair [Mon18], also exploit information from already performed changes to limit the search space for repair candidates. [LR16; LLLG16; Jia+18]

3.4 Conclusion

In this chapter we first analyzed existing approaches for automatically learning API-usage updates from examples and then studied other related approaches in the domain, which were either learning semi-automatically, or learned different kinds of transformations.

When we discussed the methods used by automatic approaches to infer API usage updates, we identified for each approach its inference capabilities in terms of the taxonomy. We found that only a few approaches can infer transformations from examples containing noise or transformations with multiple variants, and only one can infer transformations relying on control-flow dependencies. In order to handle Linux kernel API usage updates, we need to design an approach that specifically targets these challenges.

4 Contributions

In this chapter we will see in detail the contributions of this thesis through a novel approach to infer transformation rules for API-usage update using a set of transformation examples. The target of choice for our approach is the Linux kernel, a monolithic open-source code base with millions of lines of C code.[1] The structure of the Linux kernel is peculiar: API definitions coexist with API usages. This peculiarity means a change in API definition potentially affect thousands of API usages, triggering the needs for automatic transformation tools and, at the same time, providing us with a lot of API-usage migration examples.

Our approach has been implemented as a tool named SPINFER and published at the USENIX Annual Technical Conference of 2020. [Ser+20] We will first start with a few technical choices for our approach.

4.1 Technical choices

In light of what we saw in the existing work, we will here justify some technical choices used in our approach, which aims to reduce shortcomings observed in some existing tools.

4.1.1 How many examples?

We have seen in the related work that the existing approaches are divided between those that learn from one example and those that learn from multiple examples. To build our solution, we must also make a choice.

We saw that approaches learning transformations from a single example faced some challenges. Firstly, since they learn from a single example the user has to carefully choose the example to ensure that it is characteristic of the transformation. To ensure optimal results this single example must contain little to no noise and only one instance of the transformation. To mitigate this issue, some tools, like AppEvolve,

[1] Around 20 millions lines of code as of September 2020.

infer multiple transformation rules, from different examples and select the best one for their target code. Secondly, tools learning from a single example have to find the correct level of abstraction for their transformation rule. Most approaches use heuristics like systematically abstracting away terms that are usually instance-specific, like local variables, to find their abstractions. Others, like GenPat, rely on data mining on open source software repositories to find correct abstractions.

Since the mitigations for these shortcomings rely on using more examples in one way or another, we might as well, in our approach, learn transformations from multiple examples.

However, learning transformation rules from multiple examples is not exempt from challenges. Approaches learning from multiple examples need to recognize transformation variants that will be mixed together in the learning set, otherwise they could end up with incomplete transformation rules.

4.1.2 Transformation language

While most of the existing tools use an internal language to represent their transformation rules, it is generally not designed to be read by human beings. Often users are left with only the results of applying these inferred transformation rules to their target code base.

In our work, we target transformation inference for infrastructure software, like the Linux kernel. We know that the generated transformations instances will be carefully checked by developers before accepting them [Sub]. However, to determine whether a transformation is correct it is often quicker to discuss the transformation rules rather than the transformation instances. Moreover, in the case of the Linux kernel, developers are already used discussing transformation rules, when they review submitted SmPL semantic patches.

As a consequence we will use the same transformation language as Spdiff and generate SmPL rules. One advantage of this method is that developers can check the rules before applying them and even correct them if the rules are close to an ideal solution. Another advantage is, since we will rely on the Coccinelle engine to apply the generated rules to a target code base, we do not have to design an approach to apply a transformation to a target code base.

4.2 Overview

Before looking in detail at our approach, we will provide an overview complemented by a motivating example. Our motivating example will give us an idea on how to learn and generate transformation rules (using SmPL in our case) from transformation instances.

4.2.1 Motivating example

Our motivating example comes from the timer migration in the Linux kernel that we saw earlier in section 2.2.1 on page 6.

Here are 5 examples of the change, targeting 4 different variants of the transformation:

File: drivers/s390/block/dasd.c

```
- init_timer(&device->timer);
- device->timer.function = dasd_device_timeout;
- device->timer.data = (unsigned long) device;
+ setup_timer(&device->timer, dasd_device_timeout,
+              (unsigned long)device);
```

File: drivers/atm/nicstar.c

```
- init_timer(&ns_timer);
+ setup_timer(&ns_timer, ns_poll, 0UL);
  ns_timer.expires = jiffies + NS_POLL_PERIOD;
- ns_timer.data = 0UL;
- ns_timer.function = ns_poll;
```

File: drivers/nfc/st-nci/ndlc.c

```
- init_timer(&ndlc->t1_timer);
- ndlc->t1_timer.data = (unsigned long)ndlc;
- ndlc->t1_timer.function = ndlc_t1_timeout;
-
- init_timer(&ndlc->t2_timer);
- ndlc->t2_timer.data = (unsigned long)ndlc;
- ndlc->t2_timer.function = ndlc_t2_timeout;
+ setup_timer(&ndlc->t1_timer, ndlc_t1_timeout,(unsigned long)ndlc);
+ setup_timer(&ndlc->t2_timer, ndlc_t2_timeout,(unsigned long)ndlc);
```

File: arch/blackfin/kernel/nmi.c

```
- init_timer(&ntimer);
- ntimer.function = nmi_wdt_timer;
+ setup_timer(&ntimer, nmi_wdt_timer, 0UL);
```

File: drivers/media/usb/au0828/au0828-dvb.c

```
- dev->bulk_timeout.function = au0828_bulk_timeout;
- dev->bulk_timeout.data = (unsigned long) dev;
- init_timer(&dev->bulk_timeout);
+ setup_timer(&dev->bulk_timeout, au0828_bulk_timeout,
+        (unsigned long)dev);
```

Based on these 5 examples of the `init_timer` to `setup_timer` migration, a human expert could write the following semantic patch,[2] composed of 4 transformation rules:

```
@@expression T,F,D;@@
- init_timer(&T);
+ setup_timer(&T, F, D);
  ...
- T.data = D;
- T.function = F;

@@expression T,F,D;@@
- init_timer(&T);
+ setup_timer(&T, F, D);
  ...
- T.function = F;
- T.data = D;

@@expression T,F,D;@@
- T.function = F;
- T.data = D;
  ...
- init_timer(&T);
+ setup_timer(&T, F, D);

@@expression T,F;@@
- init_timer(&T);
+ setup_timer(&T, F, 0UL);
  ...
- T.function = F;
```

In this semantic patch, each rule corresponds to a different variant of the transformation.

This migration is particularly interesting because, even if it is conceptually simple it took a long time to be fully performed in the Linux kernel. The migration started in

[2] The actual semantic patch written by a kernel developer for this transformation is more complex. For clarity we present here a simplified version. The real semantic patch can be found in the Linux kernel git repository at commit b9eaf1872222.

Linux 2.6.17, in 2006, and ended up in Linux 4.15, in 2018. The final migration, impacted hundreds of files and was performed using a semantic patch.

We said earlier that, when learning from multiple examples, it is important to identify the different variants to generate correct transformation rules. However a strategy like the one used by REFAZER, where variants are identified and separated early before applying the learning algorithms, is not optimal. Indeed, processing each variant in isolation means that found abstraction will be learned from only on a subset of examples. In our case most variants are illustrated by only one example, so if we were to separate variants before learning abstractions we will not have much information to get high-quality abstractions. Instead, we can look at the properties of the human written semantic patch to get insight about a better strategy to infer a semantic patch from these examples.

We can observe that the human semantic patch consists in only 4 major constituents:

- The removal of the call to `init_timer`.
- The addition of the call to `setup_timer`.
- The removal of the initialization to the `function` field.
- The removal of the initialization to the `data` field.

Hereinafter we will call these constituents *rule fragments*, as they are fragments of rules of a semantic patch. These 4 rule fragments are the same between all the rules (except the last one that does not have the `data` field initialization removal), arranged in different orders. That means that some variant examples can contain information to find rule fragments that are useful in rules not only targeting their variant but also rules targeting other variants. Of course, this is not always the case, and some transformations will contain rules that do not share rule fragments with other rules. However this highlights the fact that, if we want to maximize information available to obtain the best possible abstractions for our rule fragments, we should consider finding these fragments before considering any variant. This observation will guide the core idea of our approach and its implementation SPINFER: finding rule fragments separately from their context and assembling them.

4.2.2 Transformation rules as an assembly of rule fragments

We want to view semantic patch rules as an assembly of rule fragments, each of which matches several similar terms in our concrete change examples. This view is illustrated by Figure 4.1.

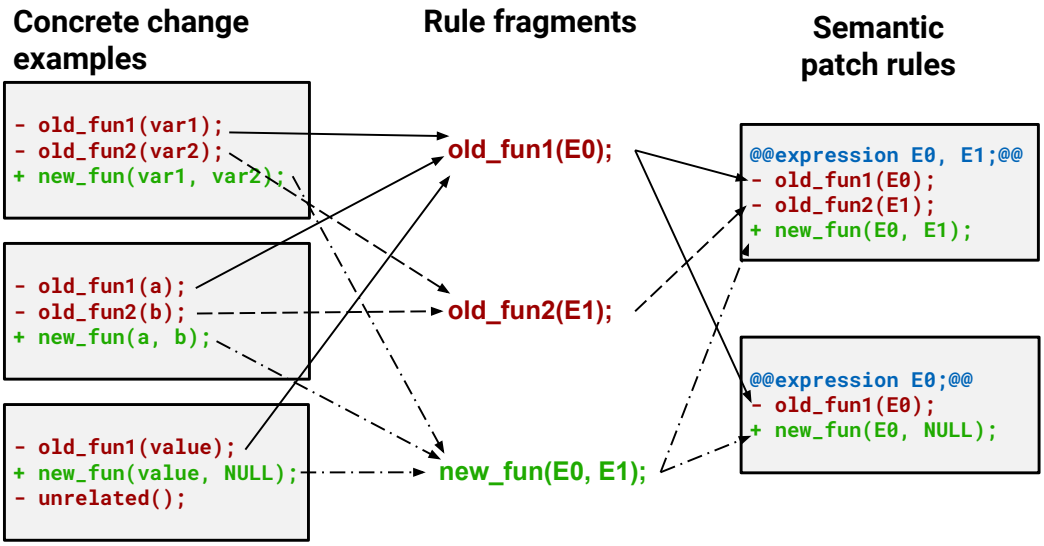


Fig. 4.1: Abstract fragments

In this view:

- Rule fragments are assembled using the control-flow dependencies of the concrete terms each rules fragment represents.
- Metavariables between fragments are found using data-flow dependencies.
- Transformation rule variants are simply different assemblies of the same set of rule fragments.
- Noise represents uncommon concrete terms that cannot form rules fragments due to their low popularity among other terms.

4.2.3 Variants

Another observation we make about the timer semantic patch is that the last rule must necessarily come after the first two. This is because every time the first two rules can be applied so can the last rule.

```
@@expression T,F;@@
- init_timer(&T);
+ setup_timer(&T, F, 0UL);
  ...
- T.function = F;
```

If we look in detail at the last rule above, we can see that it contains no data field, which in this case is assumed to be 0. Applying this rule before the first two will then lead to false positives, with `setup_timer` third argument set to 0, and false negatives with the skipped removal of the `data` field initialization. This highlights the need to being able to order the rules but also to evaluate them on examples in order to detect those false positives and false negatives.

4.2.4 Summary

With all of these observations in mind, we can summarize our semantic patch inference approach as follows:

1. **Identifying rule fragments and noise:** We first cluster similar terms from examples and generalize each cluster into a rule fragment that matches all the members of the cluster. Unpopular terms that cannot be grouped together are marked as noise and excluded from further processing.

2. **Assembling rules fragment:** We combine the rule fragments together using the control-flow dependencies in the examples from which these rules fragments are derived.

3. **Rule ordering:** We order the generated semantic patch rules to maximize the semantic patch coverage while minimizing the number of false positives and false negatives.

4.3 Identifying changes

Like any other approach learning transformations from change examples, we must first convert our change examples into a more structured format that we can analyze and extract information from.

As we saw earlier with existing approaches, there exist several possible representations for transformation examples such as abstract syntax trees (ASTs), data-flow

graphs (DFGs) or control-flow graphs (CFGs). In our case, we want to obtain SmPL rules that will be processed by the Coccinelle engine by assembling rule fragments using control-flow dependencies. Internally, Coccinelle uses control-flow graphs to process SmPL rules, and these rules themselves can describe control-flow relationships with the sequence operators. As we will need to retrieve control-flow dependencies, we might as well start early and convert our examples into control-flow graphs.

To extract control-flow graphs from our learning examples we use Coccinelle internal libraries that take a single file as input and generate a control-flow graph for each function in that file. Note that, contrary to traditional control-flow graphs used in compilation, in which each node is a basic-block composed of multiple statements [All70], Coccinelle control-flow graph nodes are at most one statement.

4.3.1 Augmenting control-flow graphs

Before exploiting control-flow information from graphs, we need to augment them by adding empty nodes whose sole purpose is to facilitate the analysis of control-flow relationships. This process will help the inference of semantic patches. To understand why we need to augment them we first need to understand what the precise meaning of the SmPL "..." sequence operator is and for that we have to introduce the *dominance* [LM69] notion for control-flow graphs.

In a control-flow graph $\mathcal{G}$, a node A *dominates* a node B if every path from the entry node of $\mathcal{G}$ to B goes through A. A is then a dominator of B. Similarly a node B *postdominates* a node A if every path from A to the exit node of $\mathcal{G}$ goes through B. B is then a postdominator of A. Moreover we said that A strictly dominates B if A dominates B and A is not equal to B.

In fact, the SmPL "..." sequence operator uses the postdominator notion, where `A ... B` means that pattern `B` must postdominates pattern `A`, which is equivalent to the notion we introduced earlier which was `B` must be after `A` in all control-flow paths from `A`. The only difference is that as `A` and `B` represent patterns the postdominator notion relies on groups of nodes instead of single nodes.

Technically Coccinelle excludes error exit branches when checking postdominator relationships. Usually these branches are of the form:

```
if (condition)
    return error_code;
```

Now we will see why we need to add supplementary information to example control-flow graphs. Let us suppose that we want to analyze this piece of code:

```
if (condition) {
    A;
    B;
}
else
    C;
```

Our goal is to deduce this semantic patch rule that matches our example code:

```
@@
@@
if (condition) {
    ...
    A;
    ...
    B;
    ...
}
else {
    ...
    C;
    ...
}
```

After using the Coccinelle libraries we can extract the control-flow graph shown in Fig 4.2:

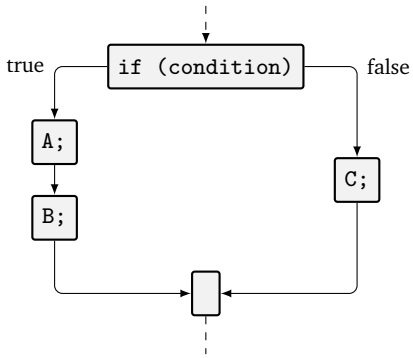


Fig. 4.2: Excerpt of a control-flow graph from Coccinelle libraries

In this control-flow graph, we can see that `B` postdominates `A`, so it will be possible to deduce the following excerpt:

```
    A;
    ...
    B;
```

However since `A` and `C` does not postdominate `if (condition)` and nothing in the resulting rule postdominates `B` and `C` then it will be harder to deduce the other relationships.

This problem can be solved by adding non-printable nodes at the start and end of each branches, such as control-flow graph in Fig. 4.3:

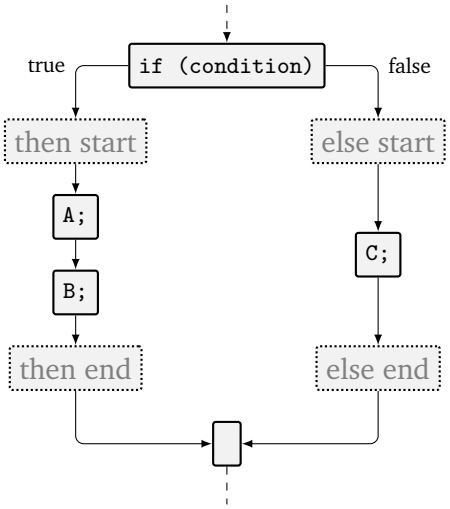


Fig. 4.3: Augmented CFG (non-printable nodes are represented with dotted pattern)

With that modification, for each branch, there is a node attached to our `if (condition)` that dominates the whole branch and another node that postdominates it. When analyzing this graph we can now deduce that, for instance `C` postdominates *else start*, which belong to our `if (condition)`, and *else end* postdominates `C` so we can construct the following rule excerpt:

```
10 else {
11     ...
12     C;
13     ...
14 }
```

We can proceed in this manner for all branching and iterator structures in the control-flow graphs associated with our transformation examples.

Now we can build a list of augmented CFGs for each pair of file examples before and after a transformation. This is done by extracting a CFG for each function in each file, and augmenting each control-flow graph using the procedure described above.

4.3.2 Control-flow graph differencing

We now want to understand what changed between functions before and after the transformation. In the same way that Diff allows us to understand how the lines in a file have been modified, we will use a form of diff to represent how the CFGs have changed. For that purpose we introduce a structure, the CFG-Diff, a Diff variant for control-flow graphs.

A CFG-Diff is a control-flow graph in which each node and edge is either marked as removed, added or unchanged. To build CFG-Diffs we need to find the common nodes and edges between the CFGs of a function before and after a transformation, that is to say find the maximum common induced subgraph between the two control-flow graphs. As this problem is NP-hard and function CFGs can potentially contain thousands of nodes and edges, we will try to find an approximation to stay within reasonable compute time. To find that approximation we will use GNU-Diff that reports the position of changed lines, although this approach can work with any tool that reports the position of code changes.[3]

Let G_B be the control-flow graph of a function before the transformation and G_A the control-flow graph after the transformation. We first use GNU-Diff report on changed line positions to build a mapping M from unmodified line positions in the after function to unmodified line positions in the before function. Then, for each node in each graph, we tag nodes as modified if their positions are not present in the mapping, namely the values of M (for G_B) or in the keys of M (for G_A).

We illustrate this process with our second timer example, that is recalled here:

File: drivers/atm/nicstar.c

```
1 - init_timer(&ns_timer);
2 + setup_timer(&ns_timer, ns_poll, 0UL);
3   ns_timer.expires = jiffies + NS_POLL_PERIOD;
4 - ns_timer.data = 0UL;
5 - ns_timer.function = ns_poll;
```

For this diff we give an extract of the source code before and after the transformation:

[3] More precise tools, like GumTree [Fal+14], can report position of changes more accurately than GNU-Diff but we choose GNU-Diff because of its extremely low execution time.

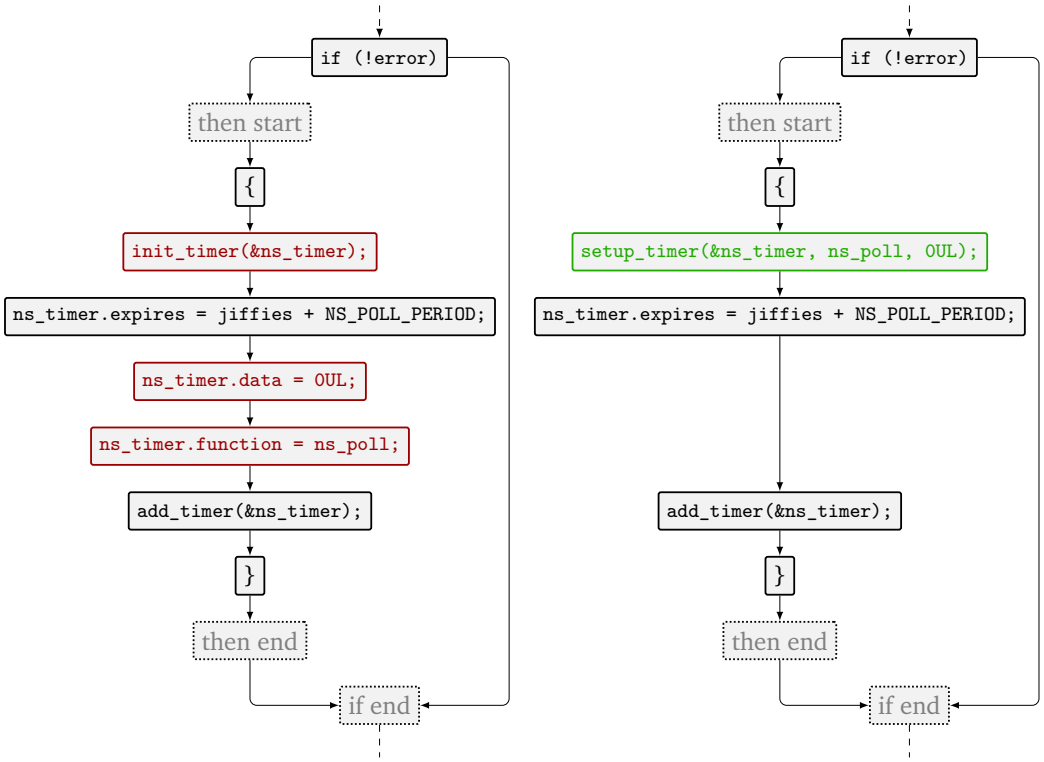
Before transformation

```
286   if (!error) {
287     init_timer(&ns_timer);
288     ns_timer.expires = jiffies + NS_POLL_PERIOD;
289     ns_timer.data = 0UL;
290     ns_timer.function = ns_poll;
291     add_timer(&ns_timer);
292   }
```

After transformation

```
286   if (!error) {
287     setup_timer(&ns_timer, ns_poll, 0UL);
288     ns_timer.expires = jiffies + NS_POLL_PERIOD;
289     add_timer(&ns_timer);
290   }
```

The associated CFGs are shown in Figure 4.4. For this example we have the following mapping of unmodified lines of code, from after to before the modification:
$\{286 \rightarrow 286,\ 288 \rightarrow 288,\ 289 \rightarrow 291,\ 290 \rightarrow 292\}$



(a) CFG before the transformation **(b)** CFG after the transformation

Fig. 4.4: Comparison of control-flow graphs, before and after the transformation

We will now build the control-flow graph diff G_{diff} for (G_B, G_A). The nodes of G_{diff} are the union of the modified and unmodified nodes of G_B and the modified nodes

of G_A. The modified nodes from G_B are marked as removed, and modified nodes from G_A are marked as added.

To build the edges we will reuse our mapping. We first start by adding all edges from G_B into G_{diff} marking them as removed. Then, we add edges from G_A into G_{diff} by mapping unmodified nodes from G_A to unmodified nodes from G_B. To map an unmodified node n from G_A to an unmodified node in G_B we look for a G_B node that is the identical to n and whose position matched the mapped line of n with our mapping M. Nodes with no line information, like `then start` in our example, can always be associated with a unique node with position information (in our case `if (!error)`) and, in that case, we use the associated node position. If any of the endpoints of an edge of G_A was unmodified, we find the corresponding node in G_B using the mapping procedure describe above, otherwise we use the modified nodes from G_A. If we add an edge (s, d) marked as added but an edge (s, d) marked as removed already exists then we replace the existing edge with an unmodified version. At the end of this procedure, we obtain a complete CFG-diff. An example of such CFG-Diff for our second `timer` example is shown in Figure 4.5 on page 54.

4.4 Identification of rule fragments

The next step is to find rule fragments, by clustering similar terms together and finding their common abstraction.

We first have to choose the granularity of the terms we want to cluster. Since we will use control-flow relationship to assemble rule fragments, we want our rule fragments not to represent more than one CFG node. Consequently, we choose the nodes of our previously constructed CFG-Diff as the limit of what a term is. For CFG-Diffs, nodes are not bigger than one statement, a function header or the header of a conditional.

Then, we proceed with the clustering of the CFG-Diff nodes to be removed and the CFG-Diff nodes to be added. In all generality, this problem of finding groups of similar terms is a code clone detection problem. However, traditional clone detection approaches have been optimized for detecting a few groups of clones that span across multiple statements in a large code base. In our case all modified terms will be at most one statement and almost all are expected to belong to a clone group. Only few will not be a clone of others and will be marked as noise terms.

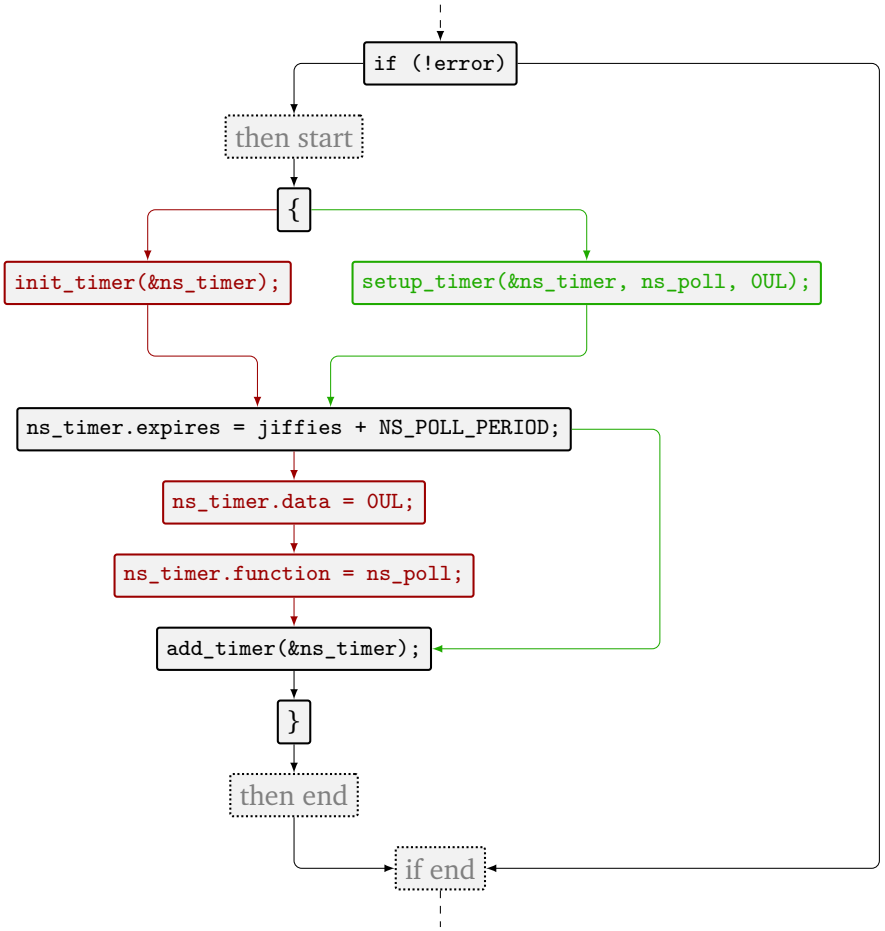


Fig. 4.5: CFG-Diff for `nicstar.c` file transformation

We propose here a novel approach for clone detection that clusters together similar changed nodes and detects nodes not belonging to any cluster. This approach focuses on the fact that each cluster of clones will be used to form an abstraction that matches all members of the cluster. This abstraction is formally known as the *anti-unifier* [Plo70] of the cluster.

> **!** Anti-unifiers and rule fragments are close notion but are not identical. Rule fragments use typed metavariables whereas anti-unifiers uses untyped placeholders. For instance, `E0.data = E1;` is a rule fragment with `E0` and `E1` being expression metavariables. On the other hand $?_1$`.data =` $?_2$`;` is an anti-unifier for the set { `a.data = b;` , `c.data = d;` }, with $?_1$ and $?_2$ being placeholders.

Clustering similar statements using anti-unification has already been used by CodeDigger [BM08; BM09], a code clone detector. However this approach requires a user parameter, which will be used as a threshold to determine whether to add a state-

ment to an existing cluster, or to add it to a new cluster. The metric defined in this paper to compute the distance between a statement and a cluster measure the cost of migrating a cluster to a new anti-unifier. This distance grows with existing clusters size and consequently, finding the best possible clusters requires tuning the user-threshold parameter [BM09].

In our case the number of examples we learn from, and so the cluster size, can vary significantly. As a consequence we use an approach that weights the anti-unifiers based on their popularity before clustering and a clustering procedure that does not require a user-provided parameter.

4.4.1 Node weighting

To facilitate clustering and noise detection, we want to give higher weight to anti-unifiers that are likely to form correct rule fragments, meaning rule fragments that are neither too concrete or too abstract.

Our approach relies on two main assumptions, that hold in the transformation examples we observed for real Linux kernel changes:

1. The majority of changed functions contain only one instance of a transformation.

2. Unrelated changes are uncommon compared to changes belonging to a transformation.

We will first see intuitively how to find a weighting function that gives such weights.

The table 4.1 shows a group of terms and 3 possible anti-unifiers for each term: one that is too specific, one that will generate appropriate rule fragments and one that is too generic. Let us say that we want our most appropriate anti-unifier to have the highest weight of the 3 proposed anti-unifiers so it will be easier to detect that terms 1 to 6 should be in one cluster and terms 7 to 11 in another cluster. One possibility is to look at the popularity of each anti-unifier. We can see that anti-unifiers that are too specific are not very popular, as each of them appears only once. On the contrary the appropriate and the too generic anti-unifiers appear in all files. To distinguish from the appropriate and the too generic anti-unifiers we can use the assumptions introduced at the beginning of this section. These assumptions hold for the commit performing our timer example in which there is no noise and only 6 out of 32 transformed functions contain multiple transformation instances [Coo]. In

Id	Terms	File
1	device−>timer.function = dasd_device_timeout;	`dasd.c`
2	ns_timer.function = ns_poll;	`nicstar.c`
3	ndlc−>t1_timer.function = ndlc_t1_timeout;	`ndlc.c`
4	ndlc−>t2_timer.function = ndlc_t2_timeout;	`ndlc.c`
5	ntimer.function = nmi_wdt_timer;	`nmi.c`
6	dev−>bulk_timeout.function = au0828_bulk_timeout;	`au0828-dvb.c`
7	device−>timer.data = (unsigned long) device;	`dasd.c`
8	ns_timer.data = 0UL;	`nicstar.c`
9	ndlc−>t1_timer.data = (unsigned long)ndlc;	`ndlc.c`
10	ndlc−>t2_timer.data = (unsigned long)ndlc;	`ndlc.c`
11	dev−>bulk_timeout.data = (unsigned long) dev;	`au0828-dvb.c`

Id	Specific AU	Correct AU	Generic AU
1	$?_1$−>timer.function = $?_2$;	$?_1$.function = $?_2$;	$?_1.?_2 = ?_3$;
2	$?_1$.function = ns_timer;	$?_1$.function = $?_2$;	$?_1.?_2 = ?_3$;
3	$?_1$−>t1_timer.function = $?_2$;	$?_1$.function = $?_2$;	$?_1.?_2 = ?_3$;
4	$?_1$−>t2_timer.function = $?_2$;	$?_1$.function = $?_2$;	$?_1.?_2 = ?_3$;
5	$?_1$.function = nmi_wdt_timer;	$?_1$.function = $?_2$;	$?_1.?_2 = ?_3$;
6	$?_1$−>bulk_timeout.function = $?_2$;	$?_1$.function = $?_2$;	$?_1.?_2 = ?_3$;
7	$?_1$−>timer.data = $?_2$;	$?_1$.data = $?_2$;	$?_1.?_2 = ?_3$;
8	$?_1$.data = 0UL;	$?_1$.data = $?_2$;	$?_1.?_2 = ?_3$;
9	$?_1$−>t1_timer.data = $?_2$;	$?_1$.data = $?_2$;	$?_1.?_2 = ?_3$;
10	$?_1$−>t2_timer.data = $?_2$;	$?_1$.data = $?_2$;	$?_1.?_2 = ?_3$;
11	$?_1$−>bulk_timeout.data = $?_2$;	$?_1$.data = $?_2$;	$?_1.?_2 = ?_3$;

Tab. 4.1: Terms and some possible anti-unifiers

our case, the too generic anti-unifier appears between 2 and 4 times per function whereas the correct one appears between 1 and 2 times.

To sum up, we want to lower the weight of anti-unifiers that are either too specific, manifested as rarely occurring across the set of examples, or too generic, manifested as occurring frequently within a single function. This goal is very similar to the goal of the *term frequency – inverse document frequency (TF-IDF)* [RU11] process used in text mining. TF-IDF is used to highlight words that particularly characterize the meaning of a given document in a corpus. Our approach requires the inverse notion, i.e., anti-unifiers that are common to many documents (i.e., functions), but do not occur too frequently in any given document.

Concretely, our approach uses a process that we can call *function frequency – inverse term frequency (FF-ITF)*. In FF-ITF terms are anti-unifiers and a term weight increases with the number of functions that contains nodes matched by this term (function frequency), and decreases with the number of nodes matched by this term which appear in the same function (inverse term frequency).

The first step is to count how many times each anti-unifier appears. To do so, we represent each node as a set of anti-unifiers that can match this node, from very abstract ones, that are likely to be shared by several nodes, to very concrete ones.[4] To limit the number of possible anti-unifiers for each node we only consider anti-unifiers that at least abstract over local variables.

Then, given an anti-unifier $\mathcal{A}$, a set of functions F and a particular function $f \in F$ that contains the set of nodes N_f, we define our weight $w_{\mathcal{A},f}$ as:

$$\begin{aligned} \text{FF}_{\mathcal{A}} &= \frac{|\{f' \in F : \mathcal{A} \in f'\}|}{|F|} \\ \text{ITF}_{\mathcal{A},f} &= \log \frac{|N_f|}{|\{n \in N_f : \mathcal{A} \in n\}| + 1} \\ w_{\mathcal{A},f} &= \text{FF}_{\mathcal{A}} \times \text{ITF}_{\mathcal{A},f} \end{aligned}$$

This weight function is closely related to the one used in TF-IDF for text mining. It complies with the observations we made when describing the anti-unifiers in table 4.1.

> **!** Note that the weight of an anti-unifier is defined per function. Consequently, in case of multiple instances of a change in a function all anti-unifier weights of modified terms in that function are impacted in the same way.

4.4.2 Noise detection

The result of the node weighting gives a set of weighted anti-unifiers for every modified CFG-Diff node of each example. The next step is to separate noise from relevant nodes.

Our weight function is constructed so that anti-unifiers have low weight when they appear too often within a single function or they do not appear in many other functions. As noise is composed of very unpopular code fragments, all its anti-unifiers are either too generic or have very low function frequency, and as a consequence,

[4] To capture changes that only replace a function call to another, we also consider anti-unifiers that match function calls in the subexpressions of the node.

have low weights. On the contrary relevant nodes typically have at least one anti-unifier with a high weight. Thus it is possible to distinguish noise nodes from relevant nodes by looking at the weight of their highest-weighted anti-unifier.

To decide if a node is relevant, we compare the weight of the node's highest weighted anti-unifier to the average of the weights of the highest weighted anti-unifier of each node. If it is below a certain distance from this average, the node is considered to be noise. In a noise detection experiment for Linux kernel changes, we estimated that the ideal distance to detect noise without producing false positives was 3 standard deviations below the mean.

Nodes marked as noise are dropped from further processing.

4.4.3 Clustering

The next step is to group together nodes that share a common high-weighted anti-unifier. We proceed with the clustering of the nodes not identified as noise.

Our approach first assigns to each node a characteristic vector encoding the weights of all anti-unifiers for this node. To create this vector we first associate an index to every anti-unifier encountered in our examples. This index will mark a position in our characteristic vectors. Then, for each node we create a characteristic vector. Position i in the vector has the weight for the node of the anti-unifier of index i. If the node does not contain the anti-unifier of index i, the weight is set to zero in the node's vector.

Once we obtain numerical vectors, we can reuse efficient clustering approaches. Our approach uses agglomerative hierarchical clustering,[5] a method to form clusters starting with clusters containing singletons and iteratively merging the closest clusters at each step. This approach has already been used for document classification in conjunction with TF-IDF weighting [ZK02]. This method needs two parameters, a metric to evaluate distances between two vectors and a method to compute the distance between two clusters.

We now describe the parameters used and give an intuition about why this method with these parameters works best for our purpose. We need to remember that we want to cluster vectors together, each vector representing a CFG-Diff node. Each non-zero component of each vector represents the weight of an anti-unifier for the node. Consequently common anti-unifiers between two vectors are all the non-zero

[5] We evaluated multiple clustering approaches and agglomerative hierarchical worked best.

components that the vectors have in common. When choosing a metric to compute the distance between two nodes, we take care that the metric does not penalize nodes with multiple instances. After all, nodes with the same anti-unifiers should be as close as possible regardless of the context surrounding those nodes. Consequently, we use the cosine distance, as defined as below:

$$\mathrm{d}_{\cos}(a, b) = 1 - \frac{a.b}{\|a\| \, \|b\|} \tag{4.1}$$

This distance normalizes the norm of each node and so looks only at the relative magnitude of each anti-unifier weight instead of their absolute values. For the method to compute the distance between clusters, we choose the complete linkage method which sets the distance between two clusters as the maximum value of the distance of all pairs of members from the two clusters.

The number of clusters is determined using the best average *Silhouette score* [Rou87], for all possible numbers of clusters. The silhouette score estimates the quality of a clustering, by comparing the distance between members of the same cluster to the distance between members of different clusters. After this procedure we obtain groups of nodes that are very similar, and that will be transformed to rule fragments in the next step.

We illustrate this procedure with the `init_timer` change. The clusters are shown on the left side of Figure 4.2. Each element of a cluster is annotated with the example from which it comes and its position in that example.

For each cluster, we create its rule fragments, which retain the common parts of the terms in the cluster and abstract the subterms that are not common to all of the terms as *metavariables*, *i.e.*, elements that can match any term. The right side of Figure 4.2 shows the rule fragment for each cluster.

At this step we do not know which metavariables represent that same terms and should have the same name. Consequently each used metavariable has a generic name, like `E0`, `E1`, ...

4.4.4 Rule fragments

Rule fragments represent sets of similar terms, but do not provide any control-flow information. In order to prepare for the next step, which constructs a semantic patch rule proposition based on control-flow constraints, our approach next expands each rule fragment into the fragment of a control-flow graph that the rule fragment

Clustered terms	Rule fragments

device−>timer.function = dasd_device_timeout;
ns_timer.function = ns_poll;
ndlc−>t1_timer.function = ndlc_t1_timeout;
ndlc−>t2_timer.function = ndlc_t2_timeout;
ntimer.function = nmi_wdt_timer;
dev−>bulk_timeout.function = au0828_bulk_timeout;
	E0.function = E1;

device−>timer.data = (unsigned long) device; ns_timer.data = 0UL; ndlc−>t1_timer.data = (unsigned long)ndlc; ndlc−>t2_timer.data = (unsigned long)ndlc; dev−>bulk_timeout.data = (unsigned long) dev;	E2.data = E3;

init_timer(&device −>timer);
init_timer(&ns_timer);
init_timer(&ndlc−>t1_timer);
init_timer(&ndlc−>t2_timer);
init_timer(&n_timer);
init_timer(&dev−>bulk_timeout);
	init_timer(&E4);

Tab. 4.2: Found clusters and their rule fragments

constituent node fragments represent, called a graph fragment. A graph fragment is composed of at least one node that contains the rule fragment, and a non-empty list of pairs of entry and exit points. Multi-node graph fragments are created for complex control-flow structures such as conditionals and loops. Figure 4.6 shows some fragments for our `init_timer` example, with the entry points at the top and the exit points at the bottom.

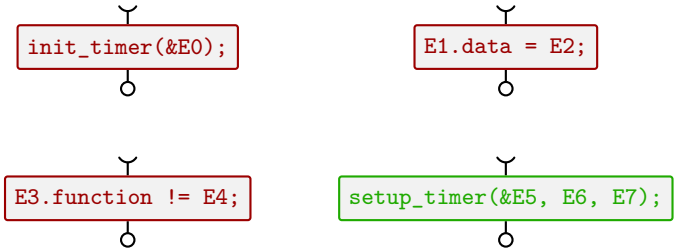
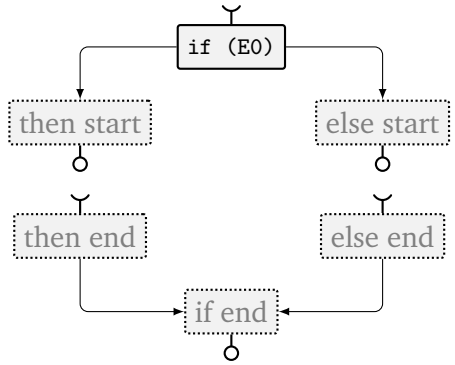


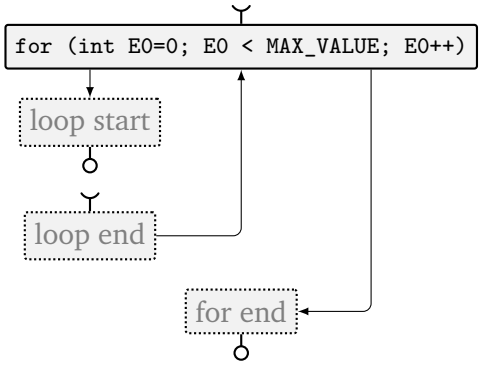
Fig. 4.6: `init_timer` graph fragments

Figure 4.7 show two examples of multi-node graph fragments. We can notice that these graph fragments have multiple entry and exit points. The `if` graph fragment has 3 entry and 3 exit points, organized as the following pairs: (`if(E0)`, `if end`), (`then end`, `then start`) and (`else end`, `else start`). The first pair is called the main

connection points, all other pairs are called secondary connection points. Multi-node fragments are always build so that the entry point of the main connection points dominates the exit point of this main pair and that the exit point of the main connection points post-dominates the entry point. This relation is inverted for the secondary connection points.



(a) An if graph fragment



(b) A for loop graph fragment

Fig. 4.7: Examples of multi-node graph fragments

4.5 Assembling rule fragments

The idea behind constructing a semantic patch rule is to construct an abstracted control-flow graph, called a *semantic patch rule graph*, by incrementally adding graph fragments as long as the dominance relations in the semantic patch rule graph respect the dominance relations in the example CFGs associated with the fragments.

4.5.1 Checking control-flow dependencies

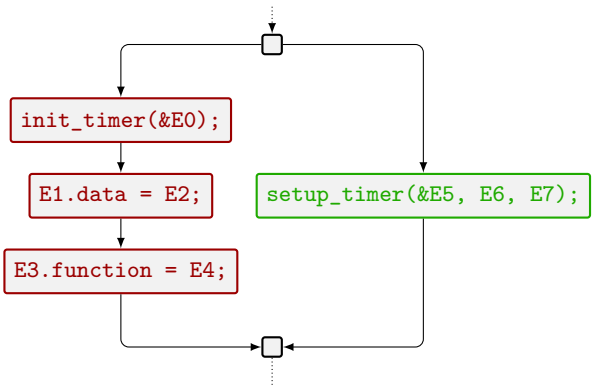
We will first define what it means for a rule graph to respect the dominance relation of the associated example CFG-Diffs. A rule graph is composed of one or several graph fragments, each of which contains a rule fragment matching one or more nodes in the example CFG-Diffs. With each rule graph is associated a set of mappings from nodes in the rule graph to nodes in the examples CFG-Diffs. Each example node is mapped at most once and the example nodes in the same mapping belong to the same example CFG-Diff.

The Figure 4.8 gives an example of a rule graph and its set of mappings for two of our motivating `init_timer` examples seen in section 4.2.1. Each group of rows indicates a different mapping for the selected file[6] instance. Here, all rule fragments are composed of single nodes, each of which contains an abstract fragment found at the previous step.

A rule graph respects the dominance relationship of its associated examples, if and only if, for each mapping of the graph the dominance relationships between all nodes in the rule graph are the same as the dominance relationships between all mapped nodes in the mapping. For instance, in our example rule graph the `init_timer(&E0);` node dominates both the `E1.data = E2;` and `E3.function = E4;` nodes. Consequently for this rule graph to respect the dominance relationship of its examples it is necessary, but not sufficient, that the node `&init_timer(ns_timer);` dominates the nodes `ns_timer.data = 0UL;` and `ns_timer.function = ns_poll;` in the `nicstart.c` file (which is the case).

As we will check dominance relationship to construct rule graphs, we have to retrieve them. Before assembling the graph fragments we compute the dominance relationships, consisting of the sets of dominators and post-dominators, for every node in every example CFG-Diff. For this computation we use the traditional data-flow algorithm for computing dominators, as described by Cooper *et al.* [CHK01]. Post-dominators are computed by reversing the edges of the CFGs and applying the same algorithm.

[6] To be consistent with the way we introduced our examples, we indicate the file of the mapping, but as mentioned earlier each mapping has the scope of a CFG-Diff, which is defined per function and not per file.



(a) Rule graph

File	Mapping

`nicstar.c`	`init_timer(&E0);` → `&init_timer(&ns_timer);`
	`E1.data = E2;` → `ns_timer.data = 0UL;`
	`E3.function = E4;` → `ns_timer.function = ns_poll;`
	`setup_timer(E5, E6, E7);` → `setup_timer(&device->timer, dasd_device_timeout, (unsigned long)device);`

`ndlc.c` *1th instance*	`init_timer(&E0);` → `&init_timer(&ndlc->t1_timer);`
	`E1.data = E2;` → `ndlc->t1_timer.data = (unsigned long)ndlc;`
	`E3.function = E4;` → `ndlc->t1_timer.function = ndlc_t1_timeout;`
	`setup_timer(E5, E6, E7);` → `setup_timer(&ndlc->t1_timer, ndlc_t1_timeout, (unsigned long)ndlc);`

`ndlc.c` *2nd instance*	`init_timer(&E0);` → `&init_timer(&ndlc->t2_timer);`
	`E1.data = E2;` → `ndlc->t2_timer.data = (unsigned long)ndlc;`
	`E3.function = E4;` → `ndlc->t2_timer.function = ndlc_t2_timeout;`
	`setup_timer(E5, E6, E7);` → `setup_timer(&ndlc->t2_timer, ndlc_t2_timeout, (unsigned long)ndlc);`

(b) Associated mapping of abstract fragments to concrete nodes

Fig. 4.8: Example of a rule graph and its associated mapping

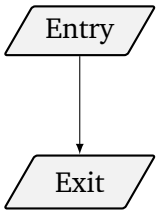
! CFG-Diffs contain information from both CFGs before (deleted + unmodified nodes and edges) and CFGs after (added + unmodified) a transformation. When computing dominance relationship, we process each part independently. Consequently each unmodified node has two sets of dominators and two sets of post-dominators, one for each part.

4.5.2 Assembling deleted graph fragments

We will now see the rule graph construction algorithm, illustrated by the construction of rule graphs for all variants of the `init_timer` change. We will suppose here that we already have found the correct abstract fragments and constructed the graph fragments from them. We also suppose that we already computed the sets of dominators and post-dominators for each node, in each of our modified example CFGs.

The initial graph

The first step is to create an initial semantic patch rule graph, composed of two special nodes, the entry node, which represents the start of the rule graph and the exit node, which represents the end of the rule graph.



By convention, the entry node dominates every node and every node post-dominates it. Similarly the exit nodes post-dominates every node and every node dominates it.

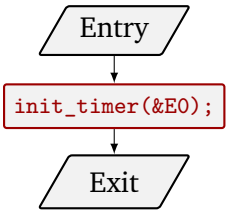
> **!** The entry node of a rule graph is not equivalent of the entry node of a regular control-flow graph, because every node of a rule graph post-dominates the entry node. This is usually not the case with a control-flow graph. The same remark applies to the exit node.

Inserting the first graph fragments

We will now add graph fragments to the rule graph. Remember that we have found the 4 graph fragments shown in Figure 4.6 on page 60. Our approach first chooses a graph fragment. For this example, let us say that the `init_timer(&E0);` fragment was chosen. We want to know if its possible to insert this graph fragment into our rule graph so that the dominance relations in the associated CFGs are respected. Inserting a graph fragment means replacing edges of the rule graph so that for

each replaced edge, we can connect its source node to one entry point of the graph fragment and connect its destination node to one exit point of the graph fragment.

For the first fragment this is always true due to the properties of the entry and exit nodes. We can then insert this graph fragment into the rule graph, obtaining the following rule graph:

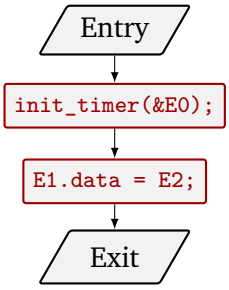


This rule graph is associated with the following mapping:

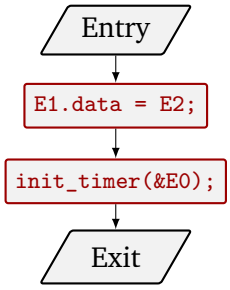
File	Mapping
dasd.c	`init_timer(&E0);` → `&init_timer(&device->timer);`
nicstar.c	`init_timer(&E0);` → `&init_timer(&ns_timer);`
ndlc.c	`init_timer(&E0);` → `&init_timer(&ndlc->t1_timer);`
ndlc.c	`init_timer(&E0);` → `&init_timer(&ndlc->t2_timer);`
nmi.c	`init_timer(&E0);` → `&init_timer(&ntimer);`
au0828-dvb.c	`init_timer(&E0);` → `&init_timer(&dev->bulk_timeout);`

As the rule graph cover all cases of `init_timer(&E0);` usage, the associated graph fragment is removed from the pool of available graph fragments.

We will now try inserting a second graph fragment into our rule graph. This time let us say that we choose to insert the `E1.data = E2;` graph fragment.



(a) First possibility



(b) Second possibility

Fig. 4.9: Two possible ways to insert `E1.data = E2`

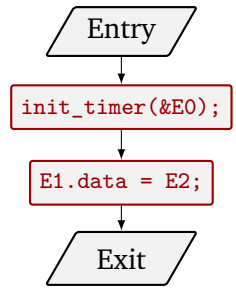
Figure 4.9 shows us that in that case there are two possibilities. One way is to insert `E1.data = E2;` below `init_timer(&E0);`. This respect the dominance relationships found in `dasd.c`, `nicstar.c` and the two instances of `dasd.c`. The other way is to insert this graph fragment above `init_timer(&E0)`, which is coherent with the dominance relationships in the `au0828-dvb.c` example. Either way, inserting this fragment prevents the rule graph from matching the `nmi.c` example which does not have concrete terms associated with the `E1.data = E2;` graph fragment. This problem will allow us to see one of the key points of our approach: *splitting*.

4.5.3 Splitting the semantic patch rule graph

To insert the `E1.data = E2;` graph fragment while keeping consistent dominance relationships between terms matched by the rule graph we have to reduce the number associated examples, *i.e.* to reduce the number of mappings. This operation is called splitting and consists in duplicating the rule graph by splitting its mapping in two parts so that it is possible to insert the graph fragment into one of these new rule graph. Splitting usually happens when there are dominance relationship inconsistencies between examples or when some examples do not contain the term associated with the graph fragment. We will see later that splitting can also happen when trying to find which metavariables should have the same name.

In our example, if we want to insert the `E1.data = E2;` graph fragment below `init_timer(&E0);` we will have to split the rule graph as shown in Figure 4.10.

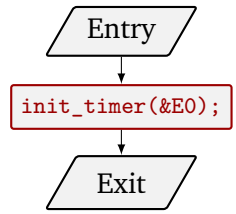
We now obtain two rule graphs, associated with different changes instances. The first rule graph contains our `E1.data = E2;` rule graph fragment. However, there still exists some instances, not matched by our first rule graph, which contains nodes associated with the `E1.data = E2;` rule graph fragment. When a graph fragment is used in a rule graph, but not all nodes associated to the graph fragment have been in the rule graph mapping, the graph fragment is kept in the pool of available fragments. However associated nodes already used in a rule graph mapping cannot be used in another mapping. If the set of available associated nodes of a graph fragment is empty, this graph fragment is removed from the pool of available graph fragment.



(a) First rule graph

Mapping
`init_timer(&E0);` → `&init_timer(&device->timer);` `E1.data = E2;` → `device->timer.data = (unsigned long) device;`
`init_timer(&E0);` → `&init_timer(&ns_timer);` `E1.data = E2;` → `ns_timer.data = 0UL`
`init_timer(&E0);` → `&init_timer(&ndlc->t1_timer);` `E1.data = E2;` → `ndlc->t1_timer.data = (unsigned long)ndlc;`
`init_timer(&E0);` → `&init_timer(&ndlc->t2_timer);` `E1.data = E2;` → `ndlc->t2_timer.data = (unsigned long)ndlc;`

(b) Mapping for the first rule graph



(c) Second rule graph

Mapping
`init_timer(&E0);` → `&init_timer(&ntimer);`
`init_timer(&E0);` → `&init_timer(&dev->bulk_timeout);`

(d) Mapping for the second rule graph

Fig. 4.10: Split rule graphs and their associated mappings

A note on mapping

When a rule graph can match multiple transformation instances in the same function obtaining the set of mappings associated with the rule graph is not immediate. This is the case for our rule graph (a) shown in Figure 4.10.

Mapping
`init_timer(&E0);` → `&init_timer(&device->timer);` `E1.data = E2;` → `device->timer.data = (unsigned long) device;`
`init_timer(&E0);` → `&init_timer(&ns_timer);` `E1.data = E2;` → `ns_timer.data = 0UL`
`init_timer(&E0);` → `&init_timer(&ndlc->t1_timer);` `E1.data = E2;` → `ndlc->t2_timer.data = (unsigned long)ndlc;`

Fig. 4.11: A correct but undesirable mapping

The mapping shown at Figure 4.11 is a possible mapping for rule graph (a) of Figure 4.10 that is correct but undesirable. The mapping is correct because it respects all the dominance relationships of the associated examples, but undesirable because it mixes the two transformation instances of the `ndlc.c` file. Below, we recall the diff for this double instances transformation:

File: drivers/nfc/st-nci/ndlc.c

```
1 - init_timer(&ndlc->t1_timer);
2 - ndlc->t1_timer.data = (unsigned long)ndlc;
3 - ndlc->t1_timer.function = ndlc_t1_timeout;
4 -
5 - init_timer(&ndlc->t2_timer);
6 - ndlc->t2_timer.data = (unsigned long)ndlc;
7 - ndlc->t2_timer.function = ndlc_t2_timeout;
8 + setup_timer(&ndlc->t1_timer, ndlc_t1_timeout,(unsigned long)ndlc);
9 + setup_timer(&ndlc->t2_timer, ndlc_t2_timeout,(unsigned long)ndlc);
```

We see that our incorrect mapping uses the expression `ndlc->t1_timer` from the first transformation instance on the `init_timer` call, but the expression `ndlc->t2_timer` from the second instance for the data field initialization.

When adding a graph fragment to an existing rule graph, one must choose the candidate CFG-Diff nodes to add to each mapping group. It is not an issue when there is at most only one instance of the transformation per function because, as stated earlier, all mapped nodes in each mapping group must come from the same CFG-Diff, hence from the same function. However, when multiple instances exists in a function we must choose the correct ones. The strategy used in this approach is to pick the closest[7] ones to the nodes already in the mapping.

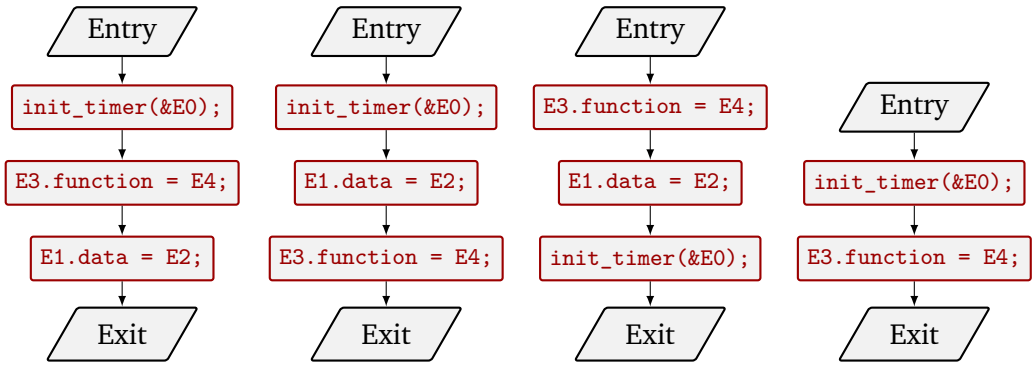
4.5.4 Assembling added graph fragments

Using the methods cited in the last two sections, we can construct the four rule graphs shown in Figure 4.12, that uses all available removed graph fragments.

In our approach we always try to insert all available removed graph fragments before trying to insert added graph fragments.

Now we can try to add the last graph fragment: `setup_timer(&E5, E6, E7);`. Since this is an added graph fragment we cannot simply use dominance relationships to removed nodes, since dominator and post-dominator notions apply only to CFGs

[7] The binary relation $A \nabla B$, defined as A dominates B and B post-dominates A, is a total order on the set of candidate nodes. In this case it is possible to define the closest node.



(a) First rule graph **(b)** Second rule graph **(c)** Third rule graph **(d)** Fourth rule graph

Fig. 4.12: Obtained rule graphs

and not CFG-Diffs. As we stated earlier, we compute the dominance relationships separately for the before and the added parts of the CFG-Diff. The only nodes in common between these two parts are the unmodified nodes. Consequently, we will use unmodified nodes to check dominance relationships with our added graph fragment.

To add an added graph fragment, we first collect the set of closest unmodified predecessors of all entry ports, and the closest unmodified successors of all exit ports for the graph fragment. When trying to insert an added graph fragment the dominance relationships to removed nodes are checked using these unmodified successors and predecessors, instead of the added nodes.

By using this method the `setup_timer(&E5, E6, E7);` graph fragment can be inserted immediately into the first, third and fourth rule graph of Figure 4.12. The second rule must be split, because in our `nicstar.c` example, the closest unmodified successor is right after `init_timer` instead of being right after the `function` field initialization.

At the end we obtain the 5 rule graphs described by Figure 4.13. In each of these rule graphs the empty unmodified nodes before and after the `setup_timer(&E5, E6, E7);` node represent this node unmodified predecessors and successors. Once inserted, unmodified predecessors and successors are used to check control-flow relationship as any other nodes, but they do not carry any rule fragment and will not be printed when we generate semantic patch rules from these rule graphs.

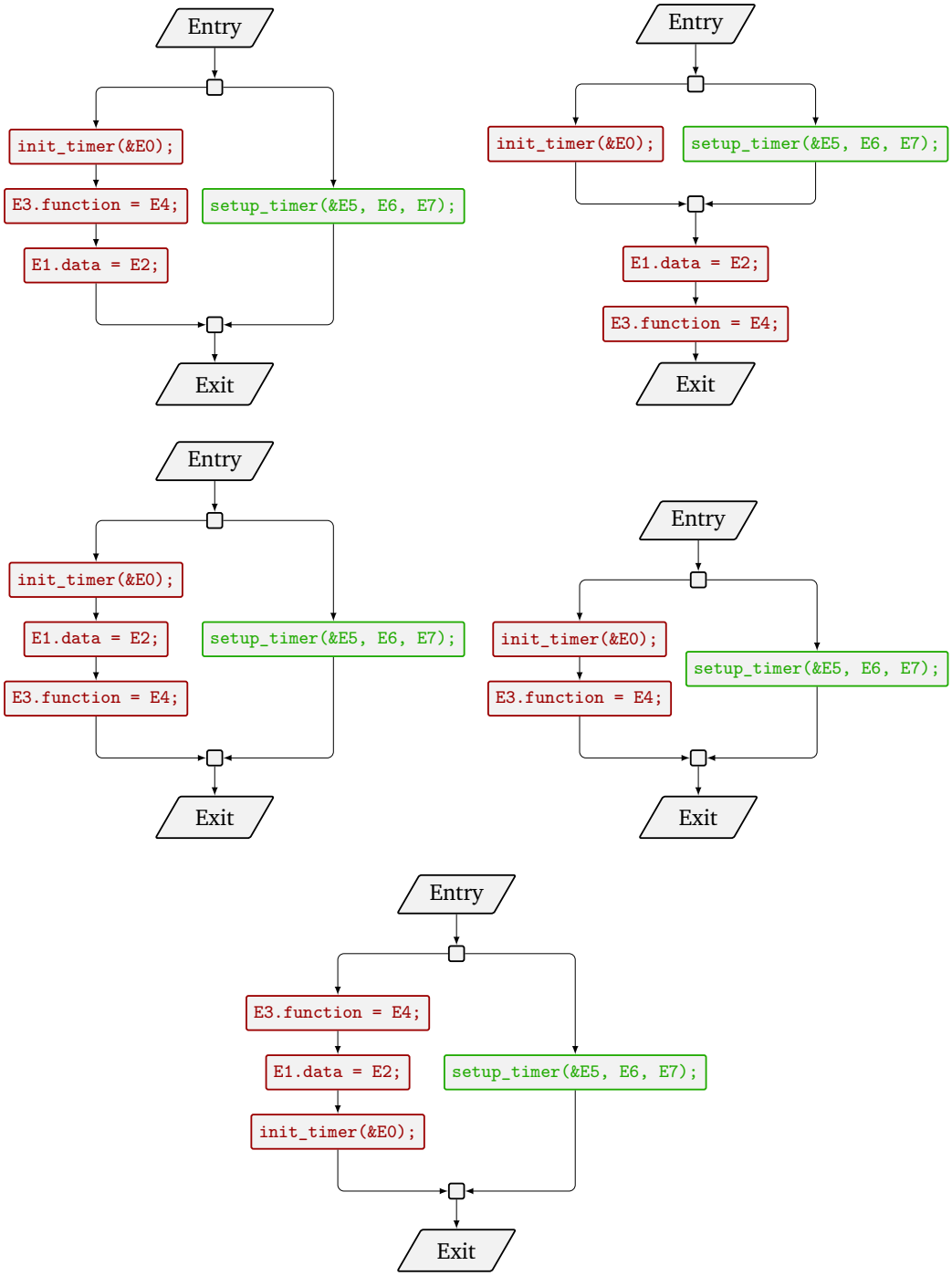


Fig. 4.13: Obtained rule graphs after assembling all fragments

4.6 From rule graphs to semantic patch rules

At this point, we have used all the available graph fragments to construct our semantic patch graph rules. Still a few key points need to be solved in order to obtain semantic patch rules from these graphs:

1. In each graph we must find common metavariables so that we can describe data-flow relationships between the changed terms.

2. We need to introduce the "..." SmPL sequence operator.

3. We have to transform each rule graph into a textual semantic patch rule.

We will discuss each of these points hereafter.

4.6.1 Handling metavariables

At this time every metavariable in every rule fragment has a generic name, such as `E0` or `E1`. Now we need to determine which of these metavariables are required to match the same terms, and thus should be the same. To solve this problem we will look at the concrete subterms the metavariables represent in each example.

Each rule graph has a list of mappings from rule fragments to concrete terms in the examples. Consequently, each metavariable can be associated with a list of subterms, one subterm for each mapping, that each metavariable represents. If two metavariables in the same rule graph share the same list of subterms the metavariables are considered to be required to match the same code fragments and so they take the same name.

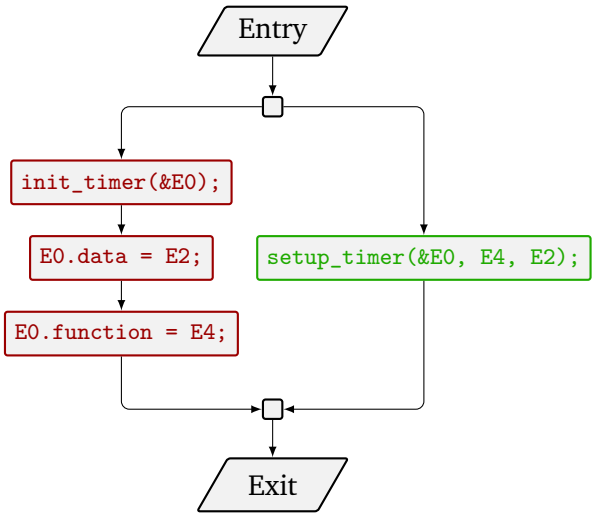
Let us look at the analysis of metavariables for the third rule graph obtained in Figure 4.13 (page 70). We first list the different subterms represented by the metavariables for the different mappings.

Metavariables	Subterms	
E0	`ndlc->t1_timer`	`ndlc->t2_timer`
E1	`ndlc->t1_timer`	`ndlc->t2_timer`
E2	`(unsigned long)ndlc`	`(unsigned long)ndlc`
E3	`ndlc->t1_timer`	`ndlc->t2_timer`
E4	`ndlc_t1_timeout`	`ndlc_t2_timeout`
E5	`ndlc->t1_timer`	`ndlc->t2_timer`
E6	`ndlc_t1_timeout`	`ndlc_t2_timeout`
E7	`(unsigned long)ndlc`	`(unsigned long)ndlc`

We this information, we can construct 3 groups of equivalent metavariables:

- E0, E1, E3 and E5
- E2 and E7
- E4 and E6

As a consequence, we can construct this updated graph rule:



Unbound metavariables

There is still one issue with this strategy: in some cases some metavariables are only present in the added parts of the rule graph, also called unbound metavariables. In this case we will not be able to produce a valid semantic patch since all metavariables used in added parts must be present in removed or unmodified parts. This case happens in the fourth rule graph of Figure 4.13 (page 70).

Unbound metavariables can be:

1. Bounded to unmodified parts of the code
2. Or correctly bound to other metavariables, but only in a subset of examples
3. Or be constant values that cannot be bound to other values

Our approach does not handle the first case. Indeed, finding unmodified terms that share metavariables requires creating unmodified graph fragments which requires clustering similar unmodified concrete terms. Our clustering algorithm does not allow us to find such fragments, since it supposes that most terms belong to the transformation. The only exception is when unbound added metavariables could be bound to unmodified function header parameters. In that case it is possible to create function header rule fragment, since they do not require clustering.

We can solve the second case by looking for a subset of examples in which the metavariable could be bound and perform a rule graph split on this subset. For this case it is possible make unbound metavariables more concrete if this enables them to be bound after this step.

For the third case we can replace the metavariable to its constant value. If there are multiple possible constants for the same metavariable, we look for the subset in which the metavariable has a single value and we perform a split on the rule graph.

In our fourth rule graph, the unbound metavariable `E7` maps to a constant with the single value of `OUL`.

4.6.2 The SmPL sequence operator

If we look back at the semantic patch rules our human expert wrote (page 44) we can see that we are missing the "..." sequence operator in our rule graphs. One strategy could be to insert ... between every pair of connected deleted nodes, but in that case the generated rules will be too general, and we will risk performing undesired changes.

A more conservative strategy is to insert the sequence operator only when we are sure it is needed. We know for sure that we need to generates sequence operator when there exists an unmodified node between removed nodes in a CFG-Diff. As a consequence, we generates sequence operator in our graph-rule at the following locations:

- Between two removed nodes in the rule graph if the rule graph represents any example containing an unmodified nodes between the removed nodes.
- At the place of an unmodified predecessor that is not preceded by the entry node.
- At the place of an unmodified successor that is not followed by the exit node.

Figure 4.14 shows an example of adding the "..." sequence operator to the second rule graph of Figure 4.13 (page 70). Here, the rule graph matched the first criterion for the sequence operator location. It is also the only rule graph where it is possible to insert this operator.

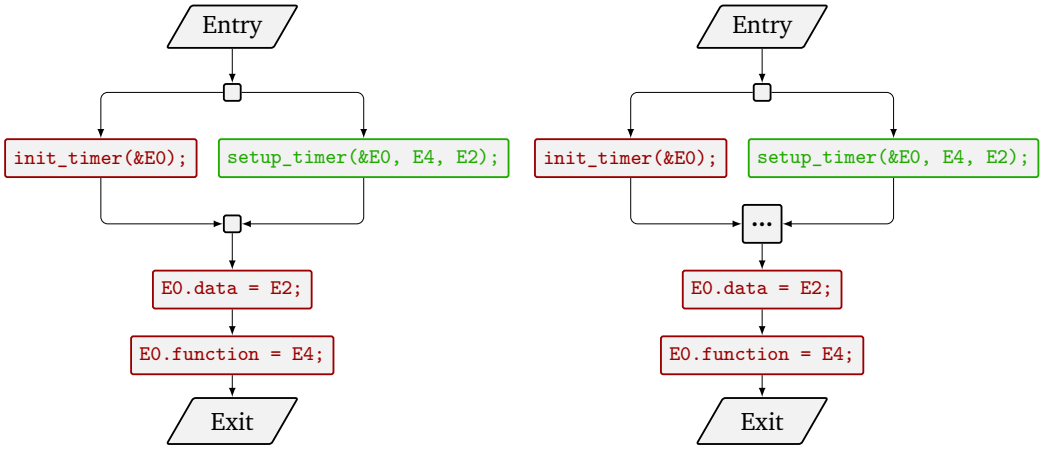


Fig. 4.14: Example of a rule graph before and after the sequence operator insertion

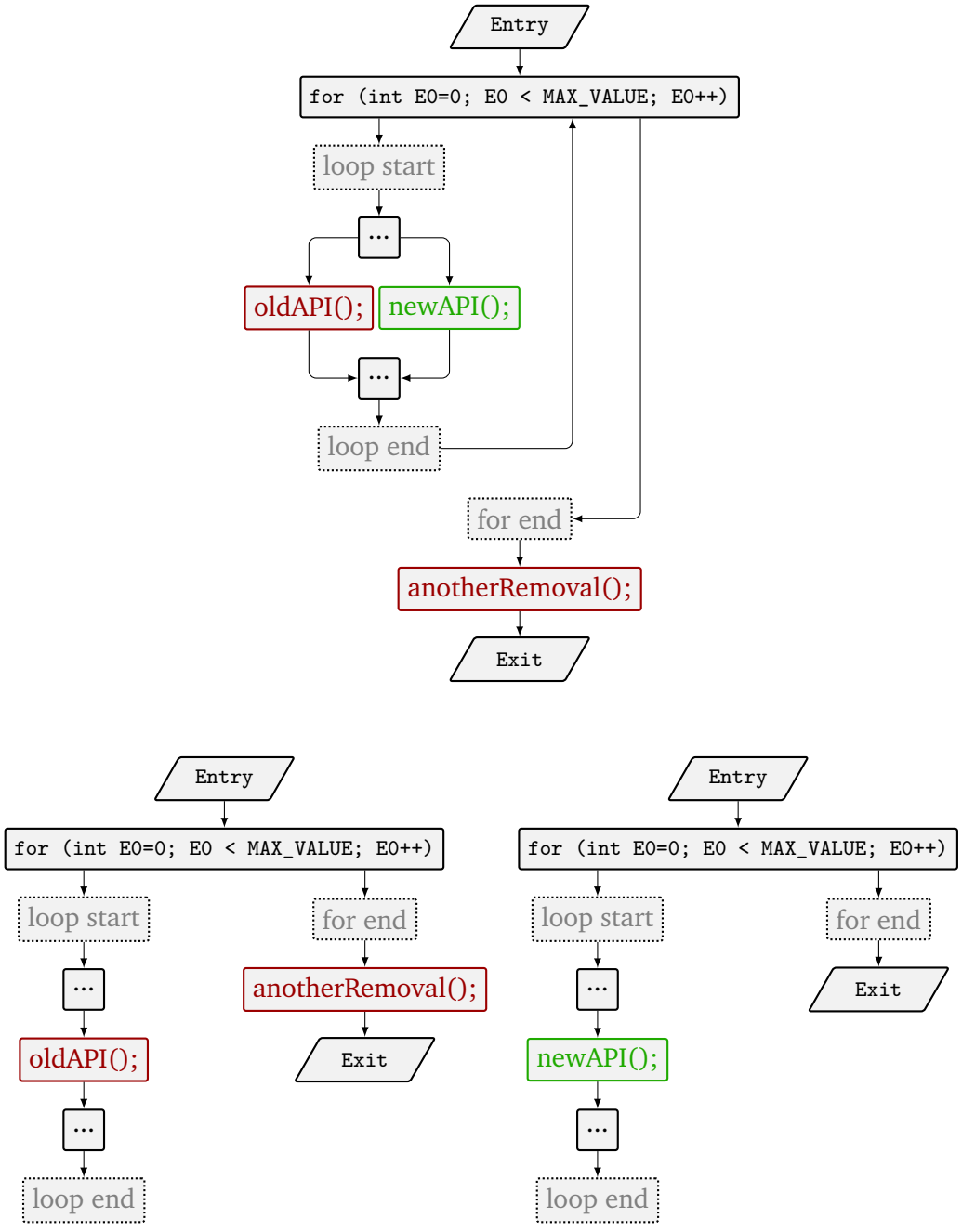
4.6.3 Generating semantic patch rules from graphs

At this point we have completed our rule graphs. The next step is to transform each rule graph into a textual semantic patch rule.

To do so we need a method to traverse the rule graph in a way that generates the correct semantic patch rule output. The main challenge is to print the added terms of the semantic patch rule at the correct location. Similarly to when we built the CFG-Diffs, we will process the before and after parts of each semantic patch rule separately before relying on the unmodified terms to bind together these two parts.

Our strategy requires to first compute the *dominator-tree* [LT79] for each part (before and after) of each rule graph. In a dominator tree, a “node’s children are those

nodes it immediately dominates". Figure 4.15 shows the two dominator trees for the following rule graph[8]:



(a) Dominator tree for the before part

(b) Dominator tree for the after part

Fig. 4.15: Examples of dominator trees

[8]We switch our example for this section because the dominator trees for our `timer` example are really uninformative.

For each pair of dominator trees we compute the pre-order traversal of each tree. In our case we obtain the following two lists, with elements in brackets representing non-printable nodes:

```
   [Entry]
   for (int E0=0; E0 < MAX_VALUE; E0++)
   [loop start]
   ...
 - oldAPI();
   ...
   [loopend]
   [forend]
 - anotherRemoval();
   [Exit]
```

Listing 4.1: Traversal of the before tree

```
   [Entry]
   for (int E0=0; E0 < MAX_VALUE; E0++)
   [loop start]
   ...
 + newAPI();
   ...
   [loopend]
   [forend]
   [Exit]
```

Listing 4.2: Traversal of the after tree

We then merge the two lists together, using the unmodified nodes as a reference. We obtain the following list:

```
   [Entry]
   for (int E0=0; E0 < MAX_VALUE; E0++)
   [loop start]
   ...
 - oldAPI();
 + newAPI();
   ...
   [loopend]
   [forend]
 - anotherRemoval();
   [Exit]
```

Listing 4.3: Merged lists

We can now replace `loopstart` by "{" and `loopend` by "}", then remove all other elements in brackets and declare the used metavariables. After this procedure we obtain the desired semantic patch rule:

```
@@ expression E0; @@
  for (int E0=0; E0 < MAX_VALUE; E0++)
  {
  ...
- oldAPI();
+ newAPI();
  ...
  }
- anotherRemoval();
```

If we apply the same method to our `timer` rule graphs we will obtain these 5 rules:

```
@@ expression E0, E2, E4; @@
- init_timer(&E0);
- E0.function = E4;
- E0.data = E2;
+ setup_timer(&E0, E4, E2);
```

Listing 4.4: Rule 1

```
@@ expression E0, E2, E4; @@
- init_timer(&E0);
+ setup_timer(&E0, E4, E2);
  ...
- E0.data = E2;
- E0.function = E4;
```

Listing 4.5: Rule 2

```
@@ expression E0, E2, E4; @@
- init_timer(&E0);
- E0.data = E2;
- E0.function = E4;
+ setup_timer(&E0, E4, E2);
```

Listing 4.6: Rule 3

```
@@ expression E0, E2, E4; @@
- init_timer(&E0);
- E0.function = E4;
+ setup_timer(&E0, E4, E2);
```

Listing 4.7: Rule 4

```
@@ expression E0, E2, E4; @@
- E0.function = E4;
- E0.data = E2;
- init_timer(&E0);
+ setup_timer(&E0, E4, E2);
```

Listing 4.8: Rule 5

4.7 Rule ordering

We obtained a set of semantic patch rules, but we do not have a semantic patch yet. To construct a semantic patch we need to decide an order for this set of rules. Indeed, since Coccinelle processes the rules in order, applying each new rule to the results of applying all the preceding rules, the final results of the semantic patch will depend on the rule order.

As we want to obtain the best overall results for our semantic patch, we must rank the rules we previously obtained. To do so our approach uses self-evaluation of the semantic rules on the examples we learned the rules from. We will first describe how to evaluate semantic patch rules and then how to order them in a semantic patch.

4.7.1 Semantic patch rule evaluation

From a set of semantic patch rules we can wonder if this set, with a certain order, produce undesirable results, or is missing some changes. We must first choose the level at which we want to investigate the effect of a sequence of semantic patch rules. In our case a natural choice is the CFG node, which translates roughly to a line of code.

To accurately describe the results a set of semantic patch rules produces, we have to introduce some metrics. We will first present the building blocks of these metrics, the classification of results in terms of correct node changes in a CFG-Diff. We reuse 4 common categories for binary classification of results [HS15]:

- True positive: A node is changed correctly and should have been changed at that position.
- True negative: A node is not changed and should not have been changed at that position.
- False positive: A node is changed either incorrectly or should not have been changed at that position.
- False negative: A node is not changed but should have been changed at that position.

Thus, to classify each change produced by our set of semantic patch rules we need to compare the generated results to the ground truth, which for our self-evaluation are the changes in the examples we learned from.

From this classification, we can use normalized metrics usually applied for evaluation of program transformation tools.

Precision

Precision is defined as follows [SW11]:

$$\text{precision} = \frac{\#\text{True positives}}{\#\text{True positives} + \#\text{False positives}} \tag{4.2}$$

Precision gives the ratio of correct changes over the total number of changes. When precision is low, most of the changes made are incorrect. When precision is high, most of the changes made are correct, but there are not necessarily a lot of made changes.

Recall

Recall is the counterpart of the precision that measures the correct amount of changes made. It is defined as follows [SW11]:

$$\text{recall} = \frac{\#\text{True positives}}{\#\text{True positives} + \#\text{False negatives}} \tag{4.3}$$

We can see that recall measures the ratio of correct changes made over the total number of changes that needed to be made.

F-score

Sometimes is useful to have a synthesis of both recall and precision in one number. For that we can use the F-score, defined as follows:

$$F_\beta = \left(1 + \beta^2\right) \frac{\text{precision} \times \text{recall}}{\beta^2 \times \text{precision} + \text{recall}} \tag{4.4}$$

Common F-score metrics used are: $F_{0.5}$, F_1 or F_2, which gives more weight to precision than recall, gives the same weights to both precision and recall or give more weight to recall than precision, respectively.

Computing these metrics

We have seen that in order to obtain precision, recall and F-score we need to compute the number of true positives, false positives and false negatives.

To do so we will build 3 different CFG-Diffs:

- The *truth CFG-Diff*, which is the difference between the before CFG and the after CFG of the ground truth.
- The *generated CFG-Diff*, which is the difference between the before CFG and the CFG after the applying an inferred semantic patch.
- The *discrepancy CFG-Diff*, which is the difference between the after CFG of ground truth and the after CFG of the generated changes.

For each CFG-Diff, we extract the set of modified nodes. We call those sets: S_{truth}, S_{gen} and S_{disc}. Then, we define our categories as follows:

- **True positives (TP)**: $(S_{\text{gen}} \cap S_{\text{truth}}) \setminus S_{\text{disc}}$
- **False positives (FP)**: $S_{\text{gen}} \cap S_{\text{disc}}$
- **False negatives (FN)**: $(S_{\text{truth}} \setminus S_{\text{gen}}) \bigcup (S_{\text{gen}} \cap S_{\text{truth}} \cap S_{\text{disc}})$

It is important to note that a node in both S_{truth} and S_{gen} is not necessarily a true positive. For instance, a node that needed to be added but that is added at the wrong location is both in S_{truth} and S_{gen} but also in S_{disc}.

From these sets we can compute the metrics introduced earlier. This will help us to rank semantic patch rules since we will be able to associate numerical values to a sequence of rules.

4.7.2 Rule ordering and subsuming

If we look at the obtained semantic patch rules on page 77, we can notice that there exist some rules that can always be applied at the same location as other rules. For instance, rule 4 can be applied at every location matched by rule 1, and rule 2 can be applied at every location matched by rule 3. If we apply rule 4 instead of rule 1 then the generated change is incorrect. But if we apply rule 2 instead of rule 3 the change is correct, making rule 3 redundant. This highlight the two notions we will now see: *rule ordering* and *subsuming*.

Rule subsuming

We saw that some generated rules can be redundant with other rules and need to be removed from the final semantic patch. We said that a rule A subsumes another rule B, if A produces as many correct changes as B and fewer incorrect changes than B. Formally, we check that TP_A is a superset of TP_B and FP_A is a subset of FP_B.

Semantic patch rules that are subsumed by another are eliminated from the final set of rules. In our case rule 3 is subsumed by rule 2. Consequently it will not be used in the final semantic patch.

Rule ordering

Now, we need to decide an order for our remaining semantic patch rules. Since rules can interact with each other, we must in theory compute all combinations of rule order to find the best one. The best combination in the one that maximize both precision and recall. Since we have two values to maximize, it is not always possible to define the best combination. Instead we can look at the F-score, which gives us a synthesis of both metrics. We use here the $F_{0.5}$ score which favor precision over recall since false positives are more undesirable than false negatives.

Still evaluating a semantic patch on the set of our input examples is very costly. Evaluating a semantic patch requires applying the semantic patch with the Coccinelle engine on our examples, then creating CFG-Diffs for the results and comparing them to the ground truth. In some cases that process can take several seconds. As a consequence, we will not try to find an optimal solution, which has a complexity of $\mathcal{O}(n!)$. Instead we evaluate an approximation of the best results, by comparing every possible order for every pair of rules.

Let R_1 and R_2 be two semantic patch rules. We compare the $F_{0.5}$ score for the two possible orders. If $F_{0.5}([R_1, R_2])$ is greater than $F_{0.5}([R_2, R_1])$, we said that R_1 must be placed before R_2. If $F_{0.5}([R_1, R_2])$ is less than $F_{0.5}([R_2, R_1])$, we said that R_1 must be placed after R_2. If the two values are equal, we said that the two rules are independent.

With these comparisons we can place rules into a directed graph. In this graph, the nodes are the semantic patch rules and an edge from node R_1 to node R_2 means that rule R_1 must be placed before rule R_2. With this structure, we can use Kahn's algorithm [Kah62] for topological sorting. If the graph does not contain any cycle, we will obtain a sorted list of rules that respect the ordering relationships we

found. Since we have not defined an order, it is possible that the graph does contain cycles. In this case we first break cycles than apply Khan's algorithm. The ordering relationship will not be respected for elements of the cycle.

With that strategy we obtain an approximate solution with a complexity of $\mathcal{O}(n^2)$ Coccinelle applications.

4.8 Conclusion

In this chapter we have seen a novel approach for learning API-usage update transformations from a set of examples. This approach differs from other approaches in the literature because it discovers variants during the learning algorithm rather than finding them at the start. Moreover abstractions are assembled using dominance notions, which allow the inferred transformations to capture control-flow relationships, a property often overlooked in existing tools. This approach is also the first to recognize that the application of a transformation variant can influence the application of other variants, and so, variants must be ordered, in our case using self-evaluation on the learning examples.

This approach led us to develop several elements:

- A control-flow representation of changes inspired by the GNU-Diff tool.
- A novel code clone clustering approach optimized for clustering changes.
- A procedure for putting together abstractions using control-flow dependencies.

However some challenges still need to be addressed.

Our approach does not cover unmodified rule fragments, so it is impossible to infer correct rules for transformations requiring data-flow or control-flow dependencies to unmodified terms, other than function headers. This is a downside of our clustering strategy, which relies on the fact that most elements to cluster will belong to a group of similar terms. We cannot use this strategy with unmodified terms, because unmodified most terms have no relation to the API-usage and thus will not be similar to other terms. One possible way to solve this problem is to use general code clone detection techniques for the unmodified terms. Another way is to perform the clustering of unmodified terms after constructing rule graphs using only modified graph fragments. That way we could extract only unmodified terms with data-flow dependencies to terms matched by the rule graphs, and so reduce the pool of terms to cluster. However, it is interesting to note that once we identify how to select

unmodified rule fragments, they can be directly used by our approach, as they work similarly to removed rule fragments.

In general the challenge of using unmodified terms in transformation rule is still to be addressed. Indeed, adding unmodified terms to a transformation rule lowers the scope of that rule. Consequently, one has to carefully select which unmodified terms are essential to the transformation and which ones are not.

Another challenge is to cover more of the features provided by SmPL. For instance, with our approach we cannot generate the <... ...> sequence operator which allows a semantic patch pattern to be captured or removed an arbitrary number of times. Also, semantic patch patterns can match multiple terms in the same transformation instance, if they are in different execution path. Our approach supposes that, for every transformation instance, each abstract fragment matches exactly one term in that instance. Learning transformation from examples with a pattern matching multiple terms in different execution paths will require another clustering algorithm as well as a new notion of dominance for groups of nodes instead of individual nodes.

5 Evaluation

In this chapter we will look at the performance of SPINFER, *i.e.* the implementation of the approach we described in the last chapter. The current implementation of SPINFER consists of around 11000 lines of OCaml that take care of all parts of the approach except the clustering. It relies on some Coccinelle internal libraries, mainly to transform source code to control-flow graphs and to pretty print rule fragments. The clustering part consists of 150 lines of Python code and relies on libraries commonly used for large data processing and clustering, namely `numpy`, `scipy` and `scikit-learn`.

The evaluation we will perform in this chapter aims to assess the strengths and weaknesses of our approach, in terms of the metrics we already introduced earlier: *precision*, *recall* and *F-score*.

For this evaluation we target the following three research questions:

- **RQ1:** Is SPINFER successful at learning Linux kernel API usage updates?
- **RQ2:** Can using SPINFER be efficient for Linux kernel developers?
- **RQ3:** What are the shortcomings of our approach?

We will start this evaluation by first describing the methodology we will use to perform experiments on Linux kernel API usage update examples. Then, we will answer each research question by evaluating SPINFER on two datasets of 40 groups of transformation each and analyzing various aspects of the results obtained. At last, we will discuss about the strengths and weaknesses of our approach and future work to do in light of this evaluation.

> **!** SPINFER, the examples used in the evaluation, the scripts to launch the evaluation and the scripts to parse the results are all available at the following url: `https://gitlab.inria.fr/spinfer`

5.1 Methodology

To evaluate SPINFER's capabilities to learn API usage updates from examples we first need to address three questions.

The first one is: **how to measure the quality of semantic patches produced by SPINFER?** It is difficult to evaluate semantic patch quality directly because there may be no ideal semantic patch for a given transformation. Sometimes multiple semantic patches can produce the same results on the same code base without any semantic patch being superior to the others. To keep the comparison simple, we will instead evaluate the results produced by applying inferred semantic patches to a set of examples. To measure the quality of the results produced, we will evaluate both the *precision* and *recall* defined by the correct number of node changes, as seen in Section 4.7.1 on page 78.

The second question is: **against what should we compare the results produced by semantic patches inferred by SPINFER?** Comparing directly to the change examples is not ideal because the change examples can contain noise. Noise can contain incorrect and missed changes and changes that are not part of the overall transformation. Consequently, even the best semantic patch written for a transformation will not have 100% recall and precision when comparing the results it produces against the change examples of this transformation. Since we want to focus on the part of the transformation that could be automated, we will also compare the results produced by the inferred semantic patches against the results produced by semantic patches written by a human expert, either a Linux kernel developer or a Coccinelle expert, for the same transformations.

The third and last question is: **from a set of examples of a transformation, on what subset should SPINFER learn the transformation and on what subset should we evaluate the inferred semantic patch?** To answer this question we propose two experiments:

1. In the first experiment, called the *synthesis experiment*, SPINFER learns the transformation and evaluates the results on the same set of files, which contains all the transformation examples. This experiment evaluates the degree to which SPINFER is able to capture and summarize the changes found in the examples provided to it. It is relevant to a user who wants to understand a previously performed change. Without our tool, such a user has to read through the entire patch and collect all of the different kinds of changes performed, with no way to validate their understanding.

2. The second experiment is called the *transformation experiment*. In this experiment, SPINFER learns the transformation from a reduced set of examples, called the *learning set*, and the results produced are evaluated on the complementary set, called the *validation set*. For each transformation, the learning set consists of the first 10 files in chronological order of modification or half of the files from the transformation example files if the transformation has fewer than 20 file examples. The modification date is retrieved from the commit author date and ties between files in the same commit are broken randomly. This experiment is relevant to the user who wants to change new code by inferring a semantic patch and applying it.

5.2 Answering the research questions

5.2.1 RQ1: Learning API usage updates

In this section we will try to understand how well SPINFER can learn Linux kernel API usage updates. To answer this question we first build a dataset that is representative of real Linux transformations.

Dataset description

We build this dataset, called the *random* dataset, by randomly sampling *large-scale* changes in the Linux kernel. We define a large-scale change as a change from one developer affecting at least 10 files. To construct this dataset we extracted 175 groups of large-scale changes from the changes performed in 2018, using the tool patchparse [PLM06].

Out of these 175 groups we randomly selected 40, and for each of these groups, our human expert wrote a semantic patch performing the transformation they saw in the examples.[1]

Figure 5.1 shows the taxonomy of the random dataset. The top labels indicate the taxonomy categories as seen in Table 2.1 on page 26 in Section 2.3.7. The x-axes represents the level in each category and the y-axis the number of occurrences for each level. The large majority of transformations fit in SPINFER's scope in terms of data-flow relationship handling (C_1). This makes our approach appropriate to

[1]The number of selected group is kept low because writing semantic patches can be very time consuming.

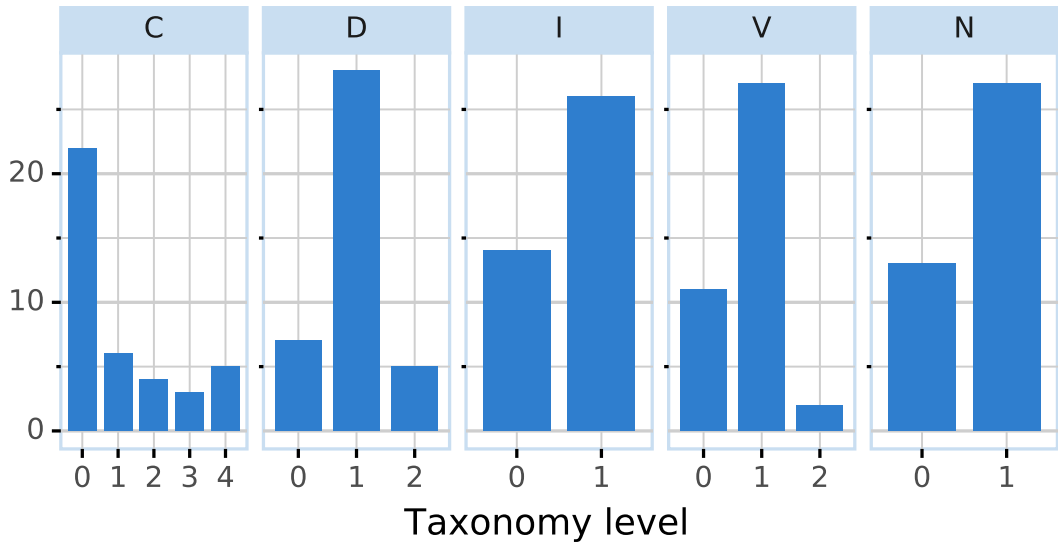


Fig. 5.1: Distribution in terms of the taxonomy for the random dataset

infer transformations for the Linux kernel. The majority of transformations have examples containing multiple instances within a function (I_1), but no transformation has examples with overlapping instances (I_2, not shown in the figure). While multiple variants are common (V_1) only a few transformations require variant ordering (V_2). Surprisingly, the proportion of transformations having examples containing noise (N_1) is very high, with almost two thirds of transformation affected.

> **!** The taxonomy of a group of changes represents the maximum encountered levels for every taxonomy category of all transformation variants for this group. It is possible that some variants of the transformation require lower levels than the maximum.

Results against the raw changes

Figure 5.2 shows the precision and recall obtained by both the semantic patches written by the human expert and the semantic patches inferred by SPINFER for the synthesis experiment. Precision and recall of the changes produced by the semantic patches are evaluated against the raw changes in the random dataset.

We first focus on the human results. We can notice that some results are missing, indicating that no semantic patch was written for the transformations. Indeed it is not always possible to write a semantic patch for a transformation. For instance, the transformation at index 1 involves a lot of string manipulations. While it is possible to write a semantic patch using very advanced SmPL syntax for this transformation, we do not expect kernel developers to write such semantic patches. Transformations

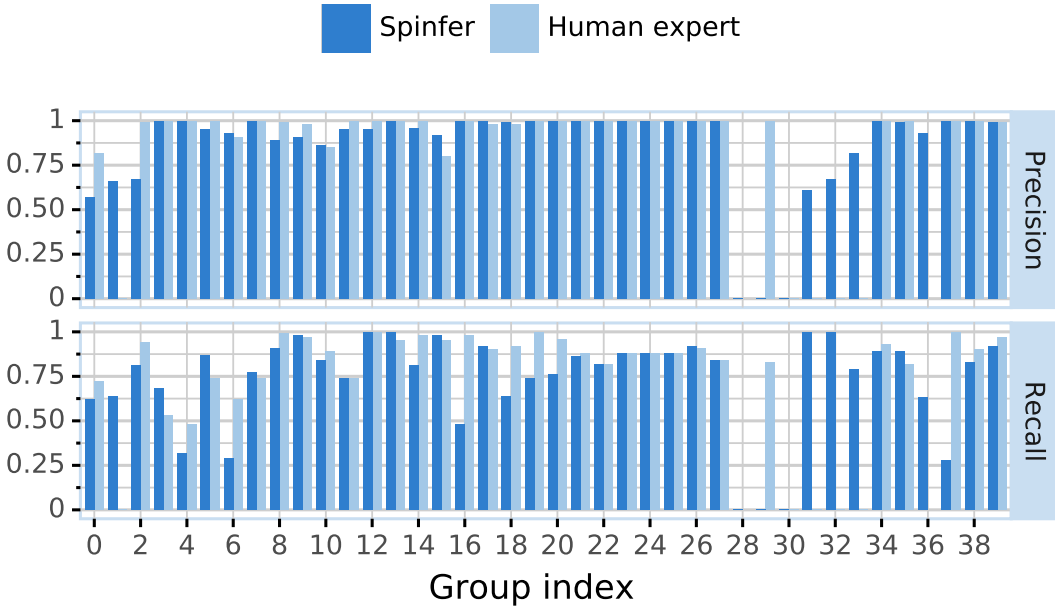


Fig. 5.2: Human vs. SPINFER semantic patches in the synthesis experiment

at indices 28, 30 to 33 and 36 are either interprocedural or rely on very specific conditions that cannot be translated into a semantic patch. When it is possible to write the semantic patch, it does not always achieve 100% precision and recall on all examples. Across all of our examples our human expert achieved in average 98% precision and 86% recall. Obtaining less than 100% precision and recall is perfectly normal since these examples contain noise, such as specific changes that are not useful or possible to automate.

If we compare the SPINFER results to the human expert results, we can see that the results are close to human ones for most transformations. We can note that SPINFER does infer some semantic patches even when the transformation cannot be fully captured with one, and sometimes infers more changes (as indicated by higher recall) than the human semantic patch. This indicates that SPINFER can infer some undesirable changes that artificially improve recall. SPINFER obtains 92% precision and 79% recall which is close to human semantic patch results.

We saw here, since Linux kernel transformations contain a lot of noise, it is not possible or useful even for a human to produce a semantic patch covering all changes.

Results against human expert semantic patch changes

In the previous evaluation we were looking at recall and precision against all changes in the dataset. Since we are more interested in inferring only the part of API-usage updates that it is reasonable to automate, we now compare the results of SPINFER semantic patches against the results produced by human semantic patches. Figure 5.3 (page 91) illustrates this comparison on the random dataset for both the synthesis and the transformation experiments.[2] In this case we separated the transformations out of the scope of SPINFER from the transformations that SPINFER can theoretically handle in terms of the taxonomy.

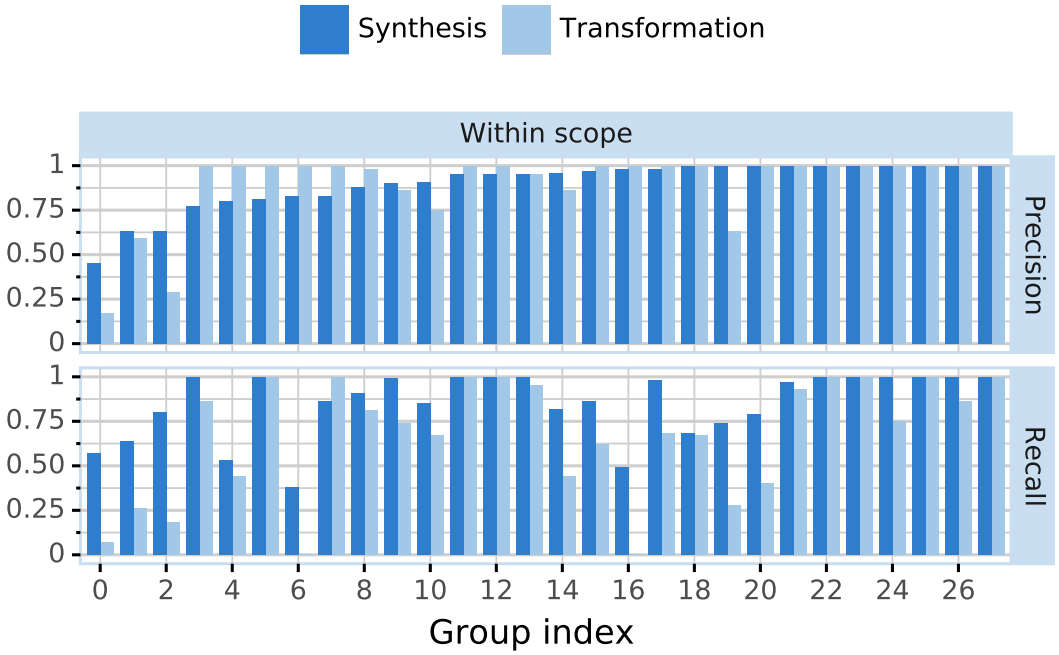
For both the synthesis and transformation experiments, SPINFER obtains better results for transformations that fit in its scope in terms of the taxonomy. In this dataset, for the synthesis experiments, SPINFER obtained on average 90% precision and 85% recall for examples in its scope and 87% precision and 83% for examples outside its scope. For the transformation experiment, SPINFER obtains 90% precision and 66% recall for examples in its scope and only 80% precision and 66% for examples outside its scope.

Indeed, transformations requiring control-flow dependencies to unmodified terms cannot be correctly inferred by our approach. In general, this will cause the inferred transformation to have fewer constraints on its application and change more terms than needed, resulting in a lower precision. Recall is mostly unaffected because this metric does not consider incorrectly modified nodes. However transformations requiring control-flow dependencies to unmodified terms or more complex relations represent only a minority of widespread transformations in the Linux kernel, provided that our dataset is representative of the transformation distribution in Linux.

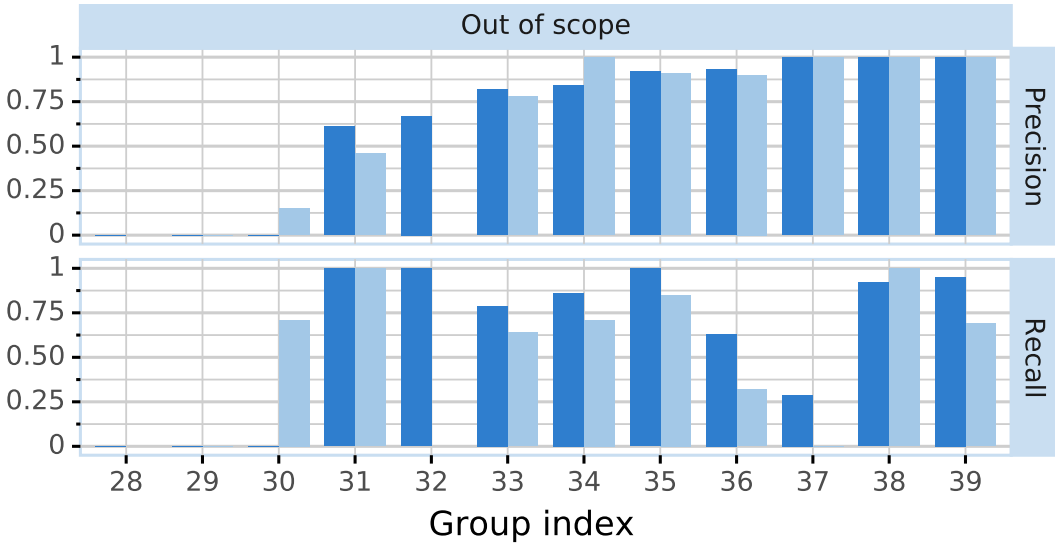
Answering RQ1: Is SPINFER successful at learning Linux kernel API usage updates?

We evaluated SPINFER's performance for learning Linux kernel transformations and observed that for both experiments SPINFER obtains very high precision and high recall compared to the results produced by a human-expert written semantic patch. SPINFER obtains worse results for transformations requiring complex-control flow dependencies but this kind of transformation is uncommon in the Linux kernel.

[2] For transformation groups where the human expert could not write a semantic patch we compare the generated results against the raw changes.



(a) Examples in SPINFER's scope



(b) Examples out of SPINFER's scope

Fig. 5.3: SPINFER changes evaluated against human changes on the *random* dataset

This suggests that SPINFER is adapted for learning Linux kernel changes for either synthesis or transformation tasks.

5.2.2 RQ2: Efficiency

We next analyze whether using SPINFER could be efficient for Linux kernel developers. With our approach we target the following workflow: a Linux kernel developer finds and writes a small number of examples for a transformation they wish to infer. The developer then runs SPINFER on their set of examples and obtains a semantic patch. Next the developer carefully reads the produced semantic patch and fixes it if necessary. Once the developer is satisfied with their semantic patch, they apply it to the Linux kernel. This workflow could be considered efficient if it is faster than writing a correct semantic patch by hand.

Execution time

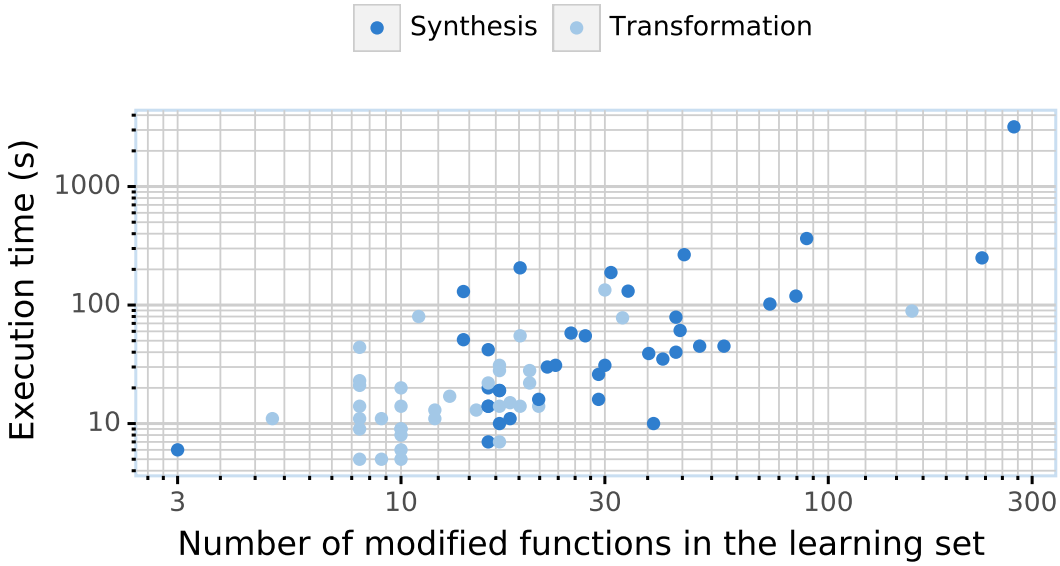


Fig. 5.4: Execution time to infer semantic patches

Figure 5.4 shows the time required for SPINFER to infer a semantic patch on the random dataset, depending on the number of modified functions in the learning set, with each axis being in logarithmic scale. This experiment was performed on a server equipped with a bi-processor Intel Xeon E5-2699 v4, with 44 threads per processor, and a maximum frequency of 3.6GHz. However SPINFER users does not require such processing power since our implementation is single-threaded.

We can see that the time required to infer a semantic patch is roughly linear with the number of modified functions. Up to 100 modified functions, the running time does not exceed 300 seconds, except for only one transformation.

Most users would get a semantic patch in under 5 minutes, which is lower than the time required to read and understand a hundred modified functions. Moreover this experiment was performed on a server CPU, but modern laptop or desktop CPUs are more optimized than server CPUs for single-threaded tasks. By way of comparison, our inferring a semantic patch for our `timer` transformation took 104 seconds on the server CPU and only 77 seconds on a laptop CPU.

Helping the user

After the semantic patch is produced, the user of SPINFER needs to review it to ensure that it is correct. If the semantic patch is actually incorrect that poses two problems: the user needs to detect that the semantic patch is incorrect and then needs to determine how to fix it.

To help the user in this task we augment the semantic patch with some useful information in comments. For each semantic patch rule SPINFER does an evaluation of the rule on the learning set and indicates the following information:

- The precision and recall of the rule in isolation evaluated against all changes in the learning set.
- The recall of the rule in isolation but restricted to functions the rule modifies.
- The functions in the learning set from which the rule is inferred.

At the end of the patch SPINFER indicates information about the complete semantic patch:[3]

- The precision and recall of the semantic patch.
- The number of functions that were completely and correctly changed (100% recall and precision) and the total number of functions in the learning set.
- The functions where the semantic patch produced incorrect results.
- The functions where the semantic patch did not apply.

[3] Since we found that this information can be useful to non-SPINFER users, it is possible to build a binary, called `eval_patch`, printing just this information for any semantic patch, using SPINFER source repository.

- The functions where the semantic patch applied but did not perform all the changes.

All this information should help the user assess if the semantic patch is ready to be applied and to fix it if it is incorrect or incomplete. For instance, by looking at functions where the semantic patch did not perform all the changes, the user sees terms considered as noise. Then the user can decide if the noise represents unrelated changes, and in that case keep the semantic patch intact, or if the noise represents terms not worth automating, and in that case they know what terms to fix manually, or again if the noise is incorrect and should have been part of the transformation, and in that case complete the semantic patch.

Answering RQ2: Can using SPINFER be efficient for Linux kernel developers?

We have seen that our approach is able to infer semantic patches quickly compared to the time required for developers to read the changes. Moreover we provide ways for developers to quickly identify incomplete or incorrect patches and fix them.

These results in conjunction with ones we obtained in previous sections, suggest that SPINFER can be used by Linux kernel developers to speed up large-scale API-usage updates.

5.2.3 RQ3: Shortcomings

We saw that, even if SPINFER obtains good results, it does not obtain 100% recall and precision in all examples that fit its scope. To understand these results we will look the shortcomings of our approach and its implementation SPINFER. For this we construct a new dataset called the *challenging* dataset.

The *challenging* dataset

This dataset consist in 40 groups of API-usage update in the Linux kernel obtained using two methods:

- Changes produced by a semantic patch using Coccinelle. The semantic patch is referenced in the commit message for the change.

- Changes coming from recurring changes found in the Linux kernel versions v4.16-v4.19, that have been identified using the tool *patchparse* [PLM06].

It is important to note that, this dataset is called the *challenging* dataset because its examples were selected to test the limits of our approach. Consequently it is not representative, in terms of the taxonomy, of the real distribution of changes in the Linux kernel.

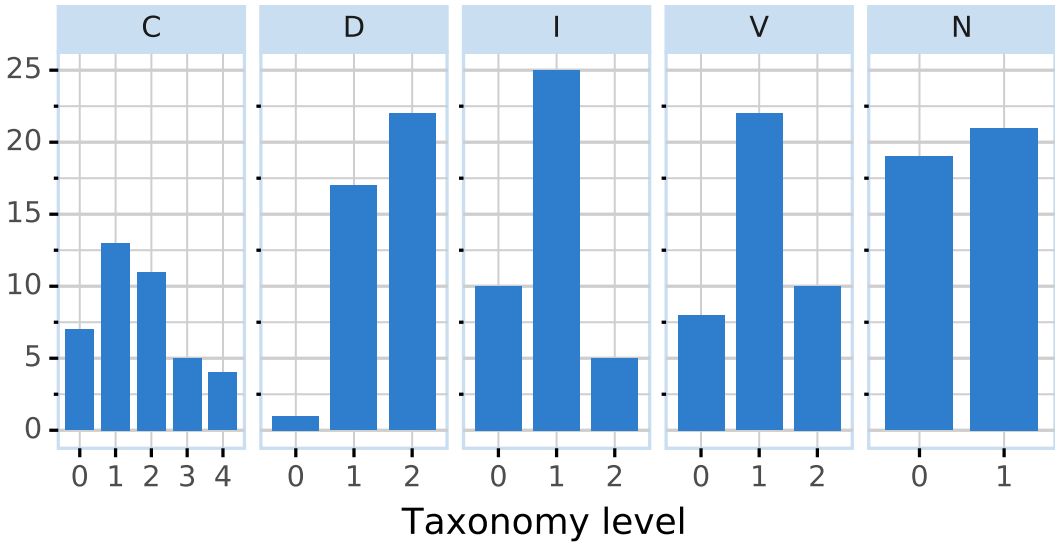
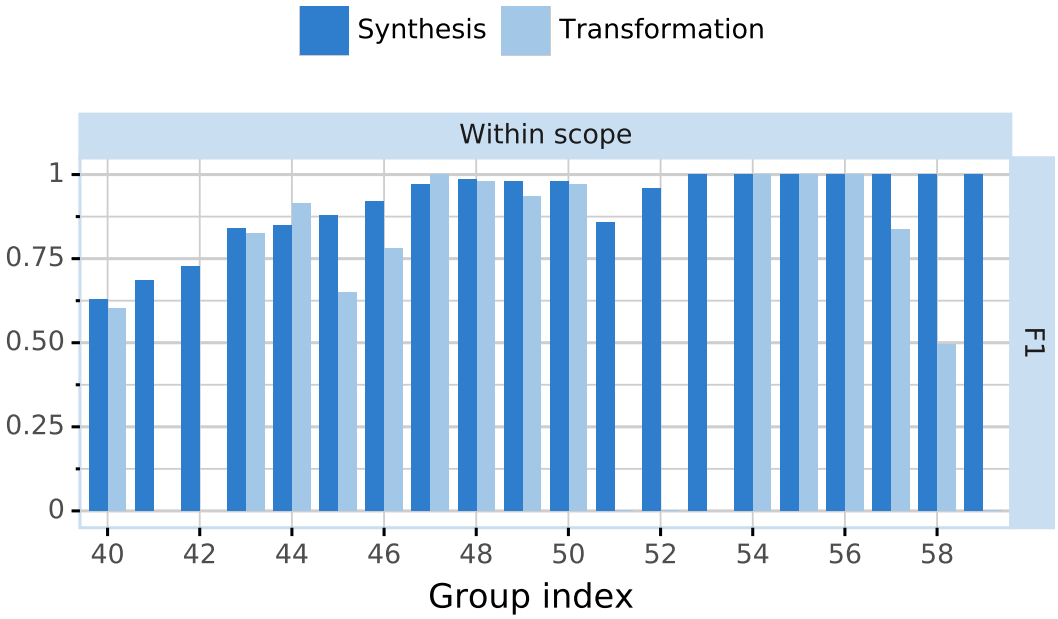


Fig. 5.5: Distribution in terms of the taxonomy for the challenging dataset

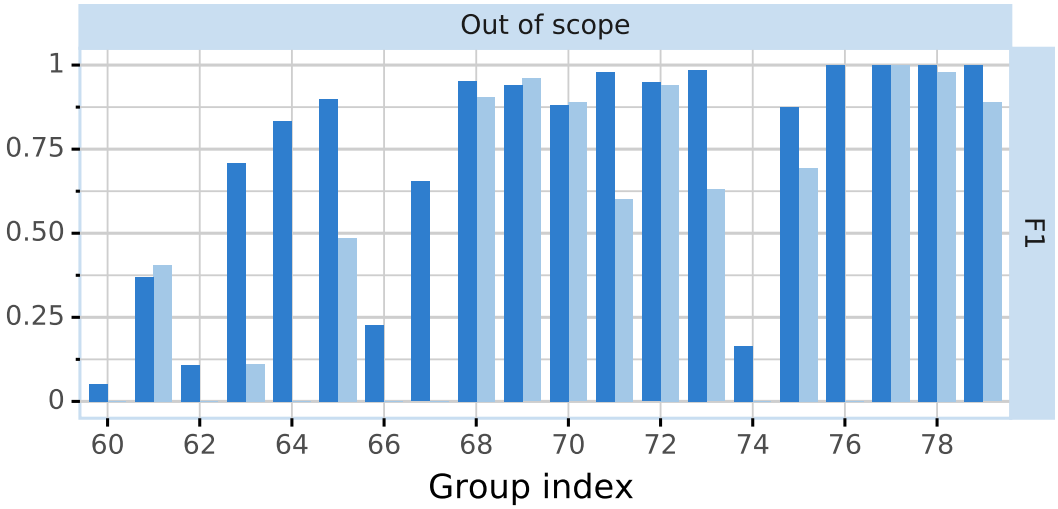
Figure 5.5 shows the taxonomy distribution of each group of changes for this dataset. We can see that most transformations (21 out of 40) require at least taking into account control-flow relationships between changed and unchanged terms (C_{2+}). This is the challenging part for our approach because SPINFER only targets control-flow relationships between changed terms only, or less (C_1). The majority of transformations also rely on data-flow relationships between changed and unchanged terms (D_2) and most transformations contain at least one example with multiple instances (I_{1+}). It is worth noting that variants play a major role in this dataset, with the majority of transformations having multiple variants (V_{1+}) and 10 transformations requiring variants to be applied in a specific order (V_2). Noise is common in this dataset, with 21 transformations containing at least one example with noise (N_1).

Figure 5.6 shows the F_1-score of SPINFER's inferred semantic patches when evaluated against the results produced by the human experts' semantic patches.[4]

[4]There was not enough files to perform the transformation experiment for indices 41, 42, 52 and 53.



(a) Examples in SPINFER's scope



(b) Examples out of SPINFER's scope

Fig. 5.6: F1-score of SPINFER's patches against human changes in the challenging dataset

First we compare the F_1-scores between transformation and synthesis. We can see that, even in SPINFER's scope, the F_1-score can be much lower in the transformation experiment. To understand why it is the case we will look at some of the inferred semantic patches. We start with the generated semantic patch for our `timer` example we saw earlier.

Analysis of the `timer` migration

Listing 5.1 shows the inferred semantic patch in the synthesis experiment, which has the index 57 in our figures. In this experiment SPINFER was learning from 31 files containing 33 modified functions.

```
@@ expression E0, E1, E2; @@
- E0.function = E1;
- E0.data = E2;
  ...
- init_timer(&E0);
+ setup_timer(&E0, E1, E2);

@@ expression E0, E1, E2; @@
- init_timer(&E0);
+ setup_timer(&E0, E1, E2);
  ...
- E0.data = E2;
- E0.function = E1;

@@ expression E0, E1, E2; @@
- init_timer(&E0);
- E0.function = E1;
- E0.data = E2;
+ setup_timer(&E0, E1, E2);

@@ expression E0, E1; @@
- init_timer(&E0);
- E0.function = E1;
+ setup_timer(&E0, E1, 0UL);
```

Listing 5.1: Inferred semantic patch for the timer change in the synthesis experiment

The inferred semantic patch achieves 100% recall and precision compared to the changes produced by the human expert semantic patch. Still, the inferred semantic patch is not perfect and could be improved. If we compare this semantic patch and the one given in our motivating example in Section 4.2.1 on page 43 we can see that the inferred patch is missing a sequence operator between the `init_timer` call

and the `function` field initialization for rules 3 and 4. This happens because in every example for these two variants the `function` field initialization is always right after the `init_timer` call with no unmodified statements in between. Our approach only generates distant control-flow relationships if they exist in the learning set.

```
@@
expression E0, E2;
identifier I1;
@@
- init_timer(&E0);
+ setup_timer(&E0, I1, E2);
  ...
- E0.data = E2;
- E0.function = I1;

@@
expression E0, E2;
identifier I1;
@@
- init_timer(&E0);
- E0.function = I1;
- E0.data = E2;
+ setup_timer(&E0, I1, E2);

@@
expression E0, E2;
identifier I1;
@@
- E0.function = I1;
- E0.data = E2;
- init_timer(&E0);
+ setup_timer(&E0, I1, E2);
```

Listing 5.2: Inferred semantic patch for the timer change in the transformation experiment

This effect is more pronounced in Listing 5.2 which shows the obtained semantic patch in the transformation experiment by learning from only 10 files. This patch obtains only 72% recall on the validation set (with 100% precision). This lower recall is explained by two phenomena:

- *Variant bias:* examples in the learning sets do not contain all possible variants of the transformation.
- *Overspecialization:* some terms are more specialized than they need to be due to lack of term diversity in the learning set.

For this semantic patch, we have only three out of four variants and all `function` field initialization values are overspecialized as identifiers. This behavior happens because examples in the learning contain neither the fourth variant nor a `function` field initialization with an expression value.

Since these issues arise because examples in the learning set are not representative of all possibilities of the transformation, we call this category of problems *learning bias*.

Focus on *overspecialization*

Sometimes, learning bias can have dramatic results on the quality of the generated patches, even on very simple transformations. Let us illustrate this problem by looking at another example of a transformation, which has index 59 in our figures. Listing 5.3 shows the human expert's semantic patch for this transformation:

```
@@
expression sock, mem, size;
@@
- memzero_explicit(mem, size);
- sock_kfree_s(sock, mem, size);
+ sock_kzfree_s(sock, mem, size);
```

Listing 5.3: Human semantic patch for the `sock_kzfree_s` transformation

In this transformation, the first removed call fills a memory region with zeros and the second removed call frees a memory region within a socket interface. This combination of functions ensures that deallocated memory from the socket interface will not leak sensitive data. These two functions calls are replaced by a new function call that does the both the freeing and the zeroing of memory at once.

However when learning from examples of this transformation SPINFER infers the following semantic patch:

```
@@
identifier I1;
identifier I2 = {crypto_ablkcipher_ivsize, crypto_ahash_digestsize};
expression E3, E0, E4;
@@
- memzero_explicit(E0->I1, I2(E3));
- sock_kfree_s(E4, E0->I1, I2(E3));
+ sock_kzfree_s(E4, E0->I1, I2(E3));
```

Listing 5.4: SPINFER semantic patch for the `sock_kzfree_s` transformation

Because the examples are very similar (both used in crytographic context), terms in the semantic patch inferred by SPINFER are more specialized than needed. As a consequence this semantic patch will not likely be applicable to examples outside the learning set.

Unsupported SmPL syntax

Some failures of SPINFER can also be explained by missing support for advanced SmPL syntax. For instance, our human expert semantic patches uses the <... ...> and <+... ...+> sequence operators seen in Section 2.2.2 on page 11. These two sequence operators were needed in 10 transformations for the random dataset and in 8 transformations for the challenging dataset. Our approach has no way of generating such operators and so it will not generate correct semantic patches in the transformation where they are needed, resulting in lower recall for the affected transformations, even if they fit in SPINFER's scope in terms of the taxonomy.

Answering RQ3: What are the shortcomings of our approach?

In this section we highlighted the fact that our approach suffers from *learning bias*, with two main phenomena: *variant bias* and *overspecialization*. Learning bias happens when examples in the learning are not representative of all possibilities of a transformation. However this issue is not only present in our approach but in all approaches that learns transformations from examples. Moreover, our approach only generates semantic patches using a subset of the SmPL syntax, preventing SPINFER from generating correct semantic patches for some transformations.

5.3 Discussion

In this section we will look at the threats of validity for our approach and at future directions to evaluate our approach.

5.3.1 Threats to validity

External validity

Our evaluation has two main threats to external validity, which may prevent our findings to be generalized to other contexts:

- We evaluated SPINFER on a group of 40 randomly picked transformations from large-scale changes in 2018. As the sample size is small, it may not be representative of all Linux kernel transformations.
- Some results were evaluated against changes produced by human semantic patches. It is possible that our human expert and other kernel developers do not agree on what changes are useful to automate, and should be part of the patch, and what changes should be excluded.

Construct validity

Our efficiency analysis uses mainly SPINFER inference time to determine the time it would take for a developer to obtain a correct patch using our approach. The time required to find, or write, examples of transformations, the readability of the inferred patch and the number of fixes to perform in the inferred semantic patch also factor in for time required to obtain a correct patch. However time added by these factors is harder to quantify.

Internal validity

As our implementation consists of thousands of lines of code, it is possible that it contains bugs. Bugs could affect the behavior of our previously described algorithms and change the results produced in unexpected ways.

5.3.2 Conclusion and future work

In this chapter we evaluated our approach for inferring API-usage updates on 80 real Linux kernel transformations. We collected a variety of metrics such as precision, recall and execution time, and analyzed in detail some inferred semantic patches. This enabled us to understand when our approach could give good results, and whether these results could be useful to kernel developers.

We found out that our approach worked best for transformations requiring no control-flow dependencies on unmodified terms and that semantic patches could be inferred in a reasonable time. SPINFER's results could then be improved by using a strategy to identify unmodified terms necessary for these control-flow dependencies. We also need to support more SmPL syntax in order to widen the range of inferable transformations.

To evaluate SPINFER further we need to evaluate the quality of the produced patches. This is rather difficult because the quality of a semantic patch does not depend only on the results produced but also on its readability. As with traditional source code, the readability of a semantic patch is affected by the personal preferences of developers.

We also need to compare our approach with other approaches in the domain. Once again this is not an easy task. Firstly, because other tools and their example sets are not always available or maintained. Secondly, because the large majority of tools targets Java code and often more precisely Android API-usage updates. Porting examples from C to Java, or inversely from Java to C, could introduce a bias in the evaluation. Application software like Android apps and infrastructure software like the Linux kernel often use very different programming paradigms and we expect tools to be optimized for some paradigms at the expense of others. However it should be possible to indirectly compare results by looking at the average metrics obtained for each taxonomy level. Obtaining accurate metrics for this kind of comparison will require a lot of examples for each taxonomy level and to be sure there is no correlation between taxonomy level in the different taxonomy categories. However, we hope that future literature in the domain will reuse our taxonomy to make comparison possible.

6 Conclusion

As the Linux kernel continues to grow in size, maintenance issues arise. Replacing APIs for performance, security or readability reasons can become insurmountable as a single API usage change can potentially affect thousands of drivers that must be kept functional. In these conditions developers need automation tools to help them in their maintenance tasks. This thesis analyzes the challenges of API usage updates and proposes a solution to automatically perform API migrations in the Linux kernel, by generating transformation rules learned from a set of migration examples. This proposed approach is capable of learning transformations composed of multiple variants and transformations that rely on control-flow relationships. It uses code clustering to find the most appropriate abstractions to use in its generated transformation rules. This approach is also capable of handling imperfect examples in its learning set and can order generated transformation rules to maximize the inferred transformation coverage. This approach generates transformation rules in a form of semantic patches, a transformation language already known by Linux kernel developers, which makes it suitable to be used by kernel maintainers.

We evaluated how well SPINFER, the implementation of our approach, could learn real Linux kernel API updates. We found out that, given few examples of an API migration, SPINFER learned semantic patches performing the remaining changes with an average of 92% precision and 68% recall compared to changes that can be produced by human-written semantic patches. Except in rare cases, the inference process took only few minutes, suggesting that SPINFER can boost maintainers efficiency at performing large-scale changes.

6.1 Future work

With our approach we have shown that it is possible to infer transformation rules suitable for the Linux kernel, by clustering similar changed codes to find abstractions, and then assembling these abstractions using control-flow dependencies.

However, even if our results are encouraging there is still room for improvement. First we have to target the two remaining levels in the taxonomy, that our approach

cannot handle: control-flow dependencies to unmodified terms and negative control-flow dependencies. The former can probably be solved by restricting the number of unmodified terms to look for using the same techniques as other approaches, *e.g.* through control or data-flow dependencies to modified terms. The latter is more difficult to address, as it needs an in-depth comprehension of the transformation. We also need to extend the range of SmPL syntax our approach can generate, with, for instance, more kinds of SmPL sequence operators.

Other kinds of improvements have to be done at the user-experience level. Quickly finding relevant examples to add to the learning set is one way to improve user experience. In our evaluation we used patchparse to find systematic edits[PLM06]. Other approaches have been developed to find examples by querying commit histories and analyzing compilation errors [Law+17]. Still, having more automatic ways to find examples will be beneficial to users. User experience is also improved by the quality of the inferred semantic patches. This depends on the precision and recall obtained by semantic patches but also on the readability of the semantic patches. To quantify readability we could, for instance, look at the number of generated rules compared to the number of changes to make, or the size of each rule. Another way to improve readability is to find meaningful names for metavariables, as it is usually the case for human-written semantic patches. To find these names we could, for instance, look at the concrete terms matched by the metavariables.

6.2 Perspectives

We also need to take a step back and look at the overall picture of software maintenance. Software maintainers do not migrate API usages for fun, they migrate them because they are forced to. It is often because the environment providing these APIs must change, for security reasons or because it becomes deprecated. In that case, developers may need to update several API usages at the same time between major API versions, which is very different from the small, individual and incremental upgrades of API usage that are targeted by approaches we described at the start of this thesis.

Migrating multiple APIs at once adds complexity, because the different API usage mixes up and automatic tools needs to recognize each individual API migration. Environmental changes also means that some API can disappear, and in that case we also need to guide users to alternative APIs providing the set of functionality.

Future approaches in the domain also need to exploit the full set of information available to software maintainers: not only examples of API usage migration but also documentation, release notes and information available on specialized question and answer sites. Current research suggests that information coming from documentation could contain a substantial amount of information to perform API migrations [LS18].

Bibliography

[All70] Frances E Allen. “Control flow analysis”. In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19 (cit. on pp. 31, 48).

[AL10] Jesper Andersen and Julia L. Lawall. “Generic patch inference”. en. In: *Automated Software Engineering* 17.2 (June 2010). Spdiff, pp. 119–148 (cit. on p. 37).

[And+12] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. “Semantic patch inference”. en. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. Spdiff. Essen, Germany: ACM Press, 2012, p. 382 (cit. on pp. 30, 37).

[Asa+13] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Massimiliano Di Penta. “Lhdiff: A language-independent hybrid approach for tracking source code lines”. In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 230–239 (cit. on p. 29).

[Bru+09] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. “A foundation for flow-based program matching: using temporal logic and model checking”. In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2009, pp. 114–126 (cit. on p. 8).

[BM09] Peter Bulychev and Marius Minea. “An evaluation of duplicate code detection using anti-unification”. In: *Proc. 3rd International Workshop on Software Clones*. Citeseer. 2009 (cit. on pp. 54, 55).

[BM08] Peter Bulychev and Marius Minea. “Duplicate code detection using anti-unification”. In: *Proceedings of the Spring/Summer Young Researchers’ Colloquium on Software Engineering*. 2. 2008 (cit. on p. 54).

[CCDP09] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. “Ldiff: An enhanced line differencing tool”. In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 595–598 (cit. on p. 29).

[CCDP08] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. “Tracking your changes: A language-independent approach”. In: *IEEE software* 26.1 (2008), pp. 50–57 (cit. on p. 29).

[CLZ14] Kai Chen, Peng Liu, and Yingjun Zhang. “Achieving accuracy and scalability simultaneously in detecting application clones on android markets”. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 175–186 (cit. on p. 31).

[Coo] Kees Cook. *treewide: init_timer() -> setup_timer()*. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b9eaf1872222`. [Online; accessed September-2020] (cit. on pp. 6, 55).

[CHK01] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. "A simple, fast dominance algorithm". In: *Software Practice & Experience* 4.1-10 (2001), pp. 1–8 (cit. on p. 62).

[Dot+17] Georg Dotzler, Marius Kamp, Patrick Kreutzer, and Michael Philippsen. "More accurate recommendations for method-level changes". en. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ARES. Paderborn, Germany: ACM Press, 2017, pp. 798–808 (cit. on p. 37).

[DP16] Georg Dotzler and Michael Philippsen. "Move-optimized source code tree differencing". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 660–671 (cit. on pp. 30, 37).

[Est+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Kdd*. Vol. 96. 34. 1996, pp. 226–231 (cit. on p. 36).

[Fdr] *F-Droid - Free and Open Source Android App Repository*. `https://f-droid.org/`. [Online; accessed October-2020] (cit. on p. 29).

[Fal+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. "Fine-grained and accurate source code differencing". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. Ed. by Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher. ACM, 2014, pp. 313–324 (cit. on pp. 30, 51).

[FXO19] Mattia Fazzini, Qi Xin, and Alessandro Orso. "Automated API-usage update for Android apps". en. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2019*. AppEvolve. Beijing, China: ACM Press, 2019, pp. 204–215 (cit. on pp. 5, 29, 30, 33).

[Flu+07] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction". en. In: *IEEE Transactions on Software Engineering* 33.11 (Nov. 2007), pp. 725–743 (cit. on pp. 30, 32).

[Git] *Git user manual glossary: Commit*. `https://git-scm.com/docs/user-manual#def_commit`. [Online; accessed September-2020] (cit. on p. 28).

[HM08] Masatomo Hashimoto and Akira Mori. "Diff/TS: A tool for fine-grained structural change analysis". In: *2008 15th working conference on reverse engineering*. IEEE. 2008, pp. 279–288 (cit. on p. 30).

[HS15] Mohammad Hossin and MN Sulaiman. "A review on evaluation metrics for data classification evaluations". In: *International Journal of Data Mining & Knowledge Management Process* 5.2 (2015), p. 1 (cit. on p. 78).

[Jia+19] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. “Inferring Program Transformations From Singular Examples via Big Code”. en. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. GenPat. San Diego, CA, USA: IEEE, Nov. 2019, pp. 255–266 (cit. on p. 35).

[Jia+18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. “Shaping program repair space with existing patches and similar code”. In: *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 2018, pp. 298–309 (cit. on p. 39).

[Jia+07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. “Deckard: Scalable and accurate tree-based detection of code clones”. In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 96–105 (cit. on p. 31).

[Joh94] J Howard Johnson. “Substring Matching for Clone Detection and Change Tracking”. In: *ICSM*. Vol. 94. 1994, pp. 120–126 (cit. on p. 31).

[Kah62] Arthur B Kahn. “Topological sorting of large networks”. In: *Communications of the ACM* 5.11 (1962), pp. 558–562 (cit. on p. 81).

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. “CCFinder: a multilinguistic token-based code clone detection system for large scale source code”. In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670 (cit. on pp. 31, 35).

[Kre+16] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael Philippsen. “Automatic clustering of code changes”. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2016, pp. 61–72 (cit. on p. 28).

[Kri01] Jens Krinke. “Identifying similar code with program dependence graphs”. In: *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE. 2001, pp. 301–309 (cit. on p. 31).

[LS18] Maxime Lamothe and Weiyi Shang. “Exploring the use of automated API migrating techniques in practice: an experience report on Android”. en. In: *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. Gothenburg, Sweden: ACM Press, 2018, pp. 503–514 (cit. on p. 105).

[LSC18] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. “A4: Automatically Assisting Android API Migrations Using Code Examples”. en. In: *arXiv:1812.04894 [cs]* (Dec. 2018). A4 (cit. on pp. 5, 28, 30, 34).

[LM18] Julia Lawall and Gilles Muller. “Coccinelle: 10 years of automated evolution in the Linux kernel”. In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 601–614 (cit. on p. 1).

[Law+17] Julia Lawall, Derek Palinski, Lukas Gnirke, and Gilles Muller. “Fast and precise retrieval of forward and back porting information for linux device drivers”. In: *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 2017, pp. 15–26 (cit. on pp. 28, 104).

[LLLG16] Xuan Bach D Le, David Lo, and Claire Le Goues. “History driven program repair”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 213–224 (cit. on p. 39).

[LT79] Thomas Lengauer and Robert Endre Tarjan. “A fast algorithm for finding dominators in a flowgraph”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141 (cit. on p. 74).

[Li+18] Li Li, Jun Gao, Tegawendé F Bissyandé, et al. “Characterising deprecated android apis”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 254–264 (cit. on p. 6).

[LR16] Fan Long and Martin Rinard. “Automatic patch generation by learning correct code”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 298–312 (cit. on p. 39).

[LM69] Edward S Lowry and Cleburne W Medlock. “Object code optimization”. In: *Communications of the ACM* 12.1 (1969), pp. 13–22 (cit. on p. 48).

[Ma+17] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. “Vurle: Automatic vulnerability detection and repair by learning from examples”. In: *European Symposium on Research in Computer Security*. Springer. 2017, pp. 229–246 (cit. on p. 38).

[MES03] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With GNU Diff and Patch*. Network Theory Ltd, Jan. 2003 (cit. on pp. 1, 6, 29).

[MM19] Matias Martinez and Martin Monperrus. “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)*. IEEE. 2019, pp. 79–82 (cit. on p. 28).

[MRK13] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. “An empirical study of api stability and adoption in the android ecosystem”. In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 70–79 (cit. on p. 6).

[Men+15] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. “Does automated refactoring obviate systematic editing?” In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. RASE. IEEE, 2015, pp. 392–402 (cit. on p. 38).

[MKM13] Na Meng, Miryung Kim, and Kathryn S. McKinley. “Lase: Locating and applying systematic edits by learning from examples”. en. In: *2013 35th International Conference on Software Engineering (ICSE)*. LASE. San Francisco, CA, USA: IEEE, May 2013, pp. 502–511 (cit. on pp. 30, 35).

[MKM11] Na Meng, Miryung Kim, and Kathryn S McKinley. “Systematic editing: generating program transformations from an example”. In: *ACM SIGPLAN Notices* 46.6 (2011). SYDIT, pp. 329–342 (cit. on pp. 5, 30–32).

[MSDR17] Tim Molderez, Reinout Stevens, and Coen De Roover. “Mining change histories for unknown systematic edits”. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 248–256 (cit. on p. 28).

[Mon18] Martin Monperrus. “Automatic Software Repair: A Bibliography”. en. In: *ACM Computing Surveys* 51.1 (Apr. 2018), pp. 1–24 (cit. on p. 39).

[Ngu+13] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. “A study of repetitiveness of code changes in software evolution”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 180–190 (cit. on p. 6).

[Ngu+10] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, et al. “A graph-based approach to API usage adaptation”. In: *ACM Sigplan Notices* 45.10 (2010), pp. 302–321 (cit. on p. 38).

[Pad+08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. “Documenting and automating collateral evolutions in Linux device drivers”. In: *Acm sigops operating systems review* 42.4 (2008), pp. 247–260 (cit. on p. 8).

[PLM07] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. “SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers”. en. In: *Electronic Notes in Theoretical Computer Science* 166 (Jan. 2007). SmPL, pp. 47–62 (cit. on pp. 5, 7).

[PLM06] 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*. 2006, pp. 59–71 (cit. on pp. 28, 87, 95, 104).

[PA11] Mateusz Pawlik and Nikolaus Augsten. “RTED: A Robust Algorithm for the Tree Edit Distance”. In: *Proceedings of the VLDB Endowment* 5.4 (2011) (cit. on p. 30).

[Plo70] Gordon D Plotkin. “A note on inductive generalization”. In: *Machine intelligence* 5.1 (1970), pp. 153–163 (cit. on p. 54).

[RU11] Anand Rajaraman and Jeffrey David Ullman. “Data Mining”. In: *Mining of Massive Datasets*. Cambridge University Press, 2011, 7–9 (cit. on p. 56).

[Ray+15] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. “The uniqueness of changes: Characteristics and applications”. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, pp. 34–44 (cit. on p. 6).

[Rol+17] Reudismam Rolim, Gustavo Soares, Loris D’Antoni, et al. “Learning Syntactic Program Transformations from Examples”. en. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. REFAZER. Buenos Aires: IEEE, May 2017, pp. 404–415 (cit. on pp. 30, 36).

[Rol+18] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D’Antoni. “Learning Quick Fixes from Code Repositories”. en. In: *arXiv:1803.03806 [cs]* (Sept. 2018). REVISAR (cit. on p. 38).

[Rou87] Peter J Rousseeuw. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of computational and applied mathematics* 20 (1987), pp. 53–65 (cit. on p. 59).

[SW11] Claude Sammut and Geoffrey I Webb. *Encyclopedia of machine learning*. Springer Science & Business Media, 2011 (cit. on p. 79).

[Ser+20] Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, et al. “SPINFER: Inferring Semantic Patches for the Linux Kernel”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 235–248 (cit. on p. 41).

[SK16] Abdullah Sheneamer and Jugal Kalita. “A survey of software clone detection techniques”. In: *International Journal of Computer Applications* 137.10 (2016), pp. 1–21 (cit. on p. 31).

[Sub] *Submitting patches: the essential guide to getting your code into the kernel*. `https://www.kernel.org/doc/html/v5.4/process/submitting-patches.html` (cit. on p. 42).

[Smp] *The SmPL Grammar (version 1.0.8)*. `https://coccinelle.gitlabpages.inria.fr/website/docs/main_grammar.pdf`. [Online; accessed September-2020] (cit. on p. 11).

[Thu+20] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, et al. “Automated Deprecated-API Usage Update for Android Apps: How Far are We?” In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 602–611 (cit. on p. 33).

[Thu+16] Ferdian Thung, Xuan-Bach D Le, David Lo, and Julia Lawall. “Recommending code changes for automatic backporting of linux device drivers”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 222–232 (cit. on p. 37).

[Whi+16] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. “Deep learning code fragments for code clone detection”. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 87–98 (cit. on p. 31).

[XDM19] Shengzhe Xu, Ziqi Dong, and Na Meng. “Meditor: Inference and Application of API Migration Edits”. en. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. Meditor. Montreal, QC, Canada: IEEE, May 2019, pp. 335–346 (cit. on pp. 28, 30, 34).

[YG12] Yang Yuan and Yao Guo. “Boreas: an accurate and scalable token-based approach to code clone detection”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 286–289 (cit. on p. 31).

[ZH02] Mohammed Javeed Zaki and Ching-Jiu Hsiao. “CHARM: An Efficient Algorithm for Closed Itemset Mining”. In: *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*. SIAM, 2002, pp. 457–473 (cit. on p. 28).

[ZS89] Kaizhong Zhang and Dennis Shasha. “Simple fast algorithms for the editing distance between trees and related problems”. In: *SIAM journal on computing* 18.6 (1989), pp. 1245–1262 (cit. on pp. 30, 36).

[ZK02] Ying Zhao and George Karypis. “Evaluation of hierarchical clustering algorithms for document datasets”. In: *Proceedings of the eleventh international conference on Information and knowledge management*. 2002, pp. 515–524 (cit. on p. 58).

List of Figures

List of Tables

List of Listings