SEMANTIC PATCH INFERENCE

Andersen, Jesper

Publication date:
2009

Document version
Publisher's PDF, also known as Version of record

SEMANTIC PATCH INFERENCE

JESPER ANDERSEN

Computer Science Department (DIKU)

The Graduate School of Science
Faculty of Science
University of Copenhagen

Copenhagen
November 2009

Supervisor: Julia L. Lawall

[November 13, 2009 at 10:43]

Dedicated to my loving wife and son. You are the sunshine that light up my day.

ABSTRACT

Collateral evolution the problem of updating several library-using programs in response to API changes in the used library. In this dissertation we address the issue of understanding collateral evolutions by automatically inferring a high-level specification of the changes evident in a given set of updated programs.

We have formalized a concept of *transformation parts* that serve as an indication of when a change specification is evident in a set of changes. Based on the transformation parts concept, we state a *subsumption relation* on change specifications. The subsumption relation allows decision of when a change specification captures a maximal amount of the evident changes in a set of changes. We state two algorithms that find high-level change specifications evident in a set of changes. Both algorithms have been implemented in a tool we call `spdiff`. Finally, a few examples of change specifications inferred by `spdiff` in Linux are shown. We find that the inferred specifications concisely capture the actual collateral evolution performed in the examples.

SAMMENFATNING

"Medført evolution" handler om nødvendigheden af at opdatere adskillige programmer medført af ændringer i et bibliotek brugt af alle programmerne. I denne afhandling behandles emnet om at forstå sådanne medførte evolutioner ved, automatisk, at aflede en høj-niveau specifikation of ændringer set i en given mængde af opdaterede programmer.

Vi har formaliseret et begreb vi kalder "transformationsdele". Transformationsdele viser hvornår en given specifikation af ændringer kan ses i en mængde af opdaterede programmer. Baseret herpå, har vi defineret en relation som beskriver hvornår en specifikation af ændringer er en del af en anden specifikation af ændringer. Dette kan yderligere bruges til at afgøre om en specifikation af ændringer er maksimal. Endelig har vi beskrevet to algoritmer til at finde høj-niveau specifikationer af ændringer i en given mængde af opdaterede programmer. Begge algoritmer er implementeret i et værktøj, som vi kalder `spdiff`. Vi viser resultatet af nogle få anvendelser af `spdiff` på ændringer i Linux. De afledte ændringer fanger på en konsis måde de medførte evolutioner, som var blevet udført.

ACKNOWLEDGMENTS

There are numerous people that I would like to thank for somehow being helpful to me during my Ph.D. studies. In particular I would like to thank my supervisor, Julia Lawall, for consistent, quick, and helpful advise on just about any of the (more or less silly) questions I have had. Your guidance changed the way I think about research in a way I think is better.

I would also like to thank Professor Siau-Cheng Khoo from the National University of Singapore for being my host when I visited NUS during winter 2008. I have so many fond memories of Singapore and I really enjoyed the collaboration with you.

Finally, I would like to thank David Lo who is now working as an assistant professor at the Singapore Management University. Your energy with respect to research and general helpfulness is an inspiration to me. I am grateful that you took the time to visit me and my family in Copenhagen.

CONTENTS

I SEMANTIC PATCH INFERENCE 1

1 INTRODUCTION 2
- 1.1 Example-based change inference 3
  - 1.1.1 Tranformation parts 3
  - 1.1.2 Algorithms and implementations 5
- 1.2 Structure of the dissertation 6

2 RELATED WORK 7
- 2.1 Change vocabulary 7
- 2.2 Program transformation systems 9
- 2.3 Program pattern discovery 10
  - 2.3.1 Inference of program behavior 11
  - 2.3.2 Clone detection 13
- 2.4 Change detection 24
  - 2.4.1 Text based differencing 24
  - 2.4.2 Tree differencing 27
  - 2.4.3 Higher level approaches 31

3 SETUP 38
- 3.1 The language of TERMS 38
  - 3.1.1 Constructing TERMS 38
- 3.2 Term patterns 40
  - 3.2.1 Abstracting terms 42

4 TRANSFORMATION PARTS 46
- 4.1 Properties of common change descriptions 46
  - 4.1.1 Towards a definition 47
- 4.2 Tree distance based transformation parts 49
  - 4.2.1 Work-function 50
  - 4.2.2 Term-distance 50
- 4.3 Subsumption of program transformations 52
- 4.4 Extending to changesets 53
- 4.5 Non-global common changes 55

II ALGORITHMS AND IMPLEMENTATION 58

5 CONTEXT-FREE PATCH INFERENCE 59
- 5.1 Motivating example 59
- 5.2 Context-free patches 62
  - 5.2.1 Application function 62

5.3 Algorithm 63
5.3.1 A simple algorithm 64
5.3.2 Towards a refined algorithm 66
5.3.3 The refined `spfind` algorithm 71
6 CONTEXT-SENSITIVE PATCH INFERENCE 73
6.1 Motivating example 73
6.2 Semantic patches 75
6.3 Semantic patterns 76
6.4 Finding semantic patterns 77
6.4.1 Occurrences & Pruning Properties 77
6.4.2 Algorithm 78
6.4.3 Constructing semantic patches 79
6.5 Implementation 80

III REAL-WORLD APPLICATION 83
7 EXPERIMENTS 84
7.1 Examples of context-free patches 84
7.2 Examples of context-sensitive patches 86
8 CONCLUSION 89
8.1 Summary 89
8.2 Future work 89
8.2.1 Evaluation and engineering 89
8.2.2 Exploration of other transformation languages 90

BIBLIOGRAPHY 92

Part I

SEMANTIC PATCH INFERENCE

Chapter 1 INTRODUCTION

In the case of open-source software, such as Linux, where the developers are widely distributed, it must be possible to exchange, distribute, and reason about source code changes. One common medium for such exchange is the patch [43]. When making a change in the source code, a developer makes a copy of the code, modifies this copy, and then uses `diff` to create a file describing the line-by-line differences between the original code and the new version. He then distributes this file, known as a *patch*, to subsystem maintainers and mailing lists for discussion. Once the patch has been approved, other developers can apply it to their own copy of the code, to update it to the new version.

Patches have been undeniably useful in the development of Linux and other open-source systems. However, it has been found that they are not very well adapted for one kind of change, the *collateral evolution* [48]. A collateral evolution is a change entailed by an evolution that affects the interface of a library, and comprises the modifications that are required to bring the library clients up to date with this evolution. Collateral evolutions range from simply replacing the name of a called library function to more complex changes that involve multiple parts of each affected file. Such changes may have to be replicated across an entire directory, subsystem implementation, or even across the entire source code. In the case of Linux, it has been shown that collateral evolutions particularly affect device drivers, where hundreds of files may depend on a single library [48].

The volume and repetitiveness of collateral evolutions strain the patch-based development model in two ways. First, the original developer has to make the changes in every file, which is tedious and error prone. Second, developers that need to read the resulting patch, either to check its correctness or to understand what it will do to their own code, may have to study hundreds of lines of patch code, which are typically all very similar, but which may contain some subtle differences. An alternative is provided by the transformation system Coccinelle, which raises the level of abstraction of patches to *semantic patches* [49]. A semantic patch describes a change at the source code level, like an ordinary patch, but is applied in terms of the syntactic and semantic structure of the source language, rather than on a line-by-line basis. Semantic patches include only the code relevant to the change, can be abstracted over irrelevant subterms using meta-variables, and are independent of the spacing and line breaks of the code to which they are applied. The level of abstraction of semantic patches furthermore implies that they can be applied to files not known to the original developer – in the case of Linux, the many drivers that are maintained outside the Linux source tree.

Despite the many advantages of semantic patches, it may not be reasonable to expect developers to simply drop the patch-based development model when performing collateral evolutions. For the developer who makes the collateral evolution, there can be a gap between the details of

an evolution within a library and the collateral evolution it entails. Therefore, he may still find it natural to make the required changes by hand in a few typical files, to better understand the range and scope of the collateral evolution that is required. Furthermore, the standard patch application process is very simple, involving only replacing one line by another, which may increase confidence in the result. Thus, developers may find it desirable to continue to distribute standard patches, with or without an associated semantic patch.

What is then needed is a means of mediating between standard patches and semantic patches, by inferring semantic patches from standard patches. We propose a tool, `spdiff`, that infers semantic patches from a collection of standard patches implementing a common set of transformations. The Linux developer who makes a change in a library that affects its interface can perform the collateral evolution in a few files based on his knowledge about how drivers typically make use of the library, and then apply `spdiff` to produce a semantic patch that can be applied to the other files automatically. Complementarily, the developer who needs to read an existing standard patch implementing a collateral evolution can apply `spdiff` to the patch to obtain a more concise, abstract representation of the changes that are performed, as well as information about any deviations from these changes, which may represent bugs or special cases of which he should be aware. If the developer maintains proprietary code outside the Linux kernel source tree, he may furthermore use the inferred semantic patch to apply the necessary changes.

1.1 EXAMPLE-BASED CHANGE INFERENCE

CONTRIBUTIONS The main contributions in the work presented in this dissertation are

A. An abstract definition of a concept of transformation parts, and

B. Two algorithms and implementations for performing common change inference of program transformations expressed using the semantic patch language, SmPL provided by Coccinelle [49].

In the following we give a brief overview of the two contributions and then proceed with a brief overview of the structure of the dissertation.

1.1.1 *Tranformation parts*

Our approach for inferring semantic patches suitable for performing collateral evolutions is based on finding changes from a representative set of programs and their corresponding modified versions. For illustration suppose a set of pairs of programs is given such that the first component of each pair corresponds to the original program and the second component corresponds to the updated version of the program. The left part of Figure 1 illustrates such a situation. Using a standard patch as extracted by the unix `diff` program, we can describe the changes between between t_i and t'_i. This is illustrated by the p_i on the arrow from t_i to t'_i.

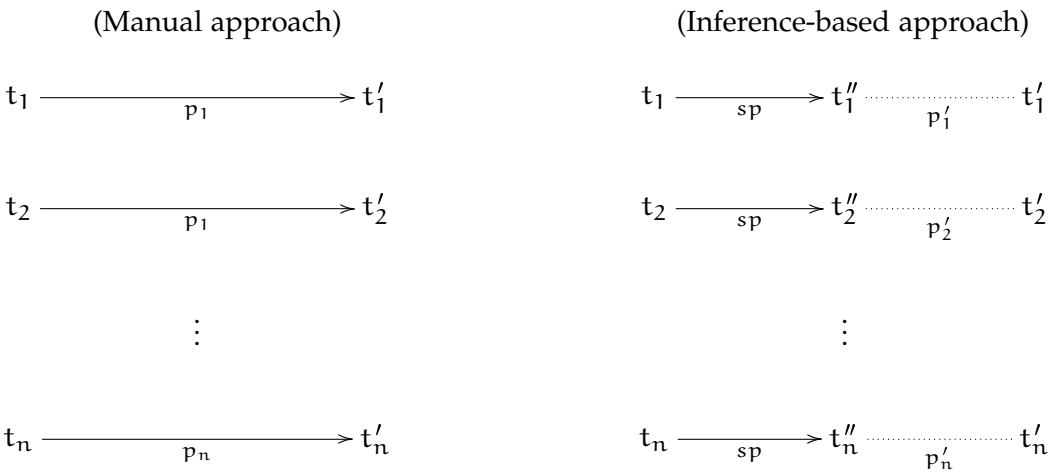


Figure 1: Applying `diff` for each pair of programs (left). Applying `spdiff` to the set of all pairs of terms (right).

The problem is now, that while each of the standard patches p_i describe how to update t_i into t'_i, the set of these patches is too verbose to convey the *common* changes that were applied to all of the original programs. In this dissertation we describe a method to extract the common changes applied in a set of pairs of programs as illustrated in the right part of Figure 1. Thus, while the situation on the left produces n standard patches ($\texttt{diff}\ (t_i, t'_i) = p_i$ for $1 \leqslant i \leqslant n$) when we apply our tool $\texttt{spdiff}\ \{(t_i, t'_i), \ldots, (t_i, t'_i)\} = sp$ we obtain just one description of the common changes. In Figure 1 we furthermore illustrate with dotted arrows that after performing the changes expressed in the common patch, there may still be changes that need to be performed for each program.

In order for the description, sp, of the common changes that `spdiff` finds to be useful there are two properties that it should have.

A. The changes it describes should be changes that are actually performed on each t_i in its transformation into t'_i.

B. The description should express as much of the common changes as possible.

DESCRIBING ONLY ACTUAL CHANGES The problem of detecting the changes actually performed between just one version of a program and its updated version is not entirely straightforward. In Section 2.4 in Chapter 2 we describe several other approaches that address this problem. The major difference between finding changes between one specific version of a program and another and common changes between several programs is that the denotation of the locations in the program to change needs to be different. For the single-program change detection, one can rely on explicitly denoting the locations that change; e.g. line n of file f was modified or the node in the abstract syntax tree with label q was removed. For the common change detection, one can generally not expect that locations can be denoted in this manner. A more general denotation of locations is needed. The problem is then to find a denotation of locations that is

general enough to cover all the locations modified in all relevant files, but no so general that it also denotes locations that were *not* changed. Consider a description such as "all calls to `kmalloc` with two arguments were modified to call `kzalloc` with the same arguments instead". It may easily be the case that not *all* calls to `kmalloc` should be modified. In Chapter 4 we addresse this issue in more detail by defining a notion of a "transformation part" which captures that a change is in fact safe to perform in the sense just outlined.

DESCRIBING AS MUCH AS POSSIBLE The second property that we require of the description of common changes that our method finds, is that it should express as much of the common changes as possible. For example if a particular function call was renamed in all locations but also had its arguments swapped, the change description should contain both of those changes. In Chapter 4 we use the transformation parts concept to define a subsumption relation on transformations. The maximal change is then the one that subsumes all others. Maximality of the inferred change descriptions therefore naturally depend on our choice of change description language. We therefore make this dependency a feature of our subsumption relation by parameterizing the definition of transformation parts by an application function of the descriptions. Thus, the definition of transformation parts relies only on the *result* of applying a change description, not on knowledge of the language in which changes are specified.

1.1.2 *Algorithms and implementations*

The abstract definition of common change inference has been used to design two algorithms that each solve the change inference problem for a specific change description language; more precisely, the first algorithm finds common changes that can be expressed in a simple term rewrite language while the second algorithm finds changes that can be sensitive to control-flow paths. The set of transformations possible in the latter language is a superset of those of the former. Both algorithms have been implemented in the functional programming language OCaml.

COMMON PARTS OF THE ALGORITHMS As mentioned, the definition of transformation parts is parametrized by an application function. For each algorithm we therefore present a change description language and the associated application function for applying specifications written in the language.

ALGORITHM FOR CONTEXT-FREE CHANGE INFERENCE The algorithm for finding context-free changes finds changes that can be expressed as a sequence of simple rewrite rules. The algorithm is split in two parts. The first part finds rewrite rules that express common changes applied in the examples provided. The second part then tries to grow those rewrite rules into a sequence of rules making use of the subsumption of change descriptions to only return the largest of the change descriptions.

ALGORITHM FOR CONTEXT-SENSITIVE CHANGE INFERENCE The second algorithm finds common changes that can be expressed with temporal properties on the control-flow paths of the programs. There are three main parts of the algorithm: 1. Finding changes between each pair of programs, 2. Finding a description of common temporal properties, and 3. Constructing the largest change description based on the common temporal properties and the changes found for each individual pair of programs.

CHANGE DESCRIPTION LANGUAGE In this work, we have chosen to express changes in terms of the SmPL language provided by the Coccinelle project. The SmPL language is specifically designed to express common changes in several programs (collateral evolutions). SmPL provides a WYSIWYG approach to the specification of changes in the sense that the descriptions of the changes *look* like the pieces of code affected. We conjecture that the WYSIWYG feature of SmPL eases the understanding of the actual impact of the changes described in the specification. This is in particular important in our setting because we seek to provide a high-level description of the changes made that makes it easy to see the common changes that have been performed in the example programs given. SmPL specifications are called semantic patches to emphasize their similarity with standard patches and their difference by allowing control-path (semantic) sensitive change specification specifications.

1.2 STRUCTURE OF THE DISSERTATION

This dissertation is organized in three main parts. The first part introduces (Chapter 1) the common change inference problem that we are addressing, presents related work, provides a framework in terms of which our approach is formalized, and finally defines the problem of common change inference that we are addressing (Chapters 1-4). The second part of the dissertation presents two algorithms that solves the common change inference problem relative to a particular language for describing changes. The final part of the dissertation presents a few examples of applying the implementation of the algorithms to examples from the Linux kernel. The final part also contains a conclusion that summarizes the main contributions of the work presented in this dissertation as well as directions for future work.

Chapter 2 RELATED WORK

In this chapter we present other work that is related to ours. Three topics are considered:

A. Transformation systems

B. Program pattern discovery

C. Change detection

A transformation system modifies a program given a transformation specification into a new program. In order for a transformation system to be useful to help performing collateral evolutions (i.e. updating library-using code in response to library changes) it is beneficial if it is possible to specify transformations that enable updating any library-using program. We therefore first present a few transformation systems that can be used to address the transformational part of the problem of collateral evolution.

Tool support for updating several code locations in potentially many files is only part of the collateral evolution problem. A main hypothesis in this work is that a significant problem is also obtaining the transformation specification in the first place. We therefore next consider work that relates to finding a collateral evolution transformation specification. A natural thing to consider is that a specification that updates many code locations in a similar fashion must somehow be able to identify common characteristics of the code locations to update. Under the title "Program pattern discovery", we consider work that identifies common characteristics of code. Finally, being able to identify the changes made in a program can be helpful to either manually or automatically construct a transformation specification for collateral evolution. We therefore also consider related work that identifies changes in programs. Some approaches find changes that are explicitly only useful for describing the changes made in a specific programs whereas some are able to generalize the detected changes in order to describe more changes compactly.

First however, we introduce concepts that will be used to classify the related work.

2.1 CHANGE VOCABULARY

The notion of *change vocabulary* is introduced by Kim et al. [32] as a characterization of the kinds of changes that can be performed by a transformation system or discovered by change detection approaches. For a program transformation system, the change vocabulary is the language in which program transformations are expressed. For a change detection system, the change vocabulary characterizes the kinds of changes that the system can detect.

In the context of collateral evolution, it is natural to consider how to specify a transformation in a manner which is able to express common characteristics. We identify two issues to consider for the related work we consider in the following: 1. how are code locations denoted, and 2. how can non-common or irrelevant details be abstracted.

DENOTATION OF LOCATIONS Transformation specifications can be split in two separate concerns: identifying the locations to transform and performing the change in the identified locations. Below, we state a taxonomy of location denotations in terms of expressiveness.

A. Explicit denotation of locations; The denotation of the location is specific to the program in which the location occurs and is not generally usable as a denotation of locations in other programs. If the program is represented as a tree with unique node labels, a location denotation could simply be the node label. If there are no unique node labels, the path from the root of the tree to the node in question could be used.

B. Context-free denotation of locations; The denotation of locations to update can only depend on the subparts of the location being denoted. For example one can specify an update of all function headers for functions of a certain type, but not an update of all function calls occurring *within* function definition of a certain type because then the locations to modify are dependent on the context in which they occur.

C. Context-sensitive denotation of locations; The denotation of locations to update can depend on the context of the locations. For example one can specify an update of all method calls occurring within a class of a certain type or all function calls performed after another function call during execution of the program.

ABSTRACTION MECHANISMS Abstraction mechanisms are ways for the transformation specification to abstract irrelevant or non-common details of the locations to update. Below, we consider three mechanisms for abstraction of *parts of* the code locations.

A. No abstraction mechanism; no abstraction of locations possible. This approach is mostly used for tools that only ever consider the changes between two programs at a time.

B. Anonymous subpart abstraction; subparts of locations can be abstracted by a single place-holder.

C. Named subpart abstraction; subparts of locations can be abstracted by multiple named place-holders. We call named placeholders *meta-variables*. Using the same meta-variable in multiple locations imposes an equality constraint on the subparts abstracted by the meta-variable.

There is a subtle interaction between the denotation of locations and abstraction mechanisms. One could imagine a system with a change vocabulary that allowed context sensitive denotation of locations, but had no abstraction mechanisms. It could then be possible to specify such things

as "all calls to `close()` that follows a call to `open()` should be changed into calls to `close_all`". However, if one added named subpart abstraction to the change vocabulary, it would be possible to specify that only the calls to `close(X)` that follow calls `open(X)` where the argument abstracted by `X` should be the same. I.e. only pair calls that handle the same resource.

2.2 PROGRAM TRANSFORMATION SYSTEMS

The goal of the work by Kingsum and Notkin [15] is to ease the update of application programs in response to library changes; i.e. they are addressing the issue of collateral evolution. They observe that, when there are many application programs using a library, putting the main part of the burden of easing the collateral evolution on the library developer is less of a burden overall. Therefore, the library developer is given the task of embedding special change descriptions *inside* the interface definition files of a library whose interface has changed; i.e. for C programs, he should put the change description inside the affected `.h` header files. The tool provided by Kingsum and Notkin can then extract the change specification from the interface files. The extracted change specification can subsequently be applied automatically to update application programs.

The following kinds of changes are supported by the system presented by Kingsum and Notkin:

- add and remove include files or change their names
- add, remove or rename function names.
- move a function from one include file to another
- add, remove and reorder function parameters
- change default arguments to non default
- change the return type of a function
- change the type of a function parameter
- change the meaning of an incoming function parameter
- remove, add, rename, or change a struct field
- remove, add, rename, or change a global variable

The changes that can be specified are *context-free* in the sense that it is not possible to specify a transformation that depends on a location separate from the transformation site or a transformation that depends on being in a specific context. For example, it is not possible to update only the function calls that appear as a parameter of another function call. However, a change *can* be dependent on information available at compile-time at the locations to be updated. E.g. a transformation can be dependent on the value or type of subterms.

Stratego/XT is a framework for constructing program transformation systems [55]. Transformations are specified in transformation rules that are rewrite rules on the abstract syntax trees of the input programs to transform. Rewrite rules may contain meta-variable. In order to reduce the gap between the concrete syntax of programs and their abstract tree representation, it is possible to specify rewrite rules in terms of the concrete syntax of the input language instead of in terms of the abstract syntax trees. Transformation rules are applied according to transformation strategies that are also specified by the user when instantiating the Stratego/XT framework. A transformation strategy allows specification of how to traverse the ASTs and in which order to apply transformation rules. Transformation strategies furthermore allows information gather during traversal to be propagated to transformation rules. Propagating information allows specification of context-sensitive rewrite rules.

JunGL is a domain-specific language for specifying refactorings. Programs are represented by a graph structure that contains all the information that the language can query. The program graph includes information about ASTs, variable bindings, control flow etc. JunGL uses features of functional programming such as higher order functions and pattern matching to express transformations and logic programming to formulate queries on the programs for collecting information for use in the transformations. In particular regular expression *path queries* can be used to identify paths in the program graph. Refactorings specified in JunGL may include meta-variables (called logic variables because of the logic programming influence) and due to the queries may be context-sensitive. In contrast to both Coccinelle and Stratego/XT, refactorings specified in JunGL are not in terms of the concrete syntax of the programs.

Coccinelle is a program transformation and query engine. It provides a specification language, Semantic Patch Language (SmPL), in which to specify transformations and queries on C code. Coccinelle was initially designed to handle the collateral evolution problem in Linux, but has since also been used to transform other systems as well as to find bugs [11, 54]. Specifications are called semantic patches or semantic matches for transformation specifications and query specifications respectively. A semantic patch (match) is similar in syntax to a standard patch as shown in Example 2.5 but includes a number of additional features. Notably, a semantic patch may specify 1. context sensitive transformations such as only changing function calls that appear on specific control-flow paths, 2. it can abstract subparts of the program, 3. and finally since semantic patches borrow syntax from that of standard patches, a semantic patch *looks* like the code that should be transformed. The final point is important because it helps minimize the gap between the specification of the transformation and the impact of the transformation on code.

2.3 PROGRAM PATTERN DISCOVERY

An underlying assumption in our approach is that common changes can be expressed in terms of common patterns. We therefore next consider some approaches that find commonly occurring patterns in programs.

We reuse the concepts of location denotation and abstraction mechanism introduced in Section 2.1 to classify the kinds of patterns that can be found by a particular approach. In

Section 2.3.2 we refine the concepts of location denotation and abstraction mechanisms to be more suitable to classify clone detection approaches.

2.3.1 *Inference of program behavior*

When programs become large or just grow old, the documentation of the behavior of the program may become incorrect or otherwise outdated. The problem can be even worse in case a formal specification was given for the original version of the program. The specification (and documentation) is useful to be able to verify, automatically or not, that the program behaves as expected. Evolving the specification together with the program can be a difficult task and at least for programs that initially had no specification it can be very difficult to define a specification of the program behavior; There may be patterns in large programs that the developer is not even fully aware of.

A solution to the problem of obtaining a program specification is to infer the specification from the program itself. An issue that needs to be addressed is that programs may not exhibit perfect behavior that can be inferred; There could be undiscovered bugs in the program or the programmer may know that certain conditions are always satisfied so complete conformance to, e.g., a strict open-close policy may not be needed. Engler et al. [19] made the observation that in mature programs, one can expect the program to behave correctly in *most* of the cases. Deviations from the common behavior can then be considered to be bugs, but also simply as exceptions. This observation is the foundation of the approaches that we consider in this section.

An number of approaches for finding program patterns uses data mining techinques to find sets of program elements (e.g. function or method calls) frequently occuring together. Data mining can find *many* sets of frequently occurring program elements. Therefore, some sort of filtering of the results is performed in order to reduce the number of patterns found.

The approach taken in PerraCotta by Yang et al. [61] is to look for a fixed set of pattern *templates* in traces generated at run-time from instrumented programs. Specifically, Yang et al. look for patterns of the form $(PS)^*$; That is, repeated patterns where an event P is followed by an event S. An example instantiation is acquiring and releasing a lock. Since the traces are obtained from run-time generated traces, PerraCotta can take contextual information about the events into account. For example, the acquire and release events can be associated with the object being acquired and released. E.g. the trace may consist of $\langle$`lock1.acq`, `lock2.acq`, `lock2.rel`, `lock1.rel`$\rangle$. Based on the two subsequences $\langle$`lock1.acq`, `lock1.rel`$\rangle$ and $\langle$`lock2.acq`, `lock2.rel`$\rangle$ PerraCotta can generalize the subsequences by abstracting the name of the concrete object into $\langle$`lock.acq`, `lock.rel`$\rangle$. When looking for such patterns, PerraCotta can find many uninteresting patterns. To reduce the number of patterns, two heuristics are employed: 1. name similarity of paired events and 2. non-embedding of events. The name similarity heuristic removes the patterns where the events have non-similar names. This is based on an observation that related events have similar names; E.g. `lock` is related to `unlock`. The second heuristic removes the patterns where the first event simply serve as a "wrapper" for the second event. Suppose the pattern is $\langle a, b\rangle$ and that a is a function call. If b occurs in the definition of the a function, we

can say that a is a wrapper for b. Finally, once all patterns of the $(PS)^*$ form is found, PerraCotta extends the found patterns by a transitive closure. The subsumed patterns are subsequently removed.

PerraCotta has been used to find patterns in large Java programs and the kernel of Windows Vista. When verifying one of the found patterns of Windows Vista a bug in the NTFS file system was found.

DynaMine is tool that finds frequently occurring program patterns [42] in Java programs. DynaMine uses data mining of revision histories to find candidate patterns and dynamic analysis to validate the found patterns. The basic observation made by Livshits and Zimmermann [42] is that methods calls checked into the revision history together often form a pattern. The data mining process returns a set of methods that were frequently introduced into the revision history together. In order to reduce the size of the returned set, DynaMine applies filtering and ranking to the set. Filtering is applied to the input set to the data mining process. The main part of the filtering removes calls to methods that are very common such as `equals`, `add`, `getString`, `size`, and `get`. To further reduce the number of found sets of method calls, DynaMine applies ranking of the found results. The ranking is based on a observation that small changes to the revision history such as one-line additions, often represent bug fixes. If a one-line addition contained a method call, DynaMine rank the found frequent sets of method calls that also include the added method call higher than ones that do not. From the sets of frequent method calls, the user can easily form simple patterns. Livshits and Zimmermann have found several patterns corresponding to matching method calls–i.e. `open` followed by `close`. The selected candidate patterns are then validated against an instrumented program analyzed using dynamic analysis.

The PR-Miner tool by Li and Zhou finds "programming rules" in large code bases that is based on frequent itemset mining [2]. A programming rule is basically an association rule of the form $A \Rightarrow B$ with confidence c where A and B are sets of program elements. The interpretation of the rule is that if a function in the program contains the elements X it also contains the elements Y with a probability of c. Program elements are taken from function definitions and can be, in contrast to DynaMine, any part of the function. PR-Miner first parses the input program and produces an itemset database. Each itemset is constructed from a function of the program by hashing selected program elements of the function into numbers. In particular, local variables are hashed based on their type instead of their name in order to capture more common characteristics. In order to reduce the number of programming rules to construct and to make the found programming rules more concise, PR-Miner only finds *closed* programming rules. Closed rules subsume other rules that have the same support and therefore allow fewer rules to be constructed while yielding more concise rules. The rules found by PR-Miner does not have to conform to rule template as assumed by DynaMine. On the other hand the programming rules can not capture the repetitiveness of the $(PS)^*$ rule template as found by DynaMine.

2.3.2 *Clone detection*

In the following we give an overview of clone detection. Then we present individual work in more detail. Very broadly speaking, clone detection is the process of finding similar pieces of code scattered throughout a program. What is meant by "similar" and a "piece of code" varies from one approach to another. We thus first present a general formulation of clone detection in which the precise meaning of "piece of code" and "similarity" are parameters.

2.3.2.1 *Clone detection framework*

Let *pcs* denote a piece of code from a program *Prg*. Let $pcs \sim pcs'$ denote that the two pieces of code denoted by *pcs* and pcs' are similar according to some definition of similarity. When $pcs \sim pcs'$ holds, pcs, pcs' is called a *clone pair*. A set of program pieces that are all similar is called a *clone class*; i.e. *cls* is a clone class if and only if $\forall t, t' \in cls : t \sim t'$. The clone detection problem is defined in Definition 2.1 below.

2.1 *Definition (Clone detection problem) Given a program Prg, let* $P(Prg)$ *be the multiset of all possible pieces of code of Prg. Clone detection is the problem of finding the set of all clone classes:*

$$Clone(Prg) = \{cls \subseteq P(Prg) \mid \forall t, t' \in cls : t \sim t'\}$$

From Definition 2.1 we can see that that the similarity measure should at least be 1) reflexive ($pcs \sim pcs$) and 2) symmetric (if $pcs \sim pcs'$ then $pcs' \sim pcs$); if the similarity measure is not reflexive and symmetric, the pair of two identical pieces of code does not constitute a clone pair. A further consequence is that no code piece belongs in any clone class.

Definition 2.1 suggests a naïve method to find all clone classes: Maintain a set of clone classes, C found so far, and consider each piece of code, *pcs* (from $P(Prg)$): For each clone class found so far, add the code piece to the class if it is similar to all code pieces in that class. Finally, add a new singleton class to the set of clone classes $\{pcs\} \cup C$. The naïve method has a run-time complexity of $\mathcal{O}(2^{|P(Prg)|})$ because there are exponentially many clone classes.

PIECES OF CODE There are four common representations of pieces of code used in clone detection. In some approaches, several representations of the same program are used.

Clear-text When using the clear text files as a representation of programs, a piece of code is defined to be a sequence of lines of code. Representing code using clear-text means no parsing of the source language is required and, in principle, approaches using clear-text as their representation of code can therefore readily be applied to programs written in any language.

Token-stream Representing a program as a stream of tokens implies that a piece of code is defined as a sequence of tokens. As with the clear-text representation, little parsing is needed.

Abstract syntax tree When using an abstract syntax tree representation of programs, pieces of code can be proper subtrees of the AST or a sequence of sibling subtrees–each sibling then represent a statement-level term of the program. Approaches based on abstract syntax trees require parsing of the programs. In most cases, a language specific front-end is designed that transforms the input programs into some internal tree structure. The actual clone detection is then done on this internal tree representation and the results then need to be mapped back to the original code.

Program dependence graph A program dependence graph (PDG) [21] is a graph representation of a program where the nodes of the graph are statements and predicates and edges represent data and control dependencies. A piece of code is then a subgraph of the program dependence graph.

SIMILARITY OF PIECES OF CODE The set of clones that can be detected by any approach depends on how the similarity of two pieces of code is defined. In almost every approach similarity of code pieces is defined in terms of a function that abstracts parts of the code pieces in order to establish that the abstracted code pieces are identical. Bellon et al. [9] and Li et al. [39] both identify a number of types of comparison functions and Evans et al. [20] further identifies an additional similarity measure. Below we generalize the different types of code similarity measures into four main groups:

Type 1 Identical code pieces: $pcs \sim pcs' \iff pcs = pcs'$. Two pieces of code therefore only constitute a clone pair when they are completely identical.

Type 2 Equivalence up to replacement of parts of the code. Two pieces of code *pcs*, *pcs′* constitute a clone pair when there is some way of replacing parts of *pcs* so that it becomes identical to *pcs′*. As an example, $pcs = axzx$ and $pcs' = ayqy$ constitute a clone pair because we can replace x with y and z with q in *pcs* to obtain *pcs′*. Typically, only a certain restricted set of parts of the code is allowed to be replaced; e.g. it is common to allow replacement of identifiers and literals, but not keywords and operators.

Type 3 Equivalence up to replacement of parts of the code with the *same* symbol. Technically, type 3 can be considered a special case of type 2 where every part is replaced by a special symbol "_".

Type 4 Equivalence of properties of the code pieces. Instead of comparing the code pieces directly, compare properties of the code pieces; for example Jiang et al. [28] compare characteristic vectors of parse trees. The characteristic vectors capture syntactic information about the parse trees such as the number of statements. Another example is to define code piece similarity using a distance metric.

Generally, the set of clones that can be found using a type i ($0 < i \leqslant 4$) similarity measure is a subset of the clones that can be found when using a type $i + 1$ similarity measure.

	Type 2	Type 3	Type 4
Clear-text	**dup**		
Tokens	Wrangler	CCFinder, *Koschke*	JPlag
AST	CloneDigger	CP-Miner, Asta	Deckard, *Kontogiannis*, CloneDR
PDG			*Gabel*, GPLAG, *Krinke*, *Komondoor*

Table 1: Clone matrix

2.3.2.2 *Description of clone detection approaches*

In this section we describe the clone detection approaches mentioned in Table 1. The entries in the table use the name of the tool implemented, if available and otherwise a surname of one of the authors describing the approach. An author name is indicated by italic letters. In the table, 15 approaches to clone detection are categorized according to their representation of code pieces and similarity measure. Each approach is categorized according to where it fits in best. Some of the approaches, however, are not perfect matches; e.g. CloneDR uses two measures for comparing code pieces, but one of them can be considered primary.

CLEAR TEXT-BASED CODE CLONES

***dup**, 1992* **dup** is a program for finding duplicated pieces of code [5, 6, 7]. A program is represented as a sequence of lines from the source text and a piece of code is a sequence of consecutive lines within this source text. Two pieces of code constitute a clone-pair if there is a one-to-one renaming of "parameters" such that the code pieces are identical. A parameter is a substring of a line of code representing a variable, a constant, a macro name, or a structure member name. **dup** finds subsequences of consecutive lines of code that appear more than one time using a modified suffix tree construction algorithm [44]. The modification allows **dup** to find parametrized matches.

2.2 *Example (Parametrized matching)* Consider the following two pieces of code.

```
x=y-z;
if (y>z)
  m=1;
h=f(x);
y=x;
```

```
x=b-c;
if (b>c)
  n=1;
h=f(x);
c=x;
```

The first four lines of each piece constitute a code clone because of the following one-to-one renaming: $\{\texttt{y} \mapsto \texttt{b}, \texttt{z} \mapsto \texttt{c}, \texttt{m} \mapsto \texttt{b}\}$. The line from both code pieces (`y=x;` and `c=x`) cannot be included because then the renaming would have to include both $\texttt{y} \mapsto \texttt{b}$ and $\texttt{y} \mapsto \texttt{c}$.

dup is able to process large code bases in reasonable time. It has been applied to code that has 1.1 million lines of code [5]. Due to the fact that **dup** works on the clear-text of programs, the clone detection process does not know about what subsequences constitute syntactic units of the program being analysed. **dup** may therefore end up finding e.g. a clone that starts with the last few lines of a function definition and the beginning of the next function definition. Another problem is that it is sensitive to line-breaks in the code.

TOKEN-STREAM BASED APPROACHES The token-stream based approaches typically split the process of detecting clones into two parts: A) a language dependent part that parses or scans the input programs in preparation for B) the language *independent* part that finds subsequences of tokens that constitute clones.

CCFinder, 2002 The CCFinder tool is split into a language dependent front-end and a language independent back-end that performs the actual clone detection [30]. There are currently front-ends for C/C++, Java, COBOL, VB, and C#.

The front-end takes a program and produces a sequence of tokens. When producing the token stream, the front-end performs a number of normalizing transformations to the program with the goal of the making code pieces with different syntactic structure but the same meaning, syntactically identical. An example for the C front-end is to convert `if (b)foo();` to `if (b){ foo (); }`. Finally, all tokens representing types names, variables, and constants are replaced with the special wildcard token.

The back-end clone detection takes a token-stream and constructs a suffix tree. Two subsequences of tokens from the program token-stream are considered to constitute a clone pair if they are identical (type 3). CCFinder has been applied to large programs (millions of lines of code).

CCFinder does not suffer from being sensitive to line-breaks as **dup** and it avoids detecting subsequences of tokens that do not represent proper syntactic units, by only allowing certain tokens to start and end a token subsequence.

JPlag, 2002 The JPlag tool described by Prechelt et al. [51] is a tool for detecting plagiarism. It takes as input a number of programs and tries to find similar pieces of code in the programs. If a substantial similarity is found, there is a good chance that one program is obtained by plagiarizing the other. In that respect plagiarism detection is analogous to clone detection. However, whereas CCFinder transforms a *single* program into a token-stream, JPlag transforms several programs into several token-streams. One could then use the suffix-tree approach of CCFinder to detect plagiarism by concatenating all the constructed token-streams into one long.

JPlag works in two steps: 1. Transform all input programs into token strings. 2. Compare all token strings for plagiarism.

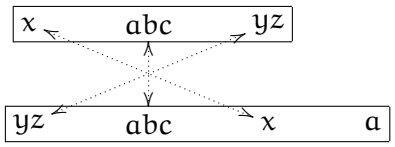
Transforming programs to token strings The first step performed by JPlag is to convert the given input programs to token strings. When producing token strings, JPlag ignores white-space (including line breaks), comments, and identifiers. Some tokens can occur in multiple contexts with different meaning. For example the open-brace token, `{` can denote both the start of a method body and the start of a compound statement. Generally, the front-end process that converts a program into a token string should therefore produce tokens that reflect the essential program structure rather than the syntactic appearance; i.e. instead of using an `OPEN_BRACE` token for the curly brace at the beginning of a method definition in Java, the front-end should use a more descriptive `METHOD_BEGIN`. Not doing so could cause an `OPEN_BRACE` corresponding to the beginning of a method to be matched with an `OPEN_BRACE` corresponding to a compound statement. JPlag has front-ends for Java, Scheme, C++, C#, and plain text.

Comparison of similarity In order to compare two token strings for similarity, JPlag uses the "greedy string tiling" algorithm described by Wise [58]. The overall aim is to find longest substrings that occur in both token strings such that none of the substrings overlap each other. Given a set of substrings found by string tiling, the similarity of two substrings is then given by: $sim(s_1, s_2) = 2 * coverage/(|s_1| + |s_2|)$, where *coverage* is the number of tokens on both strings that occur in a found substring.

2.3 *Example (Greedy string matching)* Let the strings s_1 and s_2 be given as:

$$\begin{aligned} s_1 &= xabcyz \\ s_2 &= yzabcxa \end{aligned}$$

The result of greedy string tiling of the two strings is the following set of substrings: $\{abc, yz, x\}$.



The value of *coverage* is $|abc| + |yz| + |x| = 6$ so the value of $sim(s_1, s_2)$ is $2 * 6/(6 + 7) = 12/13 \sim 0.92$.

Using a different string tiling approach than the greedy used here, one could have selected a different set of substrings: $\{ax, bc, yz\}$. This set of substrings gives rise to the same similarity of the two strings being compared.

As illustrated in Example 2.3 similarity of two token strings as computed by JPlag is not merely a matter of comparing whether one string can be renamed into the other. Therefore we categorize JPlag as using a type 4 similarity measure.

Koschke, 2005 Koschke et al. describes a clone detection approach that combines an abstract syntax tree approach with a token-stream based approach [35]; the basic idea is to produce an abstract syntax tree by parsing and then to flatten the AST into an isomorphic token-stream by a pre-order traversal. Given a token-stream, *Koschke* then uses a suffix-tree algorithm to find sequences that occur at least twice in the program.

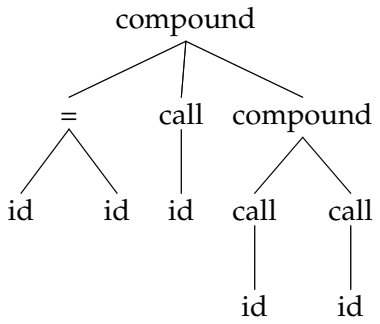
The labels on the nodes of the AST correspond to the grammatical type of the subtree rooted at that node. Tokens correspond to AST node-types and contain information about how many of the subsequent tokens corresponds to the subtree rooted at the token, but not about the actual values (lexemes) of identifiers and literals. We illustrate the conversion of an AST to a token-stream in Example 2.4 below.

2.4 *Example (Isomorphic AST flattening)* Below two pieces of code and their corresponding ASTs are shown.

Program 1:

```
{
  b = i;
  foo();
  { bar();
    fub();
  }
}
```

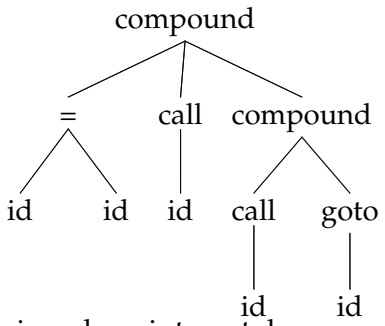
Abstract syntax tree



Program 2:

```
{
  a = i;
  foo();
  { bar();
    goto lab;
  }
}
```

Abstract syntax tree



The token-streams are stated below. Numbers in sub-script on tokens correspond to the number of subsequent tokens that occurs in subtree rooted at the token in the corresponding AST.

$$\texttt{compound}_{10}, =_2, \texttt{id}_0, \texttt{id}_0, \texttt{call}_1, \texttt{id}_0, \texttt{compound}_4, \texttt{call}_1, \texttt{id}_0, \texttt{call}_1, \texttt{id}_0$$

$$\texttt{compound}_{10}, =_2, \texttt{id}_0, \texttt{id}_0, \texttt{call}_1, \texttt{id}_0, \texttt{compound}_4, \texttt{call}_1, \texttt{id}_0, \texttt{goto}_1, \texttt{id}_0$$

Assume the two pieces of code appear somewhere in the same program. The longest subsequence of consecutive tokens occurring in both of the token-stream is:

$$\texttt{compound}_{10}, =_2, \texttt{id}_0, \texttt{id}_0, \texttt{call}_1, \texttt{id}_0, \texttt{compound}_4, \texttt{call}_1, \texttt{id}_0$$

However, that token-stream does not correspond to a complete subtree of the AST because it misses the last statement which was not common. This information is already evident in the token-stream because there are not 10 tokens following the $\texttt{compound}_{10}$ token.

As illustrated in Example 2.4 there can be subsequences of tokens that appear multiple times in the token-stream of a program that do not correspond to a complete syntactic unit. *Koschke* extracts only the part of such invalid subsequences that does correspond to a syntactic unit. Finally, *Koschke* is able to find sequences of clones that do not constitute a complete syntactic unit. From the invalid subsequence in Example 2.4, *Koschke* would therefore find the sequence $\langle =_2, \texttt{id}_0, \texttt{id}_0, \texttt{call}_1, \texttt{id}_0 \rangle$ instead of the two subsequences representing the assignment and function-call.

Wrangler, 2006 We consider the description of the clone detection tool Wrangler by Li and Thompson from their 2009 paper [39]. The Wrangler tool dates back to at least 2006 though. Wrangler is a tool for refactoring of Erlang/OTP [4] programs that can additionally perform clone detection. Wrangler detects clones in two steps. 1. In the first step clones are found from a token stream representation of programs using a type 3 similarity measure and a suffix tree method. The similarity measure considers all identifiers to be mapped to the same wildcard. 2. Then, the type 3 token based clones are mapped to their corresponding ASTs and the clones are refined using a type 2 similarity measure where there are named wildcards for identifiers and distinct wildcards for function names. During the refinement into type 2 clones, some of the pieces of codes found to be clones in the first steps may not be type 2 clones because it may not be possible to rename one code piece *consistently* into another. For example `f(a+b,b+a)` and `f(b+a,b+a)` are considered type 3 clones because both can be renamed to `f(_+_,_+_)` but Wrangler does not classify the two code pieces as type 2 clones because it is not possible to give a one-to-one renaming of the identifiers `a` and `b`; The requirement of a consistent renaming is similar to the requirement posed by **dup**. The mapping of token-based clones into AST based allows Wrangler to rename *bound* variables more easily.

APPROACHES BASED ON ABSTRACT SYNTAX TREES

Kontogiannis, 1996 The approach taken by Kontogiannis et al. [34] is based on abstract syntax trees. Two type 4 similarity measures are presented. The first measure computes feature vectors of `begin-end` blocks and uses euclidian distance to decide whether the two pieces of code are similar. An example feature for a piece of code is the number of functions it calls. The second similarity measure uses a dynamic programming approach to compute the cost of transforming one feature vector into the other.

CloneDR, 1998 Baxter et al. [8] present a tool called CloneDR, that finds clones based on the abstract syntax tree of a program. Roughly, CloneDR works by comparing every subtree with every other subtree of the tree representing the entire program to detect whether the

compared-with subtree occurs elsewhere–if it occurs elsewhere a clone pair has been found. Three issues are addressed by the tool: 1. near-miss clones, 2. sub-clones, and 3. scalability.

Near-miss clones are two subtrees that are not identical, but still similar. The similarity of two subtrees is given as a ratio between the shared parts of the subtrees and the parts that are different. This type 4 similarity measure is the main method used by CloneDR to detect when two subtrees constitute a clone pair.

Sub-clones are simply subtrees that are embedded in larger subtrees also detected as clones. CloneDR removes all sub-clones during the clone detection process.

Even for programs of moderate sizes, the number of subtrees can easily become large and comparing every subtree with every other subtree then becomes infeasible. In order to address the issue of scalability, two techniques are used: 1. pruning small subtrees and 2. pre-filtering subtrees into equivalence classes using a more coarse-grained similarity measure. Small subtrees are pruned from the multiset of code pieces (subtrees) $\mathsf{P}(Prg)$ based on the intuition that only clones over a certain (user-specified) size are interesting. In order to avoid comparing every code piece in $\mathsf{P}(Prg)$ with every other code piece, CloneDR first partitions $\mathsf{P}(Prg)$ into clone classes according to a coarse-grained (type 3) similarity measure: two subtrees are in the same clone class if and only if they are identical after all identifiers have been renamed to the same wildcard. After the partitioning, each of subtrees in the clone classes are compared using the more fine-grained similarity (type 4) function mentioned above. A further novel refinement of the pre-filtering similarity measure is that it takes into account commutative operators. Thus, two subtrees representing e.g. addition expressions but with swapped operand expressions are considered identical. Finally, all subtrees found to be clones by the above process are subsequently used to find sequences of subtrees that are all code clones.

CP-Miner, 2004 CP-Miner is a tool that looks for copy-pasted code in a program [40]. The tool works by parsing the input program into a number of sequences of numbers representing sequences of statements corresponding to a basic block of the program. A statement is turned into a number using a hashing function. The hashing function is insensitive to the names of variables and the values of constants. Thus two statements that are the same except for the names of variables and the values of constants are hashed to the same number (type 3 similarity).

In order to find frequently occurring subsequences of statements CP-Miner uses a variant of the sequence data mining algorithm by Agrawal and Srikant [1]. The subsequences found by the sequence mining algorithm used by Agrawal and Srikant [1] are not necessarily contiguous in the sequences from which it was found. For example, from the sequences $\langle 1,2,3\rangle$ and $\langle 1,5,4,3\rangle$ the subsequence $\langle 1,3\rangle$ can be mined with a frequency threshold of 2. The sequence mining variant used by Li et al. [40] in CP-Miner adds a gap constraint. The gap constraint specifies the maximum number of elements that may occur in supporting sequences of a mined subsequence. With a gap constraint of 2 (or larger), the previous subsequence can be mined, but with a gap constraint of 1 it can not because of the presence of both 5 and 4 between 1 and 3 in the sequence $\langle 1,5,4,3\rangle$. Strictly speaking, the similarity measure is no longer a type 3 measure because the similarity of two pieces of code is not simply a matter of comparing normalized versions of the code pieces.

Once CP-Miner has found all frequent subsequences it tries to form larger sequences by composing the found subsequences–recall that the initial subsequences are mined from a set of sequences corresponding to basic blocks in the program. Two subsequences can be composed if they correspond to sequences of statements that are neighbors in the program.

The fact that CP-Miner allows gaps in the subsequences of statements it finds makes it able to detect more clones than previously mentioned approaches. Naturally, CP-Miner may find false positives. The main cause of false positives is the hashing of all identifiers to the same value. In order to overcome this problem, Li et al. observe that when a developer copy-pastes a piece of code and modifies the name of a variable, all of the occurrences should be expected to be renamed in the pasted piece of code. When comparing whether two pieces of code one can therefore make a mapping of names from one piece of code to the other. If not all (or most) of the occurrences of a variable have been renamed in the other piece of code, it is deemed likely that the two pieces of code should *not* be considered clones.

DECKARD, *2007* Jiang et al. present an clone detection tool, DECKARD, that is based on characterizing subtrees using numerical vectors [28]. A vector contains information about the number of various constituent elements of the subtree represented; e.g. the number of statements, variables, conditionals. Two vectors are considered similar if their Euclidean distance is below a given threshold (type 4 similarity measure). Vectors are generated both for single subtrees, but also for sequences of (sibling) subtrees. Similar vectors are clustered into clone classes using Locality Sensitive Hashing [17]. Locality Sensitive Hashing hashes two vectors with small Euclidean distance to the same hash value and distant vectors to different hash values. Finally, DECKARD removes subsumed clones and clusters that contains only one vector–i.e. a vector is not a clone if there is only one occurrence of the represented tree.

DECKARD has been applied to large code bases such as Linux and compared with other clone detectors (CP-Miner and CloneDR). Jiang et al. report that DECKARD finds more clones than both approaches using comparable running-times.

Asta, 2007 The Asta tool by Evans et al. [20] finds type 3 clones by abstracting parts of the AST representing the program in question with the same wildcard. In contrast the approaches considered previously, Asta allows replacement of any subtree and not only a fixed set of elements such as identifiers and constants. For an AST t containing n subtrees, there are 2^n possible ways to replace subtrees with the wildcard subtree which is too many even for small programs.

In order to address this issue, Asta first finds a set of candidate trees from subtrees of the input AST and subsequently refine the candidates, to find more detailed clones. A set of candidate trees is constructed from an AST by producing all possible trees with wildcards until a user-specified depth in the AST. I.e. for $t = a(b(f(x), g(x)), h(y))$ and depth of 2, Asta generates the candidates $\{t, a(\_,\_), a(b(\_,\_), h(\_))\}$. In the refinement phase, Asta considers each wildcard of the candidate patterns and tries to refine it further with the restriction that the refined pattern should match the same locations as the candidate pattern. Subsequently, subsumed patterns are removed. Finally, Asta allows ranking of the found clones by their size, frequency, and similarity.

The similarity ranking considers the ratio between the size of the pattern and average size of the subtrees of the original that the pattern matches. Thus, the similarity ranking is a measure of how different the pattern is from the subtrees it matches. If a pattern contains no wildcards it has a similarity ranking of 1. If the subtrees abstracted by wildcards in a pattern are on average very large, the similarity ranking approaches 0.

A potential benefit of allowing replacement of more than identifiers and constants with a wildcard is that it allows finding structural clones; e.g. a pattern `foo(arg1,_)` can match both `foo(arg1, arg2->bar)` and `foo(arg1,arg2)` which approaches merely abstracting identifiers would not find. The difference in the two function calls may indicate an mistake in the latter where the developer forgot to reference the `bar` field of the `arg2` structure.

CloneDigger, 2008 The work by Bulychev and Minea refines the approach of Evans et al. by allowing subtress to be replaced with several named wildcards (type 2). Bulychev and Minea describe [12, 13] an method for finding clones based on anti-unification [50]. The algorithm has been implemented in a tool called CloneDiggerthat finds clones in Python and Java programs.

The basic idea is similar to the naïve clone detection algorithm outlined in Section 2.3.2.1: A set of clone classes found so far is maintained. Each clone class is represented by a tree with named wildcards obtained by anti-unification. When deciding whether to add an AST representing a statement to a clone class, the statement AST is anti-unified with the anti-unifier representing the clone class. If the obtained anti-unifier has a low (user defined) tree edit distance from the representing anti-unifier, the statement AST is added to the clone class. Finally, the representative anti-unifiers are used to find sequences of statements that occur more than twice using a suffix tree approach.

APPROACHES USING SEMANTIC INFORMATION

Komondoorand Krinke, 2001 Komondoor and Horwitz make use of the program dependence graph to look for clones in a program [33]. The approach works by converting each procedure in the program to its PDG and finding isomorphic subgraphs. Given the PDGs for two procedures, the tool then uses program slicing [56] to find isomorphic subgraphs. A sub-graph of the PDG for a procedure does not necessarily correspond to contiguous pieces of the procedure. Rather, the subgraph corresponds to *dependent* (control or data) parts of the procedure.

Because the ordering in the PDG is given by the data- or control-dependence of the procedure as evident in the PDG, the approach used by Komondoor and Horwitz can detect that two pieces of code constitute a clone pair even though they have different interleaved statements in the procedure in which the pieces of code appear–or even different orderings of the statements of the code pieces. The main drawback of the approach is that it does not scale well to very large code bases–deciding subgraph isomorphism is NP-complete in general. Komondoor and Horwitz describe the application of their tool to programs of 11.540 lines of code with a running time of about 1 1/2 hour.

A similar approach is taken by Krinke. Instead of considering standard PDGs, Krinke consider *fine-grained* program dependence graphs which are more detailed versions of the standard PDGs

[36]. The approach taken by Krinke uses a different approach for finding isomorphic subgraphs in the constructed PDGs that relies on limiting the lengths of paths search in the PDGs. Krinke report better running times than those of Komondoor and Horwitz but are still not able to scale the approach to very large programs.

GPLAG, 2006 Liu et al. describe a tool called GPLAG that looks for plagiarized code in programs based on PDGs [41]. GPLAG compares the PDGs of two functions in order to detect whether one of the functions can be considered a plagiarized version of the other. A PDG is considered a plagiarized version of another if a subgraph of one graph is isomorphic to a subgraph of the other and the subgraph is larger than a given value γ. In that case the PDGs are said to be γ-isomorphic.

In order to make the approach scalable to large programs, parts of the search space is pruned by two filtering techniques. In order to avoid comparing every PDG with every other PDG, GPLAG applies two filters:

- A lossless filtering in which small PDGs are removed completely from the set of PDGs that needs to be compared and also it avoids comparison of a pair of PDGs when one is larger than the other by the given value γ. The filter is considered lossless because no pairs of PDGs that could be have isomorphic subgraphs are removed.
- A lossy filtering in which historgrams of the PDGs are used in combination with statistical methods [37] to decide whether it is likely that two PDGs are γ-isomorphic. The lossy filter is faster to compute that subgraph isomorphism but may cause GPLAG to miss some pairs of γ-isomorphic pairs of PDGs.

Liu et al. show that GPLAG can find more detailed cases of plagiarized code than previous tools for detecting plagiarized code and that the filtering techniques applied have a major impact on the running time. Still, testing of subgraph isomorphism is NP-complete so some programs can cause a running-time of hours or more. GPLAG circumvents such problems by using a timeout value for γ-isomorphism testing so that other PDGs can be compared. Using a timeout is both unsound and incomplete, but allows GPLAG to proceed to testing other PDGs for isomorphism.

Gabel, 2008 A recent approach for clone detection that uses semantic information is given by Gabel et al [24]. The algorithm described by Gabel et al. finds clones from a selected set of sub-PDGs of functions, but maps the PDGs to their corresponding ASTs and use a tree similarity measure to detect clone pairs. The tree similarity measure used is the same as the one employed by DECKARD: Euclidean distance between characteristic vectors representing the trees. Selection of sub-PDGs to compare for similarity is done using a program slicing technique.

The algorithm presented by Gabel et al. have been implemented in a tool. Gabel et al have applied the tool to five open source projects–Linux being the largest of those. The tool is shown to find more clones with larger average sizes.

2.4 CHANGE DETECTION

In this section we describe related work that focuses on finding changes in programs.

2.4.1 *Text based differencing*

The `diff` tool [26, 43] is one of the earliest tools for finding differences between two versions of a text-file. It finds the minimum number of changes between two text files with a line based granularity by using a dynamic programming scheme. The basic idea is to compute a minimal edit script that can be used to update the original program to obtain the new. The only edit operations available are insertion and deletion of lines. Finding a minimal edit script is therefore dual to finding the longest common subsequence of the two programs. The run-time complexity of the differencing algorithm is in the worst case $\mathcal{O}(n^2)$ where n is the number of lines of the longest of the two input programs. The original UNIX version described by Hunt and McIlroy [26] is non-heuristic, but later implementations such as GNU `diff` [43] make use of heuristics to further improve the typical run-time behavior. GNU `diff` is based on a $\mathcal{O}(md)$ algorithm where m is the sum of the lengths of the input programs and d is the length of the shortest edit script transforming one program into the other [45]. In the following we do not distinguish between the two implementations of the differencing algorithm. The result of applying `diff` to two programs is frequently referred to as a *patch*. A patch describes the line-by-line differences of the two files by explicitly denoting the lines in the original program to delete or keep, and also explicitly denoting where new lines should be added. An example of a patch is shown in Example 2.5 below. Furthermore, `diff` does not provide any form of abstraction mechanisms, so the changes reported are usually very verbose and low-level.

2.5 *Example (Application of `diff`)* The following two programs are retrieved from `http://alfedenzo.livejournal.com/170301.html` which discusses the *patience sorting* based differencing algorithm presented next.

```c
#include <stdio.h>

 // Frobs foo heartily
int frobnitz (int foo)
{
  int i;
  for (i = 0; i < 10; i++)
  {
      printf ("Your answer is: ");
      printf ("%d\n", foo);
  }
}

int fact (int n)
{
  if (n > 1)
  {
      return fact (n - 1) * n;
  }
  return 1;
}
```

```c
#include <stdio.h>

int fib (int n)
{
  if (n > 2)
    {
      return fib (n - 1) + fib (n - 2);
    }
  return 1;
}


 // Frobs foo heartily
int frobnitz (int foo)
{
  int i;
  for (i = 0; i < 10; i++)
    {
      printf ("%d\n", foo);
    }
}
```

Part of the result of applying GNU `diff` with the left program as original and right program as updated program, is show below.

```
@@ -1,26 +1,25 @@
 #include <stdio.h>

- // Frobs foo heartily
-int frobnitz (int foo)
+int fib (int n)
 {
-  int i;
-  for (i = 0; i < 10; i++)
+  if (n > 2)
    {
-      printf ("Your answer is: ");
-      printf ("%d\n", foo);
+      return fib (n - 1) + fib (n - 2);
    }
+  return 1;
 }
```

From the patch in Example 2.5 one can see that GNU `diff` have matched the definition of `frobnitz` in the original program with the definition of `fib` in the new program. The only thing these two functions have in common, however, are the curly braces used for grouping statements. A more likely description of the changes the developer intended between the two programs can be stated as: 1. move the definition of `frobnitz` below the definition of `fib` and 2. remove `printf`-statement from the `fib` function. Implementing these two changes in a patch would result in a patch that first removes all the lines of the `frobnitz` function definition, removing the line with the `printf`-statement and finally inserting all the original lines of the removed function after the `fib` function. Because of the large amount of common curly braces, this patch has the same edit cost as the one in Example 2.5 above.

The Patience diff algorithm tries to overcome the "curly-braces" problem outlined above. It does so by trying to group the original and updated program into blocks that should correspond to each other. The grouping is based on unique common lines in the two input programs. Thus, lines which only contains curly braces or other program elements that are frequent in all programs are less likely to be used to group the input program. Instead lines containing names of functions defined are more likely to be used for grouping. The algorithm is based on finding longest common subsequences using patience sorting [10]. For Example 2.5, Patience diff would find the following patch which was previously mentioned as the more likely change the programmer intended.

```
 #include <stdio.h>

+int fib (int n)
+{
+  if (n > 2)
+  {
+      return fib (n - 1) + fib (n - 2);
+  }
+  return 1;
+}
+
   // Frobs foo heartily
  int frobnitz (int foo)
  {
    int i;
    for (i = 0; i < 10; i++)
    {
-       printf ("Your answer is: ");
        printf ("%d\n", foo);
    }
  }
-
```

```
-int fact (int n)
-{
-  if (n > 1)
-  {
-      return fact (n - 1) * n;
-  }
-  return 1;
-}
```

A reason the text based differencing algorithms sometimes does not capture the potential intend of the programmer is that they do not know anything about the structure of the programs they are extracting differences from. On the other hand they can readily be applied to programs written in any language and even programs written in multiple languages combined in the same text file. In the following we consider change detection approaches that are based on a more structured representation of the programs. The benefit is that more precise changes can be reported, but the issue is usually that the approaches are restricted to the programs that can be represented and that they have a higher run-time complexity than the simple text based differencing.

2.4.2 *Tree differencing*

A tree differencing algorithm takes as input two trees, one being the old version of the tree and the other the new version. The differencing algorithm then tries to find a minimal set of operations on the old version of the tree that transform it into the new version of the tree.

We next consider an number of tree differencing algorithms though the first algorithm is not designed specifically for finding changes in programs, but rather "structured information". The rest of the approaches are specifically designed for finding changes in trees representing programs.

CLASSIFICATION OF TREE OPERATIONS Common to all of the considered approaches is that the tree operations are specific to the tree to which they are applied; the tree operations use *explicit denotation* of the subtrees to modify. Typical operations are: 1. *inserting* of new a leaf node with a specific label and value, 2. *deleting* a node, 3. *moving* a node from one place to another, and 4. *updating* the value of a node to a different value. These tree operations can be viewed as having *no abstraction mechanism* because they can not depend on the values of the nodes to modify, only the location of the node matters.

An early approach to finding changes between two C programs is given by Yang [62]. Yang describe the cdiff tool that is based on similar dynamic programming methods as `diff`, but finds differences between two trees representing the programs. The differences are given in terms of removals or additions and are presented by synchronous printing of the programs. I.e. the programs are printed side-by-side and annotations on the printed programs mark the differences. The cdiff tool works by parsing the input programs into an internal tree representation and then

traverses the two constructed trees in parallel. Non-leaf nodes represent a non-terminal node with further structure whereas leaf-nodes represent terminal nodes such as identifiers that have no further structure. When comparing two non-terminal nodes it uses dynamic programming recursively to find the best matching of embedded nodes. Example 2.6 illustrate the matching of non-terminal nodes.

2.6 *Example (Matching of non-terminal nodes)* Consider the following two sub-parts of two programs. The left piece is intended to be part of the original program while the right piece is intended to be part of the modified program.

```
while(p) {
  x = y + z;
  a = b + c;
}
```

```
while(p) {
  x = y + z;
}
while(p) {
  a = b + c;
}
```

When finding the changes between the two pieces of code, cdiff finds that in the left while block the second assignment statement was deleted and that a while-statement was inserted after the existing while-statement (cdiff does not detect moves). In contrast, the text-based `diff` tool would simply find that two lines were inserted in the middle of the while-statement.

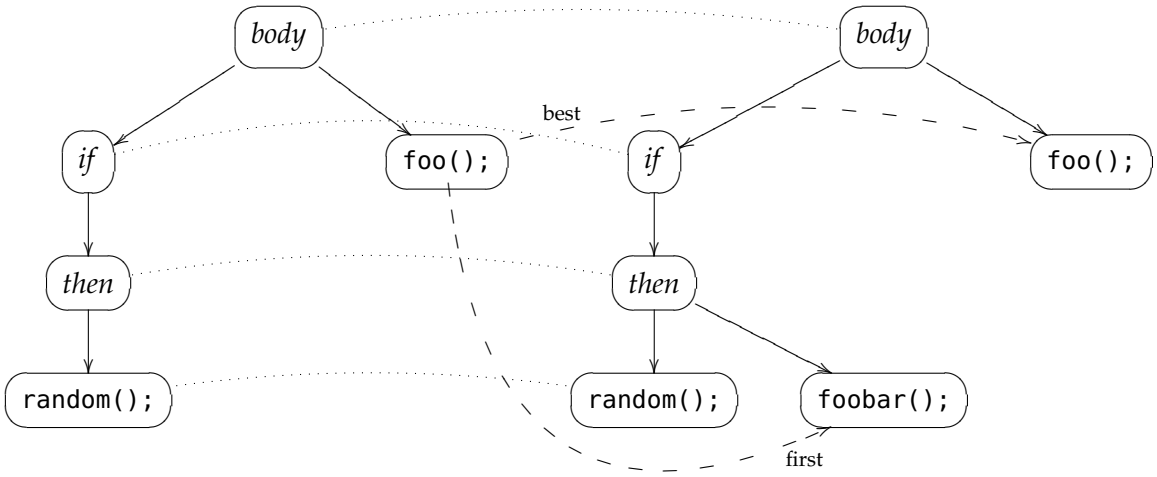
As with all the tree-differencing approaches considered, the changes found by cdiff have no abstraction mechanisms and use an explicit denotation of the modified locations. Therefore, a renaming of a variable name is reported as the deletion of the old variable and subsequent insertion of a new variable for every location in the tree in which the variable is used.

Chawathe et al. [14] describe a method to detect changes in structured information based on a ordered tree representation of the original and updated version. Their goal is to derive a compact description of the changes between the original and updated tree. To this end, they define a notion of a minimum cost edit script. An edit script is a sequence of operations where each operation has an associated cost determined by some measure of structural similarities between the values of the nodes. In order to find the minimum edit script, a matching of the nodes in the two input trees needs to be made. The matching maps nodes of the original tree to nodes of the updated tree. Nodes that are mapped by the matching should be considered equal. Nodes from the orignal tree not in the domain of the mapping are considered deleted and nodes in the new tree not in the range of the mapping are considered to be newly inserted nodes. In order to compute a matching of the tree nodes, a leaf node comparison function is used. Under the assumption that only one pair of leaf nodes is considered similar by the comparison function, the mapping of nodes from the old tree to the new can be deterministic. The consequence is that the edit script constructed from the node mapping is also a minimum edit script. This assumption generally holds for documents where the internal nodes only serve a structural role whereas the main content is found in the values of the leaf nodes; a LaTeX document is an example where the assumption holds when paragraphs are taken to be the value on leaf nodes,

because then it is almost certain that all leaf nodes will have different values. The leaf node comparison function used by Chawathe et al. is the Levenshtein string edit distance [38] finds a the minimum edit distance between two strings using only insertion and deletion operations on characters.

The CHANGEDISTILLER tool developed by Fluri et al [23] provides a version of the tree differencing algorithm by Chawathe et al. [14] better suited for finding changes in (the abstract syntax trees of) programs. Fluri et al. find that the basic assumption underlying the tree differencing method suggested by Chawathe et al. does not hold for programs when the leaf nodes represent non-compound statement-level elements of the program. It is not uncommon that very similar statements, for example printing of debugging information, are spread throughout a program.

2.7 *Example (Simple tree change)* Consider the following two trees representing the old version of part of a program body (left) and the updated version (right). For inner nodes, the labels of nodes are shown while for leaf nodes, the value is shown.



Matching of nodes is illustrated with stippled lines. Using the matching approach of Chawathe et al. the `foo();` leaf node will be matched with the `foobar();` leaf node in the tree on the right. Using the best-match approach of CHANGEDISTILLER, the `foo();` node will be matched with the `foo();` node in the tree on the right.

The consequence of matching `foo();` with `foobar();` is that the node is considered to have changed. The edit script generated by the algorithm of Chawathe et al. would first move the `foo();` node to be a sibling of the `random();` node, update `foo();` into `foobar();`, and finally insert a new node as child of `body` with value `foo();`. In contrast, CHANGEDISTILLER can simply generate an edit script that consists of inserting a new node with the value `foobar();`.

In order to make the tree differencing approach by Chawathe et al. more suitable for finding changes in programs, CHANGEDISTILLER matches leaf nodes according to the *best* match instead of the first. CHANGEDISTILLER also uses a bigram string similarity comparison function instead of the Levenshtein string edit distance metric used by Chawathe et al. Bigram string similarity is

better at identifying permutations of substrings than the Levenshtein comparison. For example, a renaming of `verticalDrawAction` into `drawVerticalAction` is detected as such by the bigram similarity comparison.

The changes found by ChangeDistiller are in terms of tree operations. The tree operations are then used by Fluri et al. to *classify* the changes made in a program across versions using *change types* [22]. A change type is a high level summary of possibly more than one tree operation. A simple example is "*Statement Ordering Change*" which summarizes a particular node moving tree operation but does not reveal what statements were moved. Example 2.8 below illustrate the difference between tree operations and change types.

2.8 *Example (Simple program update)* A function and its updated version is shown below

```
void foo () {
  int x = 42;
  x = x + x;
  bar(x);
}
```

```
void foo () {
  long y = 42;
  y = y + y;
  bar(y);
}
```

Three tree *update* operations are necessary to update the program. In terms of the operations defined by Fluri et al., the operations needed to update the original program are:

$Upd(n_1, \texttt{long y = 42;})$
$Upd(n_2, \texttt{y = y + y;})$
$Upd(n_3, \texttt{bar(y);})$

Each of the update operations explicitly denotes the node in which to set a new value (non compound statement). In terms of the change types, there would be just one: "Statement update" with a count of three.

The change types of ChangeDistiller can provide a classification of the changes performed in a program update. However, change types are not very useful as a measure of what actually changed in the program. The approach taken by Neamtiu et al. [46] can be seen as a middle-ground between the very low level tree operations and the very high level change types. Neamtiu et al. finds differences on the function level between two programs by matching the ASTs computing a bijection between types and variables along the way. The bijection makes it possible to influence matching of ASTs, so that once a variable have been identified as renamed (i.e. it is put into the bijection), the matching of two ASTs representing the variable can result in a match stating that the two ASTs does not correspond any further changes. For example, from Example 2.8 Neamtiu would extract a more concise description of what was changed in the original program: "the variable `x` was renamed to `y` everywhere in the function" and "the type of `x` changed to `long`". Neamtiu et al. compute such changes from original and updated programs by attempting to match the ASTs of the old functions with the corresponding new function. Neamtiu et al. use the name of the function in order to identify which of the functions in the original program should be matched with which functions in the updated program. Names

which disappear in the new programs are reported as removed and new names in the updated program are reported as added functions. Thus, the approach is not well adapted to changes where an old function is renamed but otherwise not changed as is the case for rename-method refactorings because it would report the old function as removed and the renamed function as added.

Matching two ASTs involves traversing the two ASTs in parallel and computing a bijection between variables (for both local and global variables) and a bijection between types. Based on these variable- and type-maps Neamtiu et al. can then report whether a program merely had its variables renamed or types changed as illustrated in Example 2.8–or whether some other change was applied in a function body.

Dex is a tool that finds changes between two C-programs represented by "abstract semantic trees" [52]. An abstract semantic tree is a traditional AST with additional semantic information about types and binding of local variables. The result of applying Dex to two trees is an edit script that transforms the original tree into the updated tree. Like in previous tree differencing approaches the edit scripts returned provides no abstract mechanism and denote locations to modify explicitly. Raghavan use Dex to collect statistics about changes related to bug-fixes. The statistics are similar to the change types that Fluri et al. find. A major difference between Dex and ChangeDistiller and the method described by Chawathe et al. is that when computing the matching of tree nodes, previous matches are taken into account. For example if a node correspond to a variable declaration is matched with a node in the updated tree that is also a variable declaration but with a different name, the variable is considered to have been renamed and this information is used when matching nodes where the variable is subsequently used. The ability to take such previous matches into account is similar to the variable name bijection computed by Neamtiu et al.

2.4.3 *Higher level approaches*

Another approach to extracting high-level descriptions of the changes made in a two versions of a program is given by Jackson and Ladd [27]. The basic approach is based on the semantics of the program being analysed. Concretely, Jackson and Ladd define the semantics of a function in a program in terms of an approximate relation between the inputs and outputs of the function. The relation basically states that the value of a variable depends on the value of another variable. Their "Semantic Diff" tool can then compute the dependence relation for the old and new version of the function and report the differences between the relation.

Example 2.9 is taken from the article by Jackson and Ladd [27] because it is very illustrative of one of the strengths of their semantic diff tool.

2.9 *Example (Semantic diff)* Suppose the old version of a function is given as follows:

```
void add (int x) {
  if (x != HI)
    TOT = TOT + x;
  else
```

```
    TOT = TOT + DEF;
}
```

The dependence relation then contains the following pairs:

$$(\mathrm{TOT}, \mathrm{TOT}), (\mathrm{TOT}, x), (\mathrm{TOT}, \mathrm{DEF}), (\mathrm{TOT}, \mathrm{HI}), (x, x), (\mathrm{DEF}, \mathrm{DEF}), (\mathrm{HI}, \mathrm{HI})$$

The meaning of a pair (a, b) is that the value of a depends on the value of b. For example we notice that TOT depends on both itself and x and the value of x depends on nothing but itself.

We now swap the branches of the conditional and invert the conditional expression. We do not intend to change the behavior of the function.

```
void add (int x) {
  if (x = HI)
    TOT = TOT + DEF;
  else
    TOT = TOT + x;
}
```

The dependence relation now contains

$$(\mathrm{TOT}, \mathrm{TOT}), (\mathrm{TOT}, \mathrm{DEF}), (\mathrm{TOT}, \mathrm{HI}), (x, \mathrm{HI}), (\mathrm{DEF}, \mathrm{DEF}), (\mathrm{HI}, \mathrm{HI})$$

The difference is that `TOT` no longer depends on `x` and `x` no longer depends on itself but `HI` instead. The reason for the change in the dependence relation is simple: we made an error in the update of the function. The conditional expression is now an assignment instead of an equality test because we wrote = where we should have written ==.

As illustrated in Example 2.9, extracting the semantic differences between two versions of a program is useful to understand the behavioral impact of the changes. Changes such as simply renaming variables or modifying code-style are not detected as semantic differences. Likewise, semantic differences may not be able to express the changes that occur when a collateral evolution is applied to a program; A program update in response to an API update in a used library need not modify the semantics of the program in terms of observable changes in input/output behavior.

The JDiff tool by Apiwattanapong et al. is a tool for finding changed in Java programs [3]. The tool is an implementation of a graph differencing algorithm that finds changes in special control-flow graphs for object oriented programs. The algorithm (CalcDiff) underlying JDiff works by first matching classes and interfaces of the original and updated version of the program being analysed. The matching of classes and interfaces is based on a name-similarity measure. Once classes and interfaces have been matched, the methods of matching classes are compared. To compare methods, CalcDiff first constructs control-flow graphs for the methods. The constructed control-flow graphs explicitly encode features particular to object-oriented programs. For example method calls are represented in the CFGs by multiple "callee" nodes denoting the class from which the method is defined and for each such node its incoming edge is labeled with the class from which the method is invoked. Example 2.10 shows two programs and CFGs for a method in the program.

2.10 *Example (Extended CFGs)* Below, we show two Java programs. The right-most is a slightly modified version of the left-most. The modification is that class `B` now provides a definition of the method `m1`. Below each program we have constructed the extended CFG for the method `m3`.

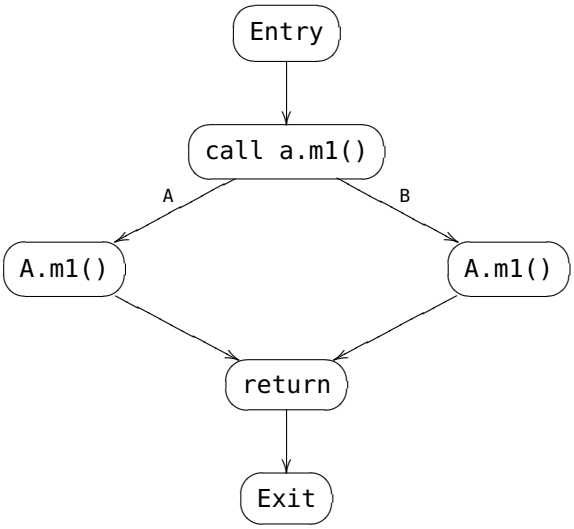
```
public class A {
  void m1 () {...}
}

public class B extends A {

  void m2 () {...}
}

public class D {
  void m3 (A a) {
    a.m1();
  }
}
```

CFG for original program

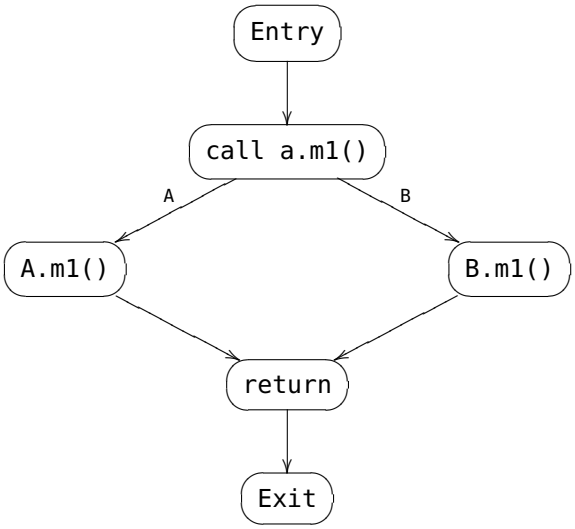



```
public class A {
  void m1 () {...}
}

public class B extends A {
  void m1 () {...}
  void m2 () {...}
}

public class D {
  void m3 (A a) {
    a.m1();
  }
}
```

CFG for updated program



Because the CFGs encode information about behavior related to object oriented programs, JDiff can find both direct changes, e.g. that a particular statement was changed syntactically, and indirect changes. Indirect changes are changes to the potential behavior of the program caused by some other change in the original program. Indirect changes are not immediate from the program text and would be missed by e.g. `diff`. Example 2.10 shows an example of an indirect change. A class that extends another class is modified to override the definition of a method inherited from the superclass. The consequence is that the method `m3` which calls `m1` may exhibit new behavior in the modified program. JDiff reports these indirect changes as well as the direct ones by explicit denotation of the locations in the code that have changed.

Kim et al. propose a method to infer "change-rules" from two versions of the same program [32]. Their goal is to construct a small set of change rules that capture many changes. Change rules express changes related to program headers (method headers, class names, package names, etc.). The basic shape of a change rule is given as: $\forall x \in$ `scope` : `transformation`, meaning that every match described by the scope is modified by the transformation. The scope, described using a variant of regular expressions, ranges over the textual representations of the previously mentioned headers. By using regular expressions as an abstraction mechanism the scope can be extended to e.g. all calls to a method that starts with the prefix `foo`. Thus, change rules can express that a given transformation was applied to a set of entities, which is more compact than simply enumerating all entities. The regular expressions do not support naming of abstracted subparts, so we classify the abstraction mechanism of the approach as "anonymous subpart abstraction".

Kim et al. also describe an extended version of the previously mentioned [31]. The method, implemented in a tool called `LSdiff`, finds *systematic changes* in programs. A systematic change is a high-level description of changes made in several locations and locations are no longer restricted to method headers only. The specification language provides an abstraction mechanism that allows abstracting subparts of locations by named meta-variables.

In `LSdiff` programs are represented by a set of *facts* that hold of the programs. For each program entity `LSdiff` computes the difference between the original and the updated entity. Based on all fact sets and difference sets, `LSdiff` can infer systematic changes evindent in the updates of the programs given. `LSdiff` expresses changes using logic rules. An example of a logic rule is `deleted_method(m, "getHost", t)` $\Rightarrow$ `added_inheritedfield ("getHost", "Service", t)`. The rule should be interpreted as "all types `t` that deleted the method with fully-qualified name `m` and short name `getHost` method, inherit `getHost` from `Service` instead.

Henkel and Diwan propose a system (CatchUp!) for evolving programs in response to library refactorings that is based on recording a log of the refactorings performed in the library [25]. CatchUp! makes use of the in build refactoring support in the Eclipse IDE to record refactorings performed to library in a log. The user can chose not to include refactorings in the log that he knows has no visible impact in libraries using the library. The log can subsequently be "replayed" to update programs that use the library. Eclipse performs refactorings by creating a refactoring object that when applied performs the specified refactoring. The create object traverses the internal representation of the source-code and produces new code. The refactoring object takes care of updating both the specific refactored location as well as usage-locations–i.e. for a rename method, it renames the method in the method declaration but also renames occurrences of the method name where it is used. To replay a refactoring log in order to update a program, CatchUp! reconstructs the refactoring object from the log. The intention is to have the refactoring object update usage locations in the program that needs updating. The system is implemented in an Eclipse plugin but there is no support for using the recorded log in other IDEs or otherwise outside of Eclipse.

Weißgerber et al. present a technique to identify likely refactorings in the changes that have been performed in Java programs [57]. Like Kim et al., they search for a fixed set of transformation types (in this case, rename method, add parameter, etc). Each transformation type has

an associated precondition that enables the transformation. They first collect various signature information about the old and new versions of a given file, and then use this information to determine whether the preconditions of any of the transformation types is satisfied. If a precondition is satisfied, the transformation is considered a refactoring candidate. They furthermore use clone-detection (CCFinder) to check whether the change performed by a candidate is likely to semantics preserving. The transformation types given by Weißgerber et al. do not support any kind of abstraction mechanisms such as meta-variables. Thus, two detected changes can not be generalised into a more compact description that covers both of them, as could potentially be done by the method given by Kim et al. Finally, the method is not able to detect when two refactorings have been applied to the same entity. According to Weißgerber et al. this is the main cause of impreciseness of their method.

Xing and Stroulia present a system, called Diff-Catchup, that helps developers update an program in response to refactorings in a library used by the program [60]. Given a point in the program that no longer compiles due to refactorings in a used library, the Diff-Catchup then 1) searches for changes to the API used at that point using *UMLDiff* [59], 2) tries to find suitable replacements using heuristics, and 3) collects usage examples for the found potential replacements. When collecting usage examples, Diff-Catchup looks for three specific usage examples: 1) *obtain-object* usage where the example shows how to obtain an object for nonstatic methods/fields of the required type in the new code, 2) parameter-list usage where the examples shows how to obtain values for added parameters to a method, and 3) replacement usage where the example shows how to update references to removed or replaced methods, constructors, or fields. In contrast to Henkel and Diwan [25] the developer is not dependent on using the same IDE as the developer of the library since no recorded log of refactorings is required. Also, the update of a program is not dependent on the library developer having constructed a complete update specification as required by Kingsum and Notkin [15]. To effectively use Diff-Catchup it is required that there is code already updated to use the new library. However, Xing and Stroulia find that the library code *itself* often contains enough such voluntary migrations. A drawback of using Diff-Catchup is related to its interactive approach. When a program does not compile due to library changes, the developer has to go through all of the compilation errors or warnings and invoke Diff-Catchup. After having performed a number of such assisted updates to the program, the developer might have learned what to do and is able to update the remaining code sites without assistance, but Diff-Catchup does not provide generalization features in order to automate the remaining updating. Indeed, one can classify Diff-Catchup as using explicit denotation of locations and providing no abstract mechanism.

Dig et al. propose a tool called RefactoringCrawler that is able to recover the refactorings performed by analysing two versions of a program (the original and refactored) [18]. The tool is finds refactorings in two steps. In the first step RefactoringCrawler identifies similar pairs of packages, classes, methods, and fields using a syntactic similarity measure. The similarity measure compares the entities by first hashing each entity into a sequence of numbers so that similar sequences represent similar entities. Thus, minor edits of entities results in only minor changes to the hashed numbers. This similarity measure is comparable to the one used by CP-Miner. Based on the found pairs of similar entities, RefactoringCrawler proceeds by iterating

a semantic analysis. In the iteration a log of refactorings discovered so far is used for discovering more refactorings which are subsequently added to the log. The semantic analysis can identify seven kinds of refactorings:

1. RenamePackage
2. RenameClass
3. RenameMethod
4. PullUpMethod
5. PushDownMethod
6. MoveMethod
7. ChangeMethodSignature

To detect that e.g. a pair of two syntactically similar methods constitute a particular refactoring, RefactoringCrawler consults a graph of "references" for the methods. The nodes of the reference graph are the packages, classes, and methods of the program. There is an edge between two nodes if the entity represented by one node somehow references the entity represented by the other node. An example of a reference is whether the entity (method) calls the other. Thus, to detect that a pair of syntactically similar entities corresponding to two methods constitute a RenameMethod refactoring, RefactoringCrawler essentially checks whether the two methods are used and occurs in similar contexts in the old and new program. If so, the pair is deemed a RenameMethod refactoring. The benefit is that RefactoringCrawler is able to detect refactorings when the old code is not removed after the refactoring, but merely declared deprecated.

In an approach based on data-mining, Schäfer et al. find framework usage changes based on already ported programs. The ported programs can be test programs within the framework itself or programs external to the framework that make use of the framework. The approach first extracts so-called *usage facts*, creates a transaction database from usage facts, and finally uses data mining to find association rules. Usage facts are extracted from pairs of corresponding classes in the old and new version of the program and includes such things as methods called by methods in the class, fields accessed, classes instantiated and classes extended etc. Usage facts additionally contains information from which version of the program it was extracted. Usage facts are then grouped into subsets according to a specific set of *change patterns* identified by Schäfer et l. Change patterns encode what usage facts correspond to specific changes in the program. An example is "*calls* becomes *accesses*" which encodes that a method call was replaced with a field reference and have the effect of grouping a usage fact in the original program about a method call with usage facts about field access in the new version into a transaction. Furthermore only usage facts from corresponding program entities are combined into transactions. I.e. usage facts from a particular method is only combined with usage facts from the corresponding updated method. Finally, transactions which do not corresponding to changes are removed. The rules found by association mining are of the form $A \Rightarrow B$ where A and

B are singleton sets of usage facts from the old and new version of the program respectively–in contrast to PR-Miner where A and B could be larger sets. In case multiple rules with the same antecedent (A) but different consequent (B) is found, only one is selected as the most likely valid change rule. The decision of which association rule should be considered the most likely is based on how similar A and B are when compared with a string similarity function.

The goal of the approach of Schäfer et al. and RefactoringCrawler is very similar, but since the former is able to find changes that are not caused by refactorings, it finds some changes that RefactoringCrawler is not able to. However, RefactoringCrawler is also able to find some changes that the approach of Schäfer et al. is not able to. The reason is that the ported programs analyzed by Schäfer et al. did not make use of the those particular parts of the framework. Finally, the approach of Schäfer et al. can find framework usage changes even when the framework did not remove the old code, but merely marked it as deprecated. The reason for this, is mainly to be found in the choice to analyze *framework-using* code over analyzing the changes within the framework itself as done by e.g CatchUp! and RefactoringCrawler.

Chapter 3

In this chapter we present a simple framework in terms of which we define our change inference method. The framework consists of a simple term language and term patterns. The term language consists of just two types of terms, atomic and compound. Using the term language we can represent the abstract syntax trees of C code. Term patterns are used to represent several terms in one compact representation, which is called pattern matching in the framework. We present a computable mechanism by which we can construct a unique pattern representing a set of terms.

3.1 THE LANGUAGE OF TERMS

While the approach described in this thesis targets C code, we formalize it using a simpler language, which we call the language of Terms. The syntax of Terms is given in Definition 3.1 below. C code is then translated to the simpler term-language, making sure that the results we find are presented in using the syntax of the C language. The two main benefits of such a separation are:

- Stating the formalization in using a simpler language allows us to focus on the core issues of change inference and makes the definitions much simpler to state and reason about.
- Separating the change inference process into a language dependent front-end and a language independent back-end makes it more easy to adapt our approach to other languages than C.

3.1 *Definition (Syntax of Terms)* $\textsc{Term} ::= \textsc{atom} \mid \textsc{atom}(\textsc{Term}^+)$

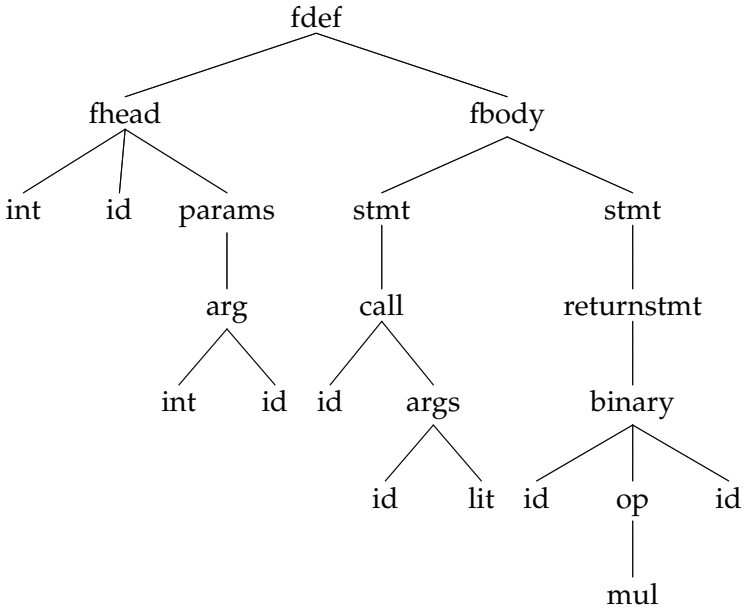
In this definition, and subsequently, t^+ indicates one or more comma-separated occurrences of the nonterminal t. Furthermore, terms will be written as a and $a(ts)$, for atomic and compound terms respectively. Intuitively, we represent a C AST by using the atom term to represent symbols of the grammar and $\textsc{atom}(ts)$ to represent trees labeled with a symbol represented by atom at the root and subtrees ts.

3.1.1 *Constructing* Terms

Given an AST obtained by parsing a C program, we wish to construct an isomorphic Term tree. We now discuss some of the issues considered in translating a C AST to a Term tree.

3.2 *Example (Translation of C code to* TERM*)* Below we present a simple C function and a simplified AST of the function. Nodes are labelled with their corresponding nonterminal or terminal symbol. Only leaf nodes are considered to have values, which correspond to the lexeme of the terminal represented at the node.

```
int foo(int x) {
  bar(x, 42);
  return x * x;
}
```



The result of translating the AST into a TERM is shown below.

```
fdef(
  fhead(int, id(x), params(arg(int, id(x)))),
  fbody(
    stmt(call, (id(bar),
      args( id(x), lit(42)))
    ),
    stmt(returnstmt(
      binary(id(x), op(*), id(x))
    )
  )
)
```

As can be seen from Example 3.2, terminal (leaf) nodes of the AST are not represented by an ATOM tree, but rather a compound $a(a')$ where a represents the terminal *type* (identifier, constant,

etc.) and a' represents the *lexeme* of the terminal. Since the C code and the corresponding TERM representation is isomorphic, we will present larger examples using the syntax of real C code, while examples that serve to illustrate points about definitions are given using their TERM representation. Also, we shall not go into any detail about the parser we have used to parse C programs as the parser used is the same as the one used in the Coccinelle project [48, 47].

3.2 TERM PATTERNS

Our approach for finding common changes in a set of changed programs is based on the assumption that there should be common characteristics about the locations that were modified. In particular, we find that it is the case that it is often possible to characterize the locations that have changed similarly by their structural commonality.

A term pattern is a TERM that may additionally contain *meta-variables*, which are placeholders for concrete terms. Re-using the concept of change vocabulary from Chapter 2, term patterns can be said to provide a *context-free denotation* of locations with *named subpart abstraction*. In the following we use the words term patterns and patterns interchangeably. The syntax of patterns is as follows:

3.3 *Definition (Syntax of Patterns)* $p ::= \text{ATOM} \mid \text{ATOM}(p^+) \mid \mathit{Meta}$

Where Meta denotes a set of meta-variables. In the examples, we use uppercase letters to denote meta-variables.

3.4 *Definition* *A sequence of patterns* p_1 *to* p_n *is denoted* $\langle p_1, \ldots, p_n \rangle$*. The empty sequence is denoted* $\langle\rangle$*.*

3.5 *Definition* *Pattern size The size of a pattern is the number of atomic patterns in it plus the number of constructors. The size of a pattern* p *is denoted* $|p|$*.*

$$\begin{aligned} |a| &= 1 \\ |a(t_1, \ldots, t_n)| &= 1 + |p_1| + \cdots + |p_n| \end{aligned}$$

We extend the pattern size function to sequences of patterns $\langle p_1, \ldots, p_n \rangle$ *by*

$$|\langle p_1, \ldots, p_n \rangle| = \Sigma_{i=1}^{n} |p_i|$$

3.6 *Definition (Environments)* *An environment is a deterministic function mapping meta-variables to patterns:* $\theta : p \to p$*. The domain of* θ *is the set of meta-variables that it maps.*

3.7 *Definition (Environment application)* *Applying the environment* θ *to a pattern yields a new pattern with meta-variables occurring in* p *from the domain of* θ *substituted.*

$$\begin{aligned} \theta X &= \begin{cases} t & \textit{if } X \mapsto t \in \theta \\ X & \textit{otherwise} \end{cases} \\ \theta a &= a \\ \theta a(p_1, ..., p_n) &= a(\theta p_1, ..., \theta p_n) \end{aligned}$$

Sometimes, we will need to do a reverse substitution. That is, given a term and an environment, we wish to substitute subterms of the term with the meta-variables binding the subterm in the environment.

3.8 Definition (Reverse substitution) *Given term* t *and environment* θ*, the pattern obtained by reverse substitution of* θ *to* t *is given as* $\theta^{-1}t$*.*

Reverse substitution is only well-defined when the environment maps distinct meta-variables to distinct terms.

A pattern p matches a term t if there exists a substitution θ of meta-variables for terms such that applying the substitution to the pattern yields a term that is syntactically equivalent to t, i.e., $\theta p = t$ where θp denotes application of θ to p.

3.9 Definition (Pattern match) *We denote that pattern* p *matches a term* t *evidenced by environment* θ *by* $p \trianglelefteq_\theta t$*:*

$$p \trianglelefteq_\theta t \iff \theta p = t$$

We sometimes omit the explicit reference to θ *and say that* p *matches* t*:*

$$p \trianglelefteq t \iff \exists \theta : \theta p = t$$

We extend the notation to sets of terms such that when a pattern matches all the terms in a set of terms ts we simply say that p *matches ts:*

$$p \trianglelefteq ts \iff \forall t \in ts : p \trianglelefteq t$$

We also define pattern matching for patterns with patterns reusing the notation and definitions from above.

A meta-variable may occur more than once in a pattern, in which case all occurrences of the meta-variable must match the same concrete term. For example, the pattern `f(X,X)` matches any concrete term that is a call to `f` with two syntactically equal arguments. We call a pattern which is simply a meta-variable a *trivial* pattern. We furthermore say that a pattern matches a set of terms if it matches all of the terms in the set.

3.10 Definition (Alpha equivalence of patterns) *Two patterns* p *and* p' *are said to be alpha equivalent if and only if they differ only in choice of names of meta-variables denoted* $p \sim p'$*. We first define an auxiliary relation* $\theta \vdash p \sim p'$ *and then let* $p \sim p'$ *denote* $\exists \theta : \theta \vdash p \sim p'$*.*

$$\frac{}{\theta \vdash a \sim a} \qquad \frac{X \mapsto X' \in \theta}{\theta \vdash X \sim X'} \qquad \frac{0 \leqslant i \leqslant n \quad \theta \vdash p_i \sim p'_i}{\theta \vdash a(p_1, ..., p_n) \sim a(p'_1, ..., p'_n)}$$

3.2.1 *Abstracting terms*

A problem that we will frequently run into is the problem of finding a pattern that matches a set of terms. The goal of this section is to define a computable mechanism with which we can find patterns matching a set of terms. We end up with a definition of a function that given a set of terms returns a unique pattern such that the pattern matches all of the terms. We denote the function applied to a set of terms as $*ts$ where ts is the set of terms. This section contains a number of definitions that are important to show the uniqueness of the pattern finding function and reveal in what sense the pattern is the "best" pattern, but the main point should be clear by Definition 3.12. Readers more interested in the core of our change inference approach can skip to Chapter 4 on a first reading.

In general, there may be more than one pattern matching a set *ts*. In particular the trivial pattern matches any set *ts*. Thus, we define a notion of a "most informative" pattern that matches the set of terms given.

MOST INFORMATIVE PATTERN Relative to a set of terms *ts*, the most informative pattern is intuitively a pattern that 1. matches *ts*, 2. retains information about which subterms are equal, and 3. abstracts as little as possible. Example 3.11 below shows gives an illustration of patterns that satisfy the properties mentioned above.

3.11 *Example (Pattern abstracting two terms)* Let two terms be given:

$$\begin{aligned} t_1 &= h(f(42), g(y), y) \\ t_2 &= h(f(42), g(x), x) \end{aligned}$$

The following three patterns all match $\{t_1, t_2\}$:

$$\begin{aligned} p_1 &= h(f(X), g(Y), Z) \\ p_2 &= h(f(X), g(Y), Y) \\ p_3 &= h(f(42), g(Y), Y) \end{aligned}$$

The first pattern only satisfy the first condition: $p_1 \trianglelefteq \{t_1, t_2\}$. The second pattern in addition satisfies the second condition that information about which subterms are equal should be retained in the pattern. The third pattern also satisfies the condition that as little as possible should be abstracted.

Concretely, we define the most informative pattern with respect to a set of terms as the most specific pattern matching the terms. The concept of the most specific pattern is similar to the concept of most specific generalization or anti-unifier first presented by Plotkin [50]. Our algorithm for computing most specific patterns, presented shortly, is similar in spirit to the one presented by Sœrensen and Glück [53].

3.12 *Definition Most specific pattern Given terms ts, a pattern* p *is the most specific pattern for ts if and only if:*

1. $p \trianglelefteq ts$ *(the pattern matches all terms)*
2. $\forall p' : p' \trianglelefteq ts \Rightarrow p' \trianglelefteq p$ *(other patterns are more general)*

We denote that pattern p *is the most specific pattern for a set of terms* ts *as* $msp(p, ts)$.

One can show that the most specific pattern for a set of terms is unique up to alpha equivalence.

3.13 *Theorem (Uniqueness of most specific pattern) Given terms* ts*, the set of patterns* $MSP(ts) = \{p \mid msp(p, ts)\}$ *are all alpha equivalent:*

$$\forall p, p' \in MSP(ts) : p \sim p'$$

3.14 Proof (Proof of Theorem 3.13) Given terms ts and two patterns p_1 and p_2 such that $msp(p_1, ts,)$ and $msp(p_2, ts,)$, we show that $p_1 \sim p_2$.

By the second clause of the definition of $msp(p, ts)$ we infer that $p_1 \trianglelefteq p_2$ and $p_2 \trianglelefteq p_1$. Unfolding the definition of pattern matching we get: a) $\exists\theta_1 : \theta_1 p_1 = p_2$ and b) $\exists\theta_2 : \theta_2 p_2 = p_1$.
We now proceed by induction on the structure of p_1 to prove that $\theta_1 p_1 = p_2 \wedge \theta_2 p_2 = p1 \Rightarrow p_1 \sim p_2$.

CASE: $p_1 \equiv a$. From a) and b) above, we get $p_2 = a$ and $a \sim a$ holds by definition.

CASE: $p_1 \equiv X$. From a) and b) above, we obtain $\theta_1 = \{X \mapsto p_2\}$ and from b) we have $\theta_2 p_2 = X$. Since the domain of substitutions is a set of meta-variables, we obtain $\theta_2 = \{X' \mapsto X\}$ and $p_2 = X'$. We can then conclude $X \sim X'$.

CASE: $p_1 \equiv a(p_1, ..., p_n)$. From a) and b) above, we obtain $p_2 = a(p_1, ..., p_n)$. By definition of substitution application we can infer $\theta_1 p_i = p_i'$ and $\theta_2 p_i' = p_i$ for $1 \leqslant i \leqslant n$. By the structural induction hypothesis we conclude $p_i \sim p_i'$ and by definition of alpha equivalence we conclude that $a(p_1, ..., p_n) \sim a(p_1', ..., p_n')$. ∎

COMPUTATION OF MOST SPECIFIC PATTERN Since the set of most specific patterns with respect to a set of terms consists of all alpha equivalent patterns, any one of those patterns represents the most specific pattern equally well. We can therefore define a deterministic function, that finds a most specific pattern relative to a set of terms. Below we define a function that takes two patterns and returns a most specific pattern for those two patterns. Later we extend the function to compute the most specific pattern for a set of patterns.

3.15 *Definition (MSP function)* *Given patterns* p_1 *and* p_2, $p_1 * p_2$ *is the most specific patten for* p_1 *and* p_2. *The definition of* $\cdot * \cdot$ *is given below:*

$$p_1 * p_2 = p_3 \text{ if } \mathit{fuse}(p_1, p_2, \varnothing) = (p_3, \sigma)$$

where

$$\mathit{fuse}(p, p', \sigma_0) = \begin{cases} (p, \sigma_0) & \text{if } p = p' \\ (a(p''_1, ..., p''_n), \sigma_n) & \text{if } p \neq p' \wedge p = a(p_1, ..., p'_n) \wedge \\ & \quad p' = a(p'_1, ..., p'_n) \wedge \\ & \quad \mathit{fuse}(p_i, p'_i, \sigma_{i-1}) = (p''_i, \sigma_i) \text{ for } 1 \leqslant i \leqslant n \\ (Z, \sigma_0) & \text{if } p \neq p' \wedge (Z \mapsto (p, p')) \in \sigma_0 \\ (X_{|\sigma|}, \sigma[X_{|\sigma|} \mapsto (p, p')]) & \text{otherwise} \end{cases}$$

In order to show that $p_1 * p_2 = p_3$ computes a most specific pattern relative to the patterns p_1 and p_2 we must show that $p_3 \trianglelefteq \{p_1, p_2\}$ and $\forall p : p \trianglelefteq \{p_1, p_2\} \Rightarrow p \trianglelefteq p_3$.

3.16 Proof ($\mathit{msp}(p, p')$ computes a matching pattern) We show by induction on the derivation of $\mathit{fuse}(p, p', \sigma) = (p'', \sigma')$ that $p'' \trianglelefteq \{p, p'\}$ for any σ. $\mathit{msp}(p, p') \trianglelefteq \{p, p'\}$ then follows directly. There are four cases to consider.

$\mathit{fuse}(p, p', \sigma) = (p, \sigma)$ (THE FIRST CLAUSE IN *fuse*): From the definition of *fuse* we know that $p = p'$ and that $p'' = p = p'$. We conclude that $p'' \trianglelefteq \{p, p'\}$.

$\mathit{fuse}(p, p', \sigma) = (a(p''_1, ..., p''_n), \sigma_n)$ (THE SECOND CLAUSE IN *fuse*): We infer that $p = a(p_1, ..., p_n)$ and $p' = a(p'_1, ..., p'_n)$. Using the inductive hypothesis, we show that $p''_i \trianglelefteq p_i$ and $p''_i \trianglelefteq p'_i$ for $1 \leqslant i \leqslant n$. It follows that $a(p''_1, ..., p''_n) \trianglelefteq \{a(p_1, ..., p_n), a(p'_1, ..., p'_n)\}$ as required.

$\mathit{fuse}(p, p', \sigma) = (X, \sigma)$ (THE THIRD AND FOURTH CLAUSE IN *fuse*): It follows directly that $X \trianglelefteq \{p, p'\}$ for any meta-variable X. ∎

3.17 Proof ($\mathit{msp}(p, p')$ is most specific) We show that $\mathit{fuse}(p, p', \sigma) = (p'', \sigma') \Rightarrow \forall p_a : p_a \trianglelefteq p''$ for any σ by induction on the derivation of $\mathit{fuse}(p, p', \sigma) = (p'', \sigma)$. It then follows that $\forall p_a : p_a \trianglelefteq \mathit{msp}(p, p')$.

There are four cases to consider.

$\mathit{fuse}(p, p', \sigma) = (p, \sigma)$ (THE FIRST CLAUSE IN *fuse*): Since $p = p'$ and $p'' = p = p'$ by definition of *fuse* it follows that $\forall p_a : p_a \trianglelefteq \{p, p'\} \Rightarrow p_a \trianglelefteq p''$.

$\mathit{fuse}(p, p', \sigma) = (a(p''_1, ..., p''_n), \sigma_n)$ (THE SECOND CLAUSE IN *fuse*): We infer that $p = a(p_1, ..., p_n)$ and $p' = a(p'_1, ..., p'_n)$. From this we can see that $\forall p_a : p_a \trianglelefteq \{p, p'\} \Rightarrow p = X \vee p = a(p_{a,1}, ..., p_{a,n})$ for some meta-variable X. If $p_a = X$ we can conclude $X \trianglelefteq a(p''_1, ..., p''_n)$ as required. If $p_a = a(p_{a,1}, ..., p_{a,n})$ we need to show that (*) $p_{a,i} \trianglelefteq p''_i$ for $1 \leqslant i \leqslant n$ because then we can conclude $a(p_{a,1}, ..., p_{a,n}) \trianglelefteq a(p''_1, ..., p''_n)$. We can show (*) using the inductive hypothesis.

$\mathit{fuse}(p, p', \sigma) = (X, \sigma)$ (THE THIRD AND FOURTH CLAUSE IN *fuse*): We know that $p \neq p'$ and that $\neg(p = a(p_1, ..., p_n) \wedge p' = a(p'_1, ..., p'_n))$. By case analysis on the structure of p and p' one can show that $\forall p_a : p_a \trianglelefteq \{p, p'\} = p_a = Y$ for some meta-variable Y. It follows that $Y \trianglelefteq X$ as required. ∎

3.18 *Theorem ($p * p'$ is the most specific pattern for p and p')* *Given patterns p and p' the pattern computed by $p * p'$ is alpha-equivalent to a most-specific pattern according to Definition 3.12.*

$$p * p' = p'' \Rightarrow \forall p''' : msp(p''', \{p, p'\}) \Rightarrow p'' \sim p'''$$

3.19 Proof (Proof of Theorem 3.18) The theorem follows from Proofs 3.16 and 3.17. ∎

We can extend the $\cdot * \cdot$ function to sets of patterns with the same properties as for pairs of patterns, denoted $*ps$ where ps is a set of patterns; We wish to obtain a definition of $*ps$ such that 1. $p \trianglelefteq ps$ and 2. $\forall p' : p' \trianglelefteq p$.

Define $Q(ps)$ as $Q(ps) = \{p \mid ps' \subseteq ps \wedge p \trianglelefteq ps'\}$ i.e. $Q(ps)$ is the set of all patterns that match any subset of ps. We can then define an ordering of the patterns in $Q(ps)$ as $p \sqsubseteq p' \iff p' \trianglelefteq p$. We show that $(\sqsubseteq, Q(ps))$ is a partially ordered set by showing that $\cdot \sqsubseteq \cdot$ is reflexive, anti-symmetric, and transitive.

(Reflexivity) $p \sqsubseteq p$ follows from the first clause of the definition of *fuse*.

(Anti-symmetry) $p \sqsubseteq p' \wedge p' \sqsubseteq p' \Rightarrow p \sim p'$. This was already proved in the latter part of Proof 3.14.

(Transitivity) $p \sqsubseteq p' \wedge p' \sqsubseteq p'' \Rightarrow p \sqsubseteq p''$. We unfold the definition of $\sqsubseteq$ to get $\exists \sigma, \sigma' : \sigma p' = p \wedge \sigma' p'' = p'$. Thus, we can compose σ and σ' to obtain a σ'' such that $\sigma'' p'' = p$ by $\sigma(\sigma' p'') = \sigma p' = p$.

When $p \sqsubseteq p'$ holds, p is more specific than p' (or p' is more general than p). The least upper bound of a pair of patterns from $Q(ps)$ is given by $p \sqcup p'$. Thus, when $p \sqcup p' = p''$ it should hold that $p \sqsubseteq p'' \wedge p' \sqsubseteq p'' \wedge \forall p_a : p \sqsubseteq p_a \wedge p' \sqsubseteq p_a \Rightarrow p'' \sqsubseteq p_a$. By Theorem 3.18 we can set $p \sqcup p' = p * p'$. Thus, the least upper bound of a set ps such that $ps \subseteq Q(ps)$ is given by $\bigsqcup ps$ which means that we can compute it by $p_1 * \ldots * p_n$ if $ps = \{p_1, \ldots, p_n\}$.

3.20 *Definition (Set-extension of $\cdot * \cdot$)* *Let ps be a set of patterns $ps = \{p_1, \ldots, p_n\}$. We denote least upper bound or ps with respect to the ordering given above as $*ps$. Above, we have already shown that $*ps$ has the properties of being the most specific pattern for ps.*

TRANSFORMATION PARTS

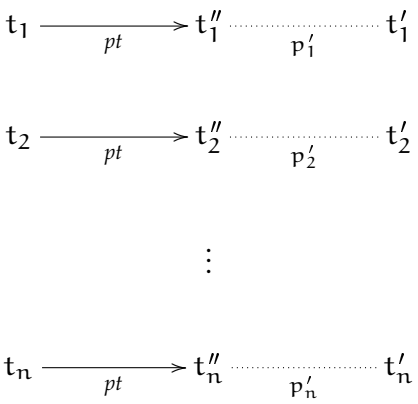
Chapter 4

In this Chapter we present an abstract description of the common change inference problem outlined in Section 1.1. Concretely, we define a concept of a *transformation part* that captures when one transformation is part of another. We present the defintions in terms of the TERM language described earlier.

4.1 PROPERTIES OF COMMON CHANGE DESCRIPTIONS

In this section we revisit the properties mentioned in Section 1.1 that should hold of the change descriptions that we infer.

A set of pairs of terms $\{(t_1, t_1'), \ldots, (t_n, t_n')\}$ that represents a set of original and updated programs is called a *changeset* and is denoted *CS*. Recall the right part of Figure 1 repeated below for ease of reference.



The picture is meant to illustrate the common change inference problem by indicating that for each program t_i, a common change *pt* is applied as part of turning t_i into t_i'. Stated differently: Given a changeset *CS* we are looking for a change description (we will henceforth use the simpler term *program transformation* instead) *pt* such that:

$$\forall (t_i, t_i') \in CS \exists t_i'' : [\![pt]\!](t_i) = t_i''$$

and the inferred program transformation *pt* should satisfy the following two properties:

A. The changes performed by *pt* on each t_i should be part of turning t_i into t_i' and,

B. *pt* should express as much as possible of the common changes applied in the changeset.

Our problem is now to figure out what these two properties actually mean. In the following section we investigate a number of possibilities before presenting the one that turned out to be most useful. Section 4.1.1 discusses two attempts at defining the meaning of $pt \preceq (t, t')$. In the end, both attempts are rejected for an third definition, however the discussion of the attempts leads to the definition of the meaning of $pt \preceq (t, t')$ that turned out to be useful. Readers more interested in our actual definition can skip to Section 4.2 on a first reading.

4.1.1 *Towards a definition*

We start our investigation of how to formalize properties A and B from above by fixing some notation and defining the composition of program transformations. Specifically, let $pt \preceq (t, t')$ denote that the program transformation pt is part of turning t into t'. The composition of two program transformations pt and pt' is denoted $pt' \circ pt$ with the meaning $[\![(pt' \circ pt)]\!](t) = [\![pt']\!]([\![pt]\!](t))$.

COMPOSABILITY Our first way to approach the problem of the meaning of $pt \preceq (t, t')$ is by saying that pt should be considered part of (t, t') if the transformation of t into t' can be split into two parts one of which is pt.

$$pt \preceq (t, t') \iff \exists pt' : [\![pt \circ pt']\!](t) = t' \vee [\![pt' \circ pt]\!](t) = t'$$

The biggest problem with this definition is that it potentially admits *any* program transformation to be part of any pair of terms. E.g. suppose $[\![pt]\!](t) = t''$ and $t'' \neq t'$. We can now let pt' behave such that $\forall t'' : [\![pt']\!](t'') = t'$ which in turn satisfies the requirement that $[\![pt' \circ pt]\!](t) = t'$. Therefore, we can conclude that $\forall pt, t, t' : pt \preceq (t, t')$.

COMMUTATIVITY The problem in the previous definition is that we allow the existentially quantified program transformation pt' to "undo" some of the changes that application of pt have already performed. We can try to overcome the problem by strengthening the requirement on pt' by requiring that pt commutes with pt'. The intuition is that commutativity of pt and pt' implies that the two transformations are independent somehow. Alternatively, we could explicitly state in the definition of $\preceq$ that their domains should be disjoint. However, commutativity can be specified without knowledge of the underlying transformation language. We therefore obtain the following definition:

$$pt \preceq (t, t') \iff \exists pt' : [\![pt' \circ pt]\!](t) = [\![pt \circ pt']\!](t) = t'$$

The previous problem where any pt can be part of any pair (t, t') is less evident now, because the existentially quantified pt' needs to commute with pt. Example 4.1 shows an example of a program transformation that is *not* admitted by the commutative definition.

4.1 Example (Non-commuting transformation) Let the following three terms be given:

$$\begin{aligned} t_1 &= f(a, b, c) \\ t_2 &= q \\ t_3 &= f(a', c) \end{aligned}$$

Assume $[\![pt]\!](t_1) = t_2$ and for all other t, pt behaves as the identity function. We now wish to decide whether $pt \preceq (t_1, t_3)$. To do so, we need to find a pt' that commutes with pt. We could let $[\![pt']\!](t_2) = t_3$ but we also need to specify the result of $[\![pt']\!](t_1)$. However, if $[\![pt']\!](t_1) = t_4$ and $t_4 \neg t_1$ then $[\![pt]\!](t_4) = t_4$ which is not equal to the "goal" term t_3. On the other hand, if $t_4 = t_1$ then $[\![pt]\!](t_4) = t_2$ and t_2 is also not the goal term t_3. The conclusion is that there can be no commuting transformation pt' and thus it is *not* the case that $pt \preceq (t_1, t_3)$.

The commutative definition of $\preceq$ is more restrictive than the previous one, but there is still a problem. Suppose the underlying transformation language allows specification of transformations that depend on the input in the following sense: when some pt is applied to t_a it returns t_1 and if pt is applied to t_b it returns t_2. For any pair (t, t') it seems reasonable that one can specify two transformations pt_a and pt_b such that $[\![pt_a]\!](t) = t_a$ and $[\![pt_b]\!](t) = t_b$ and furthermore $[\![pt_b]\!](t_a) = t'$ and $[\![pt_a]\!](t_b) = t'$. In terms of Example 4.1, we would be able to decide that $pt \preceq (t_1, t_3)$ if pt was such a dependent transformation. The problem is that it is not at all obvious that such a pt should be considered part of turning t into t' even though the underlying transformation language allows it. A variant of this problem goes in the opposite direction: there might be a pt for which it seems perfectly obvious that it should be considered part of some pair (t, t'), but the underlying transformation language may not be expressive enough to capture this. Example 4.2 shows such a situation.

4.2 Example Let the following terms be given:

$$\begin{aligned} t_1 &= f(a, b) \\ t_2 &= f(b, b) \\ t_3 &= f(b, a) \end{aligned}$$

Assume that the underlying transformation language can only express "one-shot" term rewrites with no abstraction mechanisms. Let rewrite specifications be denoted $p \rightsquigarrow p'$ with the meaning: when applying the specification to a term, rewrite all subterms matching p into p'. Let $pt = a \rightsquigarrow b$. Thus, $[\![pt]\!](t_1) = t_2$. It would seem reasonable to assume that pt should be considered part of (t, t'); what remains is simply to replace the second b with a. Thus, we need to find a pt' such that pt commutes with pt'. We could define $pt' = t_2 \rightsquigarrow t_3$, but it can be seen that this does not commute with pt. Alternatively, we could define $pt' = b \rightsquigarrow a$ which implies $[\![pt']\!](t_2) = f(a, a)$ which is not equal to t_3. In fact, one can show that there is no pt' with which pt commutes in the sense that $[\![pt' \circ pt]\!](t_1) = [\![pt \circ pt']\!](t_1) = t_3$.

The reason the first definition of the meaning of $pt \preceq (t, t')$ is rejected is because it can admit *any pt*, while the latter may not admit reasonable ones because of the underlying transformation language may not be expressive enough. Finally, both of the definitions are not constructive in the sense that they can be decided only if we are able to find a certain existentially quantified program transformation.

The reason the first definition admits any transformation as part of any pair (t, t') is that the existentially quantified program transformation pt' is allowed to "undo" changes already made by pt. We tried to overcome this, by requiring commutativity of pt and pt' with respect to (t, t'), but commutativity brings its own problems to the definition. Instead, we could try to formulate the no-undoing property directly. This is the goal of the following section.

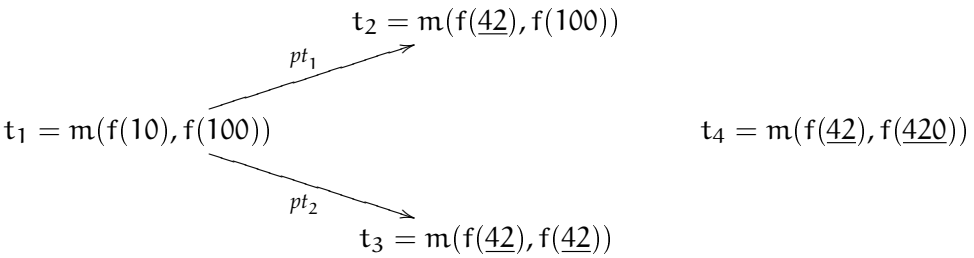
4.2 TREE DISTANCE BASED TRANSFORMATION PARTS

In this section we state the final definition of the meaning of $pt \preceq (t, t')$. The intuitive meaning is "pt should be considered part of turning t into t' when the changes pt makes when applied to t does not need to be undone again in order to reach t'". We motivate the definition in terms of an example.

MOTIVATING EXAMPLE Consider the following four terms:

$$\begin{aligned} t_1 &= m(f(10), f(100)) \\ t_2 &= m(f(42), f(100)) \\ t_3 &= m(f(42), f(42)) \\ t_4 &= m(f(42), f(420)) \end{aligned}$$

Suppose now, that we are given two program transformations pt_1 and pt_2 and we wish to decide whether $pt_i \preceq (t_1, t_4)$. The diagram below illustrates the application of pt_i to t_1. Underlined subterms mark subterms that have changed with respect to t_1. The arrows from t_1 $(m(f(10), f(100)))$ are labelled with the program transformation that needs has been applied.



From the diagram it evident that in order to transform the top-most term t_2 we simply need to modify the occurrence of 100 into 420 in order to obtain t_4. However, to reach t_4 from t_3 we need to modify the second occurrence of 42 which is already changed as can be seen from the underlining. We conclude that pt_1 should be considered part of (t_1, t_4) while pt_2 should not.

4.2.1 *Work-function*

We can capture the no-undoing property using a "work-function" that formalizes the changes performed between two terms. For any t_a and t_b, suppose $\mathcal{W}(t_a, t_b)$ denotes the work required to turn t_a into t_b somehow.

Given a *pt* and (t, t') such that $[\![pt]\!](t) = t''$ then either $t'' = t'$ or $t'' \neq t'$. If $t'' = t'$ it is the case that no more work should be required to turn t'' into t':

$$\mathcal{W}(t, t'') = \mathcal{W}(t, t')$$

If $t'' \neq t'$ there is still some work to do in order to reach t'. It may be that *pt* made a change that needs to be undone in order to reach t' from t''. Thus, it must be the case that *more* work is done when first changing t into t'' and then changing t'' into t' than the work done by changing t directly into t':

$$\mathcal{W}(t, t') \leqslant \mathcal{W}(t, t'') \oplus \mathcal{W}(t'', t')$$

On the other hand, if *pt* did *not* make a change that needs to be undone, the work remaining from t'' to t' should be exactly the work between t and t' that *pt* did not perform.

$$\mathcal{W}(t, t'') \oplus \mathcal{W}(t'', t') = \mathcal{W}(t, t')$$

From the discussion above, we can define the no-undoing property using the work-function as

4.3 *Definition (Transformation part) The work required to turn* t *into* t'' *and subsequently turning* t'' *into* t' *should be exactly the same as the work required just transforming* t *into* t'*.*

$$pt \preceq (t, t') \iff [\![pt]\!](t) = t'' \wedge \mathcal{W}(t, t'') \oplus \mathcal{W}(t'', t') = \mathcal{W}(t, t')$$

We next turn our attention towards finding a concrete work-function with which to instantiate the above definition.

4.2.2 *Term-distance*

We can now define a concrete version of the $\mathcal{W}(t, t')$ function. Specifically, we define a term distance function on terms which is a metric function. Being a metric directly implies that the term-distance function satisfies the requirements of the work-function stated above. We denote the term distance between terms t and t' by $\delta(t, t')$.

4.4 *Definition (Term distance)* *The term-distance function takes two terms as input and returns a number representing the number of primitive term operations (insertion, deletion) needed to transform one into the other.*

$$\delta(a, a') = \begin{cases} 0 & \textit{if } a = a' \\ 1 & \textit{otherwise} \end{cases}$$

$$\delta(a, c(ts)) = \delta(c(ts), a) = |a| + |c(ts)|$$

$$\delta(a_1(ts_1), a_2(ts_2)) = \begin{cases} \Delta(ts_1, ts_2) & \textit{if } a_1 = a_2 \\ 1 + \Delta(ts_1, ts_2) & \textit{otherwise} \end{cases}$$

$$\Delta((t : ts), (t' : ts')) = \min \left\{ \begin{array}{l} \delta(t, t') + \Delta(ts, ts'), \\ \Delta((t : ts), ts') + |t'|, \\ \Delta(ts, (t' : ts')) + |t| \end{array} \right\}$$

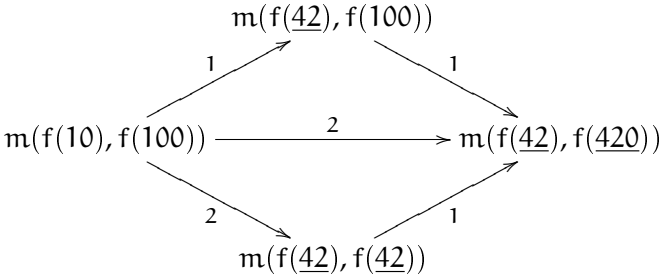
$$\Delta(\epsilon, ts) = \Delta(ts, \epsilon) = |ts|$$

The definition of $\delta(t, t')$ makes use of a distance function on sequences of terms, $\Delta(ts_1, ts_2)$. The distance between two sequences is found using a dynamic programming scheme. Overall, the distance function is comparable to the one used by Yang [62]. The main difference is that we have fixed all constants to 1 whereas Yang allow user-specified weights.

4.5 *Example* Recall the four terms from the motivating example above.

$$\begin{aligned} t_1 &= m(f(10), f(100)) \\ t_2 &= m(f(42), f(100)) \\ t_3 &= m(f(42), f(42)) \\ t_4 &= m(f(42), f(420)) \end{aligned}$$

Some term distances are shown in the diagram below. Like in the previous diagram, subterms that have changed with respect to t_1 (left-most) have been underlined.



We see that $\delta(t_1, t_2) + \delta(t_2, t_4) = \delta(t_1, t_4)$ but also that $\delta(t_1, t_3) + \delta(t_3, t_4) > \delta(t_1, t_4)$ supporting the intuition that a program transformation turning t_1 into t_3 should not be considered part of (t_1, t_4).

The definition of when a program transformation is part of a pair of terms can now be state simply as follows.

4.6 *Definition (Term-distance based transformation part)* *Given pt and* (t, t'), *pt is called a transformation part of* (t, t') *if and only if:*

$$[\![pt]\!](t) = t'' \wedge \delta(t, t'') + \delta(t'', t') = \delta(t, t')$$

Although strictly it is redundant, we will sometimes say that a *pt* which satisfies Definition 4.6 is a *safe* transformation part of simply safe for (t, t'). We add the term "safe" to indicate that when applying *pt* to t nothing will be changed in t that should not have been changed with respect to t'. Definition 4.6 provides a meaning to the first property (A) mentioned in Section 4.1. In the next section we define what it means for a program transformation to express the maximal amount of common changes.

4.3 SUBSUMPTION OF PROGRAM TRANSFORMATIONS

Recall the second (B) of the two properties from Section 4.1 stating the following about the common changes, *pt* inferred from a changeset *CS*

> *pt* should express as much as possible of the common changes applied in the changeset.

We formalize the notion of expressing more changes by giving a subsumption relation on program transformations. While a safe transformation part describes the relationship between a program transformation and a pair of terms (t, t'), program transformation subsumption describes the relationship between one program transformation and another. Intuitively, given a pair of terms (t, t'), a program transformation is subsumed by another if the former performs a safe transformation part of the latter, relative to transforming t into t'. The subsumption relation is defined in Definition 4.7 below.

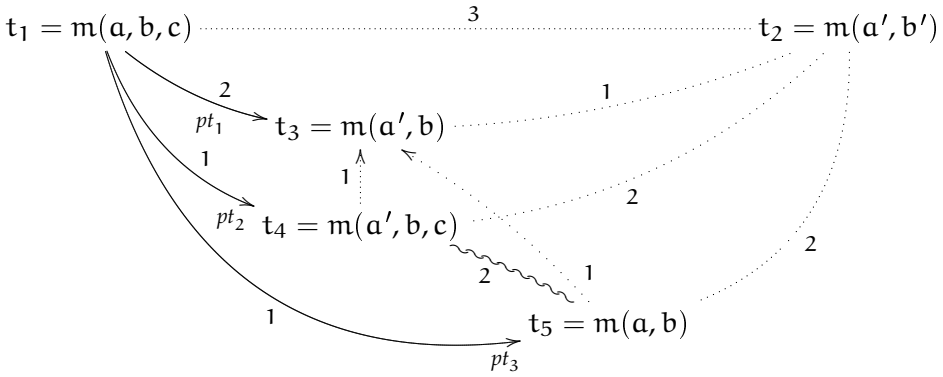
4.7 *Definition (Subsumption relation)* *Let pt and* pt' *be two program transformations. We say that pt is a sub-transformation of* pt' *relative to a pair of terms* (t, t') *denoted* $pt \preceq_{(t,t')} pt'$ *if and only if*

1. $pt' \preceq (t, t')$
2. $[\![pt']\!](t) = t''$
3. $pt \preceq (t, t'')$

4.8 Example Let the following terms be given:

$$\begin{aligned} t_1 &= m(a,b,c) \\ t_2 &= m(a',b') \\ t_3 &= m(a',b) \\ t_4 &= m(a',b,c) \\ t_5 &= m(a,b) \end{aligned}$$

Also, let there be three program transformations pt_1, pt_2, and pt_3 such that $[\![pt_1]\!](t_1) = t_3$, $[\![pt_2]\!](t_1) = t_4$, $[\![pt_3]\!](t_1) = t_5$ as indicated by the arrows in the diagram below. The numbers on the arcs in the diagram indicate term distances. Solid arcs represents that a program transformation (also shown on the arc) was responsible for the change. Dotted or curly arcs are labelled with the term distance between the connected terms.



The top-most dotted arc is labelled with 3, thus any other path from t_1 to t_2 that has a summarized cost of 3 is somehow indicative of a transformation part or subsumption relation. Using the diagram, one can therefore verify that, for $i \in \{1,2,3\}$, $pt_i \preceq (t_1, t_2)$. It can also be seen that $pt_3 \preceq_{(t_1,t_2)} pt_1$ and $pt_2 \preceq_{(t_1,t_2)} pt_1$ by the paths $m(a,b,c) \to m(a,b) \to m(a',b) \to m(a',b')$ and $m(a,b,c) \to m(a',b,c) \to m(a',b) \to m(a',b')$ respectively. However, we can also see that it is *not* the case that $pt_3 \preceq_{(t_1,t_2)} pt_2$. This is evident from the cost of the path $m(a,b,c) \to m(a,b) \to m(a',b,c) \to m(a',b')$ which equals 5 but should equal 3.

4.4 EXTENDING TO CHANGESETS

Both the transformation part relation and the subsumption relation are defined relative to a single pair of terms (t,t'). We can extend them straightforwardly to a changeset by quantifying over all pairs of terms in the changeset:

4.9 Definition (Common transformation part) Given program transformation pt and a changeset CS, the transformation part relation is extended to CS by the following. We denote the extension $pt \preceq CS$.

$$pt \preceq CS \iff \forall (t,t') \in CS : pt \preceq (t,t')$$

When $pt \preceq (t, t')$ holds we also say that pt is a common part of CS or that pt is a safe for CS.

4.10 *Definition (Changeset subsumption)* *Given program transformations pt and pt and a changeset CS, the program transformation relation is extended to a changeset by the following. We denote the extension as* $pt \preceq_{CS} pt'$

$$pt \preceq_{CS} pt' \iff \forall (t, t') \in CS : pt \preceq_{(t,t')} pt'$$

We can now state a formal interpretation of the maximality property B as a program transformation for which there exists no larger program transformation according to the subsumption relation. We call such a program transformation a maximal common transformation (relative to a given changeset).

4.11 *Definition (Maximal common transformations)* *Given a changeset CS the set of maximal common transformations, denoted* $LCT(CS)$ *is the following set:*

$$LCT(CS) = \{pt \mid pt \preceq CS \wedge \forall pt' : pt' \preceq CS \Rightarrow pt' \preceq_{CS} pt\}$$

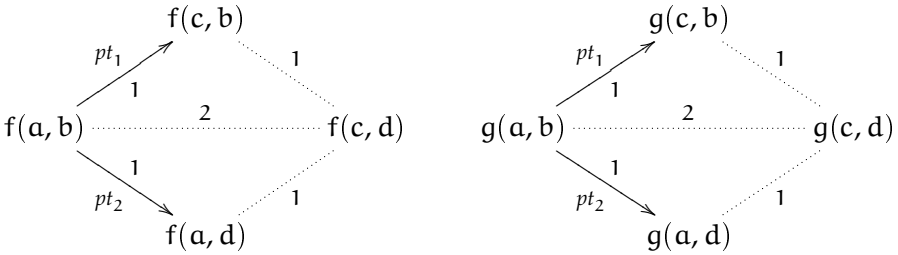
There is a problem with the above definition in that depending on the transformation language, there could be two program transformations $pt_1 \preceq CS$ and $pt_2 \preceq CS$ such that neither of $pt_1 \preceq_{CS} pt_2$ or $pt_2 \preceq_{CS} pt_1$ holds and at there same time there is no third pt_3 which subsumes both pt_1 and pt_2. Example 4.12 shows such a situation. An alternative definition replaces the implication of Definition 4.11 with the following.

$$\forall pt' : pt' \preceq CS \Rightarrow \neg(pt \preceq_{CS} pt') \vee \forall (t, t') \in CS : [\![pt]\!](t) = [\![pt']\!](t)$$

4.12 *Example* Assume the transformation parts concept is instantiated with the transformation language of Example 4.2. Let the following terms be given:

$$\begin{aligned} t_1 &= f(a, b) \\ t_1' &= f(c, d) \\ t_2 &= g(a, b) \\ t_2' &= g(c, d) \end{aligned}$$

Let $CS = \{(t_1, t_1'), (t_2, t_2')\}$ and define $pt_1 = a \rightsquigarrow c$ and $pt_2 = b \rightsquigarrow d$. The below diagram illustrates the application of the two transformations to both t_1 and t_2.



By using the diagram, one can verify that $pt_1 \preceq CS$ and $pt_2 \preceq CS$. However there is no way to construct a simple term-rewrite rule that correctly updates *both* t_1 and t_2 and consequently there

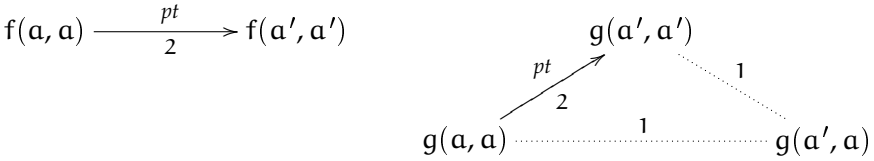
is no pt_3 such that it subsumes both pt_1 and pt_2. Using the direct formulation of maximality given in Definition 4.11, we obtain $LCT(CS) = \varnothing$. Using the indirect definition we get $LCT(CS) = \{pt_1, pt_2\}$.

Although the problem solved using the indirect definition of $LCT(CS)$ illustrate in Example 4.12 there can still be cases where the set $LCT(CS)$ is empty though. For example when the underlying transformation language is not expressive enough to capture the common characteristics of the locations in which the changes are to be made. Another cause could simply be that the changeset is "buggy". In Section 4.5 we therefore present an approach that address both of the mentioned problems.

4.13 *Example ("Buggy" changeset)* Assume the following four terms are given and that the transformation language used is the same as in Example 4.12. We let $CS = \{(t_1, t_1'), (t_2, t_2')\}$.

$$\begin{aligned} t_1 &= f(a, a) \\ t_1' &= f(a', a') \\ t_2 &= g(a, a) \\ t_2' &= g(a', a) \end{aligned}$$

One can quickly verify that the set of maximal common program transformations is empty. However, consider the program transformation $pt = a \rightsquigarrow a'$. The two diagrams below illustrate application of pt to t_1 and t_2.



Although pt is *not* safe for CS it is safe for (t_1, t_1'). The reason pt is not safe for (t_2, t_2') is that it changes all occurrences of a into a' whereas, according to t_2', only one occurrence should have been modified. Since all other occurrences of a in t_1 and t_2 was changed, one could wonder whether this particular a should not *also* have been changed. Maybe the developer forgot to update this last occurrence of a. In Section 7.1 we will see cases where precisely this has happened when a developer performed the collateral evolution of a number of files by hand.

4.5 NON-GLOBAL COMMON CHANGES

The ideal case with respect to computing $LCT(CS)$ is when the same changes were performed in all of the original programs and the underlying transformation language is able to express them, because then the program transformations in the set $LCT(CS)$ are precisely all of those common changes. However, it may be the case that some changes were performed only in a subset of the

original programs. The set of maximal common program transformations, $LCT(CS)$ does not capture any of those as is illustrated in Example 4.14 below.

In this section we therefore relax the definition of transformation parts extended to changesets. Specifically, we modify the requirement of the definition of $pt \preceq CS$ to not require that pt is safe *for all* pairs of terms in CS.

4.14 *Example (Non-global common changes)* Assume we extend the transformation language from previous examples to additionally including a composition operator denoted $pt; pt'$. The interpretation of the program transformation $pt; pt'$ is then that $[\![pt; pt']\!](t) = [\![pt']\!]([\![pt]\!](t))$. Let the following terms be given:

$$\begin{array}{lllllll} t_1 & = & f(a,b,c) & t_1' & = & f(aa,bb,c) \\ t_2 & = & h(a,b,c) & t_2' & = & h(aa,bb,cc) \\ t_3 & = & g(a,b,c) & t_3' & = & g(aa,b,cc) \end{array}$$

Let $CS = \bigcup_{i=1}^{3}(t_i, t_i')$. One can now verify that the set of maximal changes is given by the singleton set $LCT(CS) = \{a \rightsquigarrow aa\}$. However, if we restrict CS to the subset $CS' = \bigcup_{i=1}^{2}(t_i, t_i')$ we would get $LCT(CS) = \{a \rightsquigarrow aa; b \rightsquigarrow bb\}$. On the other hand if CS is restricted to the subset $CS'' = \bigcup_{i=2}^{3}(t_i, t_i')$ we get $LCT(CS'') = \{a \rightsquigarrow aa; c \rightsquigarrow cc\}$.

THRESHOLDS The idea is to introduce a user-specified threshold to specify for how many pairs of terms in the changeset, the program transformation must be safe for. Let n be the threshold. Initially, one could conceive the following definition of safety relative to a threshold:

$$pt \preceq^n CS : \exists CS' \subseteq CS : |CS'| \geqslant n \wedge pt \preceq CS'$$

The definition of safety relative to a threshold makes use of the previous definition *without* thresholds. The somewhat subtle notational difference is in the subscripted n. The problem with this definition is that it can classify a program transformation as safe for a changeset even though it is actually "unsafe" for some of the pairs. I.e. there can be a $(t, t') \in CS$ such that $(t, t') \notin CS'$ and $pt \not\preceq (t, t')$. Instead, we wish that all program transformations $pt \in CS \backslash CS'$ are not unsafe for any pair of terms in that set. We have not explicitly defined a notion of unsafety, but intuitively unsafety means that the program transformation performed some changes that it should not have with respect to a pair of terms (t, t'). Consider the *negation* of a transformation part (without threshold) of a pair of terms (t, t'):

$$\begin{array}{rcl} \neg(pt \preceq (t,t')) & \iff & \neg(\exists t'' : [\![pt]\!](t) = t'' \wedge \delta(t,t'') + \delta(t'',t') = \delta(t,t')) \\ & \iff & \nexists t'' : [\![pt]\!](t) = t'' \vee \exists t'' : [\![pt]\!](t) = t'' \wedge \delta(t,t'') + \delta(t'',t') \neq \delta(t,t') \end{array}$$

When program transformation is *not* part of a pair of terms there can be two reasons: either the application of the program transformation failed or the changes made by the program transformation were not compatible with the changes needed. The first part of the disjunction fits very well with what not unsafe should mean: the transformation does not "do" anything wrong, because it does not do anything. We therefore get the following definition of a safe transformation part relative to a given threshold:

4.15 *Definition (Transformation part with threshold)* *Given program transformation pt, a changeset CS, and a threshold n, pt is said to be a safe transformation part of the changeset, denoted* $pt \preceq^{n} CS$ *when the following is satisfied:*

$$pt \preceq^{n} CS \iff CS' = \{(t,t') \mid pt \preceq (t,t') \wedge |CS'| \geqslant n \wedge pt \preceq CS' \wedge \forall (t,t') \in CS \backslash CS' : \nexists t'' : [\![pt]\!](t) = t''$$

The definition of the subsumption relation can also be stated relative to a threshold similarly

4.16 *Definition (Changeset subsumption with threshold)* *The supsumption relation on program transformations relative to a threshold is denoted* $pt \preceq^{n}_{CS} pt'$

$$\begin{aligned} pt \preceq^{n}_{CS} pt' \iff\ & CS_r = \{(t,t') \in CS \mid pt' \preceq (t,t')\} \wedge \\ & CS_l = \{(t,t') \in CS \mid pt \preceq (t,t')\} \wedge \\ & CS_r \subseteq CS_l \wedge |CS_r| >= n \wedge \\ & pt \preceq_{CS_r} pt' \wedge \\ & \nexists t'' : \forall (t,t') \in CS \backslash CS_l : [\![pt]\!](t) = t'' \wedge \\ & \nexists t'' : \forall (t,t') \in CS \backslash CS_r : [\![pt']\!](t) = t'' \end{aligned}$$

The definition may look a little intimidating at first. The essence is that the subsumption relation specifies that everywhere pt' is safe, pt performs only part of the changes performed by pt'. Specifically, one can derive the following property:

4.17 Property (Safety of subsumed transformations) Let a changeset CS and two program transformations pt and pt' be given. From Definition 4.16 we derive:

$$pt \preceq^{n} CS \wedge pt' \preceq^{n}_{CS} pt \Rightarrow pt' \preceq^{n} CS$$ ∎

The converse variant of the above property states that if a program transformation is not safe for a changeset, then there is no subsuming transformation which is also safe for the changeset.

Finally, the set of maximal common changes $LCT(CS)$ relative to a threshold is defined below.

$$LCT(CS,n) = \left\{ pt \mid pt \preceq^{n} CS \forall pt' \preceq^{n} CS : \begin{array}{l} CS_l = \{(t,t') \in CS \mid pt \preceq^{n} CS\} \wedge \\ CS_r = \{(t,t') \in CS \mid pt' \preceq^{n} CS\} \wedge \\ CS_l \subseteq CS_r \end{array} \Rightarrow pt' \preceq^{n}_{CS} pt \right\}$$

One can verify that $LCT(CS,|CS|) = LCT(CS)$. In the following, when we omit the threshold we assume a threshold set to the size of the changeset; e.g. $LCT(CS)$ should be read as $LCT(CS,|CS|)$.

In the next part we describe two algorithms that attempts to find the set $LCT(CS,n)$ in terms of two different transformation languages. The first transformation language is a simple term rewrite language similar to what have been used above, except it additionally includes an abstraction mechanism for subterms and allows several rewrites to be specified in one transformation. The second transformation language is a simple version of the SmPL language that is provided by Coccinelle. The transformation language extends the previous language with a context-sensitive denotation of locations to update.

Part II

ALGORITHMS AND IMPLEMENTATION

Chapter 5 CONTEXT-FREE PATCH INFERENCE

In this chapter we develop an algorithm which we call `spfind` for finding program transformations specified in a language with named-subpart abstractions and context-free denotation of locations. We call such a program transformation a context-free patch for two reasons: 1. The program transformations are basically sequences of term rewrite rules, so they have a strong similarity with context-free grammar production rules, and 2. the specifications can be viewed as a subset of the semantic patches specified using the SmPL language.

We begin with a motivating example that illustrate the kinds of patches that can be inferred by the context-free patch inference method.

5.1 MOTIVATING EXAMPLE

To motivate the design of `spfind`, we begin with a simple example of a collateral evolution from March 2007[1] and consider the issues involved in inferring a context-free patch for it. The collateral evolution required replacing uses of the general-purpose memory copying function `memcpy` that manages network buffers by calls to a special-purpose function, `skb_copy_from_linear`.

Figure 2 shows extracts of two files affected by this collateral evolution and the updates to these files. The lines prefixed with `-` and `+` indicate where code was removed and added, respectively. Furthermore, the line that is prefixed with `!` has superficially the same form as the others, in that it represents a call to `memcpy`, but it is not affected by the collateral evolution. In the first file, two calls to `memcpy` are present initially and only one is affected and in the second file only one such call is affected.

A summary of the changes is shown in in the bottom-right part of Figure 2. The summary reveals that although there are differences in how the two files were modified, there are also compelling similarities:

1. All calls to `memcpy` where the second argument references the field `data` from a `sk_buff` structure are changed into calls to `skb_copy_from_linear_data`. On the other hand, in the call to `memcpy` marked with a `!`, the second argument does *not* reference the field `data`.

2. The first argument becomes the second.

1 Git SHA1 identification codes `1a4e2d093fd5f3eaf8cffc04a1b803f8b0ddef6d` and `d626f62b11e00c16e81e4308ab93d3f13551812a`.
All patches in this chapter can be obtained from
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

```
 static int ax25_rx_fragment(
                ax25_cb *ax25,
                struct sk_buff *skb)
 {
   struct sk_buff *skbn, *skbo;

   if (ax25->fragno != 0) {
   ...
   /* Copy data from the fragments */
   while ((skbo = skb_dequeue(
                  &ax25->frag_queue))
          != NULL) {
-    memcpy(skb_put(skbn, skbo->len),
-           skbo->data,
-           skbo->len);
+    skb_copy_from_linear_data(
+           skbo,
+           skb_put(skbn, skbo->len),
+           skbo->len);

     kfree_skb(skbo);
   }
   ...
 }
 static int ax25_rcv(
           struct sk_buff *skb, ...)
 {
   ...
   if (dp.ndigi == 0) {
     kfree(ax25->digipeat);
     ax25->digipeat = NULL;
   } else {
     /*Reverse the source SABM's path*/
!    memcpy(ax25->digipeat, &reverse_dp,
!           sizeof(ax25_digi));
   }
   ...
 }
```

File: `net/ax25/ax25_in.c`

```
 static
 struct sk_buff *dnrmg_build_message(
        struct sk_buff *rt_skb,
        int *errp)
 {
   struct sk_buff *skb = NULL;
   ...
   if (!skb)
     goto nlmsg_failure;
   ...
-  memcpy(ptr, rt_skb->data,
-              rt_skb->len);
+  skb_copy_from_linear_data(
+         rt_skb, ptr, rt_skb->len);
   ...
 nlmsg_failure:
   if (skb)
     kfree_skb(skb);
   ...
 }
```

File: `net/decnet/netfilter/dn_rtmsg.c`

```
-  memcpy(skb_put(skbn, skbo->len),
-         skbo->data,
-         skbo->len);
+  skb_copy_from_linear_data(
+       skbo,
+       skb_put(skbn, skbo->len),
+       skbo->len);

-  memcpy(ptr, rt_skb->data,
-              rt_skb->len);
+  skb_copy_from_linear_data(
+         rt_skb, ptr, rt_skb->len);
```

Set of changes for the two files

Figure 2: Extracts of the two files and the set of changes for the two files. (bottom-right).

3. The field reference to `data` in the second argument is dropped. The resulting expression becomes the first argument of the new function call.

4. The third argument of `memcpy` is copied as-is to the third argument of the new function call.

The changes made to the two mentioned files can be summarised compactly as the following context-free patch derived by `spfind`:

```
@@
    expression X0;
```

```
    struct sk_buff * X1;
    expression X2;
@@
-   memcpy(X0,X1->data,X2)
+   skb_copy_from_linear_data(X1,X0,X2)
```

where `X0`, `X1`, and `X2` serve as placeholders (meta-variables) for concrete arguments. The meta-variables are declared to match arbitrary `expressions` (`X0` and `X2`) and an expression of type `struct sk_buff *` (`X1`). Intuitively, the context-free patch is an abstract representation of the changes made: in the context of a call to `memcpy` where the first and third arguments are arbitrary expressions and the second is of type `struct sk_buff *` and references the `data` field of the `sk_buff` structure, then change the called function to `skb_copy_from_linear_data`, move the first argument to the second position, remove the `data` field reference of the second argument and make it the first argument in the new function call, and copy the third argument as-is. Thus, the combined requirements on the context in which to make a transformation ensure that only the calls marked with `-` are affected and leave out the call to `memcpy` marked with `!`, as required.

There are two main issues to be considered when inferring context-free patches: 1) compactness and 2) safety.

COMPACTNESS The most trivial way to construct a context-free patch is simply to enumerate the changes, as initially done for the example earlier in this section. The result, however, would be no better than a standard patch, and it would generally not be applicable to files other than the ones used for the inference. Finally, it would generally not be readable as high-level documentation of the changes performed. We prefer, therefore, a more compact description of the changes. We produce the more compact description by replacing subterms that are not affected by the transformation by meta-variables. The use of meta-variables is illustrated in the context-free patch above where *e.g.*, `X0` is used rather than the concrete terms `skb_put(skbn,skbo->len)` (in the file `ax25_in.c`) and `ptr` (in the file `dn_rtmsg.c`).

SAFETY The safety of a context-free patch requires that only things that actually changed in the original file should be changed by the inferred context-free patch. In our example, one of the calls to `memcpy` was not changed. We saw that we could ensure safety by imposing structural and type-based restrictions on the second argument to `memcpy`: only those calls where the second argument had the correct type and referenced the `data` field should be affected.

INSTANTIATION OF TRANSFORMATION PART FRAMEWORK In the following we define the syntax and semantics of context-free patches precisely. Our aim is to instantiate the transformation part and subsumption relation given in Chapter 4 to the language of context-free patches. In the following we will use the term *subpatch* to denote the program transformation subsumption relation instantiated with the context-free patch language. Once, the transformation language, and in particular the *application function* for context-free patches have been defined, we can the state an algorithm that tries to find the set $LCT(CS)$.

5.2 CONTEXT-FREE PATCHES

A context-free patch is specification of a program transformation that consists of a sequence of rewrite rules. We call the rewrite rules term replacement patches and a sequence of such patches a context-free patch. The examples in Chapter 4 made use of a simple variant of this language without multiple rules and meta-variables.

TERM REPLACEMENT PATCHES A term replacement patch describes how to transform any (sub)terms that match a given pattern.

5.1 *Definition (Syntax of Term Replacement Patches) The syntax of term replacement patches is a combination of two term patterns* $\mathfrak{p}$. *Recall that we have already defined the syntax of term patterns in Definition 3.3.*

$$trp ::= \mathfrak{p} \rightsquigarrow \mathfrak{p}$$

5.2.1 *Application function*

The application of a term replacement patch to a term is defined by the rules shown in Figure 3.[2] Rule $\mathfrak{a}$ is concerned with the case where a term $\mathfrak{t}$ matches the pattern $\mathfrak{p}$ according to some substitution θ. The matching term is replaced with $\theta\mathfrak{p}'$. We further require that the meta-variables in the right-hand side pattern $MV(\mathfrak{p}')$ be a subset of those in the left-hand side pattern $MV(\mathfrak{p})$–otherwise a right-hand side could contain meta-variables for which no value would be known. The remaining rules traverse the term top-down along all subterms of the term (rule $\mathfrak{b}$) until reaching a subterm at which rule $\mathfrak{a}$ applies or until reaching an atomic term (rule $\mathfrak{c}$). (Note that rules $\mathfrak{b}$ and $\mathfrak{c}$ only apply if rule $\mathfrak{a}$ does not.) If there is no matching subterm, the application of a term replacement patch behaves as the identity function.

The application of a term replacement patch additionally returns a flag $\top$, when a match has been found, or $\bot$, when no match has been found. These are ordered as $\bot \sqsubseteq \top$. Note that even if there is a match, the resulting generated term might be the same as the original one, *e.g.*, if the term replacement patch specifies that the two arguments of a function would be switched, and they are actually textually equal.

CONTEXT-FREE PATCHES A context-free patch is a sequence of one or more term replacement patches, as defined by the following grammar:

5.2 *Definition (Syntax of Context-free Patches) A context-free patch is either a term replacement patch or a sequence of context-free patches.*

$$gp ::= \mathfrak{p} \rightsquigarrow \mathfrak{p} \mid gp; gp$$

2 Note that although term replacement patches have the form of rewrite rules, they are not applied iteratively as done in term rewriting systems.

$$(a)\quad \frac{\exists\theta : \theta p = t \qquad \theta p' = t' \qquad MV(p') \subseteq MV(p)}{(p \rightsquigarrow p')(t) = t', \top}$$

$$(b)\quad \frac{\begin{array}{c}\neg\exists\theta : \theta p = a(t_0, \ldots, t_n) \\ (p \rightsquigarrow p')(t_i) = t'_i, f_i \text{ for all } 0 \leqslant i \leqslant n \\ F = \bigsqcup f_i\end{array}}{(p \rightsquigarrow p')(a(t_0, \ldots, t_n) = a(t'_0, \ldots, t'_n), F}$$

$$(c)\quad \frac{\neg\exists\theta : \theta p = a}{(p \rightsquigarrow p')(a) = a, \bot}$$

Figure 3: Application of a term replacement patch.

Subsequently, whenever we say "patch," we mean context-free patch unless stated otherwise. Also, we say that a context-free patch is *non-abstract* if no pattern used in the context-free patch contains any meta-variables.

The rules for applying a context-free patch are shown below. The application of a term replacement patch $p_1 \rightsquigarrow p_2$ is defined according to the rules of Figure 3, but here the application only succeeds if the pattern matches somewhere, as indicated by the flag $\top$. A sequence of patches $gp_1; gp_2$ first applies gp_1 to the term and then applies gp_2 to the result.

$$\begin{array}{lcll}[\![p \rightsquigarrow p']\!]t & = & t'' & \text{if } (p \rightsquigarrow p')(t) = t'', \top \\ [\![gp_1; gp_2]\!]t & = & [\![gp_2]\!]([\![gp_1]\!]t)\end{array}$$

It is possible to show that application of context-free patches is associative. I.e. $[\![(gp_1; gp_2); gp_3]\!]t = [\![gp_1; (gp_2; gp_3)]\!]t$. If the result of applying $gp_1; gp_2$ to a term is independent of the ordering of gp_1 and gp_2, the patches are said to be *commutative*.

Two context-free patches are equivalent with respect to a set of terms T if and only if application of the patches have the same effect on all terms:

$$gp_1 \cong gp_2 \iff \forall t \in T : [\![gp_1]\!]t = [\![gp_2]\!]t$$

Two context-free patches are equivalent with respect to a changeset CS if and only if application of the patches have the same effect on all *left-hand side* terms in CS:

$$gp_1 \cong_C gp_2 \iff \forall (t, t') \in CS : [\![gp_1]\!]t = [\![gp_2]\!]t$$

Whenever the changeset, CS, is clear from the context, we will write $gp_1 \cong gp_2$ instead of $gp_1 \cong_{CS} bp_2$.

5.3 ALGORITHM

We now present an algorithm for finding the set of maximal common subpatches given a set of pairs of terms, CS, as well as details about the implementation of the algorithm.

We first present a very simple algorithm for computing maximal common subpatches that leaves out any performance concerns and other details that must be addressed when implementing the algorithm. This very simple algorithm is mainly to be considered as an outline of the algorithm's basic structure. Based on an observation about a "sufficient level of abstraction" we then refine the algorithm in two steps. The refined algorithm returns a subset of what the simple algorithm returns such that all patches returned are of minimal abstractness. Finally, we present the main two remaining issues we have addressed in our implementation: noise in the input data and constant-time comparison of terms.

5.3.1 *A simple algorithm*

We now present the simple version of our algorithm to compute the set of maximal common subpatches for a given changeset. The algorithm is denoted `spfind`. Overall, `spfind` works in two steps: 1) finding term replacement patches (i.e. patches of form $p \rightsquigarrow p'$) and 2) growing larger sequential patches (i.e., patches of form $bp_1;...;bp_n$) from the set of found term replacement patches.

The pseudocode of the `spfind` algorithm is written in the style of a functional programming language, such as O'Caml or Standard ML. In the pseudocode we frequently make use of set-comprehensions to compute sets.

5.3 Definition (`spfind` – simple version) The simple version of the algorithm finding context-free patches is given below.

```
simple_pairs CS =
  { p->p' | (lhs,rhs) ∈ CS,
            t is a subterm of lhs,
            t' is a subterm of rhs,
            ∃θ,p,p': θp = t , θp' = t',
            p⇝p'≤(lhs,rhs)
  }

computeNext CS B cur = {p⇝p'| p⇝p' ∈ B, (cur;p⇝p') ≤ CS}

gen CS B cur =
  let next = computeNext CS B cur in
  if next == {}
  then {cur}
  else {gp | p⇝p' ∈ next, gp ∈ gen CS B (cur;p⇝p')}

spfind CS =
  let B = simple_pairs CS in
  {gp | gp ∈ gen CS B bp, bp ∈ B}
```

The main functions in the algorithm are `simple_pairs` and `gen`. The function `simple_pairs` constructs term-replacement patches and the function `gen` grows larger sequential context-free patches.

CONSTRUCTION OF TERM-REPLACEMENT PATCHES The `simple_pairs` function takes a set of pairs of terms CS and constructs the set $\{gp \mid gp \preceq CS\}$ by considering abstractions of all subterms in the given set of pairs of terms CS. In the pseudocode, patterns `p` and `p'` (that may contain meta-variables) more or less "magically" appear in the line with existential quantification. In the refined algorithm in Section 5.3.2 we describe a method to construct such patterns from a set of terms. Another, more serious, issue is that even for small examples, the `simple_pairs` function can return *many* term replacement patches–an upper bound for $p \rightsquigarrow p'$ is $\mathcal{O}(2^{|p|*|p'|})$ where $|\cdot|$ denotes the number of subterms in a pattern. We illustrate this in Example 5.4 below.

5.4 *Example* Let the following be given

```
t1 = g(f(42))       t1'= g(f(117))
t1'= g(f(42,42))    t2'= g(f(117,117))
CS = { (t1,t1') ,   (t2, t2') }
```

Part of the result of `simple_pairs C` is show below. Recall that capital letters denote meta-variables.

```
simple_pairs C =
{
   g(f(X)) ⇝ g(f(X,X)),  g(Y(X)) ⇝ g(f(X,X)),
   Z(f(X)) ⇝ g(f(X,X)),  Z(Y(X)) ⇝ g(f(X,X)),
   g(Y(X)) ⇝ g(Y(X,X)),  Z(f(X)) ⇝ Z(f(X,X)),
   Z(Y(X)) ⇝ Z(Y(X,X)),  ...
}
```

The above result-set for `simple_pairs` illustrates that many very similar patches can be returned. In this example the complete set contains $\mathcal{O}(2^{5*6})$ term replacement patches where 5 is the number of subterms of `g(f(42))` and 6 is the number of subterms of `g(f(42,42))`. A few of those are not safe for the changeset, but most are. Thus, we need to find a way to limit the size of this set to return fewer representative patches. This will be done in Section 5.3.2.

GROWING SEQUENTIAL CONTEXT-FREE PATCHES The `gen` function takes three inputs: a changeset `CS`, a set of term-replacement patches `B`, and a context-free patch `cur`. The function then tries recursively to extend the context-free patch `cur` with term replacement patches from the set `B`. Thus, a call to `gen CS B cur` returns a set of patches that have the shape $\texttt{cur}; p_1 \rightsquigarrow p_1; \cdots; p_n \rightsquigarrow p'_n$ where $p_i \rightsquigarrow p'_i \in \texttt{B}$ and $\texttt{cur}; p_1 \rightsquigarrow p_1; \cdots; p_n \rightsquigarrow p'_n \preceq \texttt{CS}$.

The main function of the `spfind` algorithm calls `gen` for each term replacement patch `bp` (`cur` in `gen`) found by `simple_pairs`.

RELATIONSHIP BETWEEN ALGORITHM AND SPECIFICATION Unfortunately, the `spfind` algorithm is neither sound nor complete in the strict sense. There are patches in $LCT(CS)$ that `spfind` will not find and indeed some patches found by `spfind` are not largest. An example illustrate both cases and hints at why the lack of soundness and completeness is not as bad as it sounds.

5.5 *Example (No soundness or completeness)* Assume the following definitions are given:

```
t1  = h(f(1),1)
t1' = h(f(2),3)
t2  = t(f(1),42,1,117))
t2' = t(f(2),42,3,117))
CS  = {(t1,t1'), (t2,t2')}
```

We notice that $gp_1 = \texttt{f(1)} \rightsquigarrow \texttt{f(2)}$ is safe for `CS` while $gp_2 = \texttt{1} \rightsquigarrow \texttt{3}$ is not because it updates too many occurrences of `1`. However, $gp_1;gp_2$ *is* safe for `CS` because now, when gp_2 is applied, there is only one occurrence of `1` and that occurrence is supposed to be updated to `3`. In fact, $gp_1;gp_2$ is one of the largest common subpatches for `CS`.

All patches returned by `spfind` are either term replacement patches or sequences of term replacement patches all of which are individually safe for the changeset given. Therefore, `spfind C` will not contain a patch that is composed of $\texttt{1} \rightsquigarrow \texttt{3}$. It can also be shown that `spfind CS` cannot contain a patch that is equivalent to $gp_1;gp_2$ because the occurrences of `1` in `t1` and `t2` that have to be changed have no common term structure expressible in a context-free patch.

Therefore, we can conclude that `spfind` is not complete. The result of `spfind CS` will contain gp_1 which is not in $LCT(CS)$. We can therefore also conclude that `spfind` is not sound in the strict sense that for all patches found by `spfind CS`, there must exist an *equivalent* patch in $LCT(CS)$.

Example 5.5 gives a counterexample that shows that `spfind` is neither sound nor complete. However, it *is* the case that the patches returned by `spfind CS` are either equivalent to some in $LCT(CS)$ or they are each a subpatch of some patch in $LCT(CS)$ and the application of the subpatch has an impact on the changeset (i.e. it is not the identity patch). Thus, each patch found by `spfind CS` is safe for `CS` and when applied, it will have *some* safe effect on the terms.

5.3.2 *Towards a refined algorithm*

There are two problems in the simple version of the `spfind` algorithm:

Term replacement patches: In the function `simple_pairs`, we use existential quantification to introduce patterns that contain meta-variables. There are two problems to tackle: 1) how do we actually find such patterns and 2) is there a *sufficient* subset of this set of patterns?

Search-space pruning: The `gen` function uses the `computeNext` function to find a subset of the set of term replacement patches `B` that can be used to extend the context-free patch being generated (`cur`). For each term replacement patch in this subset, `gen` will try to extend the current context-free patch with the particular element. Thus, if we can limit the size of the `next` subset, fewer calls to `gen` ensue.

SUFFICIENT TERM REPLACEMENT PATCHES Consider once more the terms from Example 5.4:

```
t1 = g(f(42))       t1'= g(f(117))
t1'= g(f(42,42))    t2'= g(f(117,117))
CS = { (t1,t1') ,   (t2, t2') }
```

As shown in Example 5.4 there are many potential term replacement patches that are safe for the set `CS`. Let $B = \{p \rightsquigarrow p' \mid p \rightsquigarrow p' \preceq \texttt{CS}\}$. In order to define a sufficient subset of B, we observe that some of the term replacement patches are needlessly abstract. A term replacement patch is needlessly abstract if there is an equivalent term replacement patch that is less abstract. For the set `CS` above, the most abstract patch is `X(Y(Z(Q)))` $\rightsquigarrow$ `X(Y(Z(Q,Q)))` and the least abstract is `g(f(X))` $\rightsquigarrow$ `g(f(X,X))`. It is easy to see that in this example all elements of B are equivalent. The important property of two equivalent term replacement patches is that the patches that can be grown from one are equivalent to the patches that can be grown from the other:

5.6 Lemma Equivalent patches imply equivalent suffix extensions

$$\forall trp_1, trp_2, gp : trp_1 \cong trp_2 \Rightarrow trp_1; gp \cong trp_2; gp$$

The conclusion one can draw from Lemma 5.6 is that for a set of equivalent term replacement patches, we only need to return one of them.

FINDING TERM REPLACEMENT PATCHES Given the set $\{p \rightsquigarrow p' \mid p \rightsquigarrow p' \preceq C\}$ we can construct the sufficient subset of term replacement patches based on the above observations. The naive approach takes the result of `simple_pairs` a returns only those that are not needlessly abstract according to the description above. The thusly found sufficient term replacement patches can then be partitioned in equivalence classes and we then only need to return one term replacement patch from each (cf. Lemma 5.6).

However, we would still need to construct the initial set and that is potentially very time-consuming. With the goal of avoiding the construction of the complete initial set, we now define a fusion operator on term replacement patches, $trp_1 \otimes trp_2$. The patch fusion operator constructs a new term replacement patch subsuming both of the given patches. The *patch* fusion operator relies on a *pattern* fusion operator which we state requirements for below. The definition of patch fusion is then given in Definition 5.7.

In order to perform patch fusion, we need to be able to fuse two patterns into a new pattern that is a superpattern of both given patterns. A simple definition of the pattern fusion function could be one that always returns the trivial pattern X but such a definition is useless for our

purpose. Instead we would like the pattern resulting from $p * p'$ to abstract as little as possible in order to obtain a pattern that matches both p and p'. In Chapter 3 we defined the concept of the most specific pattern, which matches the current requirements precisely. The function computing the most specific is given in Definition 3.15. We denote the fusion of patterns p and p' as $p * p'$.

Using the pattern fusion function we can define fusion of term replacement *patches* $gp \otimes gp'$ as the fusion of the embedded patterns in the patches and a renaming of meta-variables. The definition can be seen in Definition 5.7 below.

5.7 *Definition (Fusion of term replacement patches)* *Fusion of two patches into a new one is defined as:*

$$\begin{aligned} &p_1 \rightsquigarrow p_1' \otimes p_2 \rightsquigarrow p_2' = p_3 \rightsquigarrow (\theta_3 p_3') \iff \\ &\quad \theta_1 \vdash p_1 * p_2 = p_3 \wedge \\ &\quad \theta_2 \vdash p_1' * p_2' = p_3' \wedge \\ &\quad \theta_2 \textit{ renamed\_by } \theta_1 = \theta_3 \end{aligned}$$

θ renamed_by θ' renames meta-variables in θ using those in θ'. The resulting θ'' is therefore a mapping from meta-variables to meta-variables.

$$\theta \textit{ renamed\_by } \theta' = \theta'' \iff \forall X \in \mathit{dom}(\theta) : \theta''(X) = \begin{cases} Y & \textit{if } \exists Y : \theta'(Y) = \theta(X) \\ \hat{X} & \textit{otherwise, where } \hat{X} \notin \mathit{dom}(\theta') \end{cases}$$

It is instructive to consider an example of term replacement patch fusion to see what is going on.

5.8 *Example (Patch Fusion)* Let the following be given:

```
gp1  = f(42,h(117))  ⇝  f(43,h(117+117))
gp2  = f(42,q(118))  ⇝  f(43,q(118+118))
```

We now compute `gp1` $\otimes$ `gp2`, step by step. First we need to fuse the two left-hand sides and the two right-hand sides corresponding to the first two conjuncts in Definition 5.7.

```
[left-hand sides]
f(42,h(117)) * f(42,q(118)) = f(42,X0(X1))

[right-hand sides]
f(43,h(117+117)) * f(43,q(118+118)) = f(43,X1(X3+X3))
```

The environments associated with each fusion are as follows:

```
env   = { X0 ↦ (h,q), X1 ↦ (117,118) }
env'  = { X1 ↦ (h,q), X3 ↦ (117,118) }
```

When fusing patterns we are allowed to select any meta-variable names as long as the inference rules in Definition 3.15 can be used to derive the desired fused patterns. For illustration, when fusing the left hand side (non-abstract) patterns we used the meta-variables `X0` and `X1` while when fusing the right hand side (non-abstract) patterns, we used `X1` and `X3`. In fact, the environments need not be consistent with each other. For example, `X1` in the left hand side pattern maps to `(117,118)` while in the right hand side pattern `X1` maps to `(h,q)`. This selection of names is mainly done to illustrate how θ *renamed_by* θ' works and why it is needed.

```
env'' = env' renamed_by env = { X1 ↦ X0, X3 ↦ X1}
```

One can verify that `env"` satisfies the requirements set out in the definition of `p` *renamed_by* `p'` in Definition 5.7 above. Finally, we can construct the fused term replacement patch as:

```
gp1 ⊗ gp2 = f(42,X0(X1)) ⇝ env''(f(42,X1(X3+X3)))
          = f(42,X0(X1)) ⇝ f(42,X0(X1+X1))
```

SEARCH-SPACE PRUNING Even with the above mentioned refinements of the `simple_pairs` function, the `gen` function can end up constructing an exponential number of context-free patches. We now consider how to further reduce the number of generated context-free patches.

The `gen` function in the simple version of the `spfind` algorithm is given a set `B` of term replacement patches and tries to grow sequential patches starting from each patch in `B`. If all those patches are safe for the changeset *CS* also given to `gen`, the `gen` function would have to compute the powerset of `B`. Suppose however, that all term replacement patches in `B` are *commutative*. The `gen` function will first try to grow context-free patches starting from some element *gp* in `B`. After having found all context-free patches that start with *gp*, the `gen` function then tries to grow context-free patches starting with some other element *gp'* from `B`. Since we assumed all patches in `B` to be commutative, all context-free patches starting with *gp* found by `gen` are equivalent to those found to be starting with *gp'*. The following corollary is a generalization of the observation just made that whenever the `gen` function has already found some safe context-free patch for *CS* denoted *gp*, there is no need to try to grow from a subpatch *gp'*. The corollary holds even when there are non-commutative patches in the set `B` as otherwise assumed above.

5.9 *Corollary (Search-space pruning) For any context-free patches* gp, gp', gp''*:*

$$gp \in LCT(CS) \wedge gp' \preceq^{CS} gp \wedge gp';gp'' \preceq CS \Rightarrow gp';gp'' \preceq^{CS} gp$$

The corollary follows directly from the definition of what it means for a program transformation to be maximal and the subsumption relation. (Definitions 4.11 and 4.10).

```
simple_pairs_one (t,t') =
  let loop lhs rhs =
    if lhs == rhs
    then {}
    else case (lhs,rhs) of
      | (a1(ts1), a2(ts2)) ->
          let R = {tu | t1 ∈ ts1, t2 ∈ ts2,
                        tu ∈ simple_pairs_one t1 t2
                  } in
          {lhs⇝rhs} ∪ R
      | otherwise -> {lhs⇝rhs}
  in
    {tu | tu ∈ loop t t', tu≤(t,t')}

simple_pairs C =
  let loop tu_lists = case tu_lists of
      | [] -> {}
      | [tu_list] -> tu_list
      | (tu_list::tu_lists) ->
           let tu_merged = {tu | tu ∈ loop tu_lists} in
           {tu ⊗ tu_merged | tu ∈ tu_list, tum ∈ tu_merged,
                             (tu ⊗ tum) ≤ CS}
in
  let lists = map find_simple_updates_one CS
  in
    loop lists
```

Figure 4: Generation of term replacement patches

5.3.3 *The refined `spfind` algorithm*

We now proceed to describe the refined `spfind` algorithm making use of the solutions to the two problems identified for the simple algorithm.

The `spfind_refined` algorithm is split into two main parts just like the simple version, `spfind`. The former part (the `simple_pairs` function) can be seen in Figures 4 and the latter (the `gen` function as well as the entry point of the algorithm) in Figure 5.

GENERATION OF TERM REPLACEMENT PATCHES In order to generate a set of term replacement patches based on a given changeset, we first find a set of sets of non-abstract patches and then fuse them to obtain term replacement patches with meta-variables.

The function `simple_pairs_one` defined at the top of Figure 4 finds non-abstract term replacement patches. The function works by taking a pair of terms and finding non-abstract patches that all satisfy the safety criterion: $t_i \rightsquigarrow t'_i \preceq (t, t')$. The function does so by traversing the given pair of terms simultaneously. If the two terms are not equal, a non-abstract patch is added to an intermediate result set. If the two terms are also compound, the function calls itself recursively on the embedded subterms. Once the traversal is done, all non-safe patches are filtered out and only the safe ones are returned.

The `simple_pairs` function (bottom Figure 4) is given a changeset $C = \{(t_1, t'_1), ..., (t_n, t'_n)\}$ and applies the `simple_pairs_one` function to each pair, obtaining a set of sets of non-abstract term replacement patches:

$$\texttt{tu\_lists} = \left\{ \left\{ \begin{array}{c} t_{1,1} \rightsquigarrow t'_{1,1} \\ t_{1,2} \rightsquigarrow t'_{1,2} \\ \vdots \\ t_{1,k} \rightsquigarrow t'_{1,k} \end{array} \right\}, \cdots, \left\{ \begin{array}{c} t_{n,1} \rightsquigarrow t'_{n,1} \\ t_{n,2} \rightsquigarrow t'_{n,2} \\ \vdots \\ t_{n,q} \rightsquigarrow t'_{n,q} \end{array} \right\} \right\}$$

where $t_{i,j} \rightsquigarrow t'_{i,j}$ corresponds to the jth non-abstract patch for the ith pair in C. This function now uses the patch fusion operator $trp \otimes trp'$ to construct more abstract patches. In the following, let `tu_merged` denote the fused patches so far. We describe the three cases of the local `loop` function of `simple_pairs` below. When `loop` is done, `tu_merged` will contain a set of term replacement patches that are safe for all pairs in the changeset.

`tu_lists` is empty: If `tu_lists` is empty, the changeset *CS* was also empty and thus `tu_merged` should also be empty.

prgtu_lists is a singleton: In this case there is no other set of patches to fuse with, so we simply let `tu_merged` be the only list in prgtu_lists at this point. Doing so ensures that the patches returned are minimally abstract.

prgtu_lists contains two or more sets of patches Let $\texttt{tu\_lists}_i$ denote one set of patches $t_{i,j} \rightsquigarrow t'_{i,j}$ for some `i` and `tu_lists'` denote $\texttt{tu\_lists} \backslash \texttt{tu\_lists}_i$. The `loop` function first fuses the patches in `tu_lists'` and then combines each of the fused patches with one from $\texttt{tu\_lists}_i$.

```
computeNext C B cur = { p⇝p' | p⇝p' ∈ B , cur;p⇝p' ≤ C }

gen C B acc_res cur =
  let next = computeNext C B cur in
  if next == {}
  then {cur} ∪ acc_res
  else fold (λacc_res bp ->
              if ∃gp ∈ acc_res:(cur;bp)≤_C gp        (***)
              then acc_res
              else gen C B acc_res (cur;bp)
            ) acc_res next

spfind_refined C =
  let B = simple_pairs C in
  fold (λacc_res bp ->
              if ∃gp ∈ acc_res:bp≤_C gp
              then acc_res
              else gen C B acc_res bp
            ) [] B
```

Figure 5: Growing sequential context-free patches and entry point of the refined algorithm.

If the fused patch is not safe for the changeset, it is not included in `tu_merged` and otherwise it is. Using the safety check to limit the size of `tu_merged` is crucial to reduce the running time of the `loop` function. If we had deferred the safety check to outside of the `loop` function it would need to compute $|\mathtt{prgtu_1ists_1}| \times |\mathtt{prgtu_1ists_2}| \times \cdots \times |\mathtt{prgtu_1ists_n}|$ fused patches.

GROWING SEQUENTIAL CONTEXT-FREE PATCHES The essential difference between the refined `gen` function shown in Figure 5 and the simple version is that before each recursive call to `gen`, the refined version checks that the extended patch (`cur;bp`) is not a subpatch of one it already found in the accumulated results, `acc_res`. The conditional test is marked with $***$ to indicate the essential difference between the old and new version of the `gen` function. Based on Corollary 5.9 we can see that extending `cur` with `bp` will only allow us to find patches that are subpatches the patches that are already in `acc_res`. Therefore, the recursive call is not performed and instead `acc_res` is taken as the result.

CONTEXT-SENSITIVE PATCH INFERENCE

Chapter 6

In this section we instantiate the transformation parts framework of Chapter 4 with a transformation language that is more expressive than the one used to express context-free patches. Specifically, the transformation language described in the following is a simple subset of the SmPL language used in the Coccinelle project. The language we describe allows specification of transformations with a context-sensitive denotation of locations (control-flow paths) and named subpart abstractions (meta-variables). We therefore refer to such programs transformations as *semantic patches*. We first present a motivating example that shows a collateral evolution that can not be expressed as a context-free patch. We then describe the transformation language along with some key properties and then finally the actual algorithm for finding the set $LCT(CS, n)$ expressible with semantic patches.

6.1 MOTIVATING EXAMPLE

We consider a collateral evolution that was performed between May and July 2007.[1] The corresponding evolution changed the way file-system drivers clear certain memory regions. Before the evolution, memory was cleared by a call to `memset`, preceded and followed by calls to `kmap_atomic` and `kunmap_atomic`, respectively. To reduce the amount of duplicated code, a new library function was introduced to handle all three operations. This evolution then required a collateral evolution: replacing the associated fragments of code in drivers by a call to the new library function.

Below is an excerpt of the first standard patch in a sequence of standard patches implementing the collateral evolution mentioned above.

```
--- inode.c.orig 2009-08-24 14:23:31.000000000 +0200
+++ inode.c.new  2009-08-24 14:23:31.000000000 +0200
@@ -1767,7 +1767,6 @@ static int ext3_block_truncate_page(hand
@@ -1835,11 +1831,7 @@ static int ext3_block_truncate_page(hand
    goto unlock;
  }

- kaddr = kmap_atomic(page, KM_USER0);
- memset(kaddr + offset, 0, length);
- flush_dcache_page(page);
- kunmap_atomic(kaddr, KM_USER0);

+ zero_user_page(page, offset, length, KM_USER0);
  BUFFER_TRACE(bh, "zeroed end of block");
```

1 The initial change occurs with `git` SHA1 identification code 01f2705daf5a36208e69d7cf95db9c330f843af6. obtained from http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

The standard patch code removes the call to `memset` and associated calls to `kmap_atomic` and `kunmap_atomic`, and replaced them with a call to `zero_user_page`. A call to the function `flush_dcache_page` is also removed. The following is another excerpt from the same standard patch:

```
--- loop.c.orig 2009-08-18 15:28:16.000000000 +0200
+++ loop.c.new 2009-08-18 15:28:16.000000000 +0200
@@ -243,17 +243,13 @@ static int do_lo_send_aops(struct loop_d
   transfer_result = lo_do_transfer(lo, WRITE, page, offset,
     bvec->bv_page, bv_offs, size, IV);
   if (unlikely(transfer_result)) {
-   char *kaddr;
-
    /*
     * The transfer failed, but we still write the data to
     * keep prepare/commit calls balanced.
     */
    printk(KERN_ERR "loop: transfer error block %llu\n",
       (unsigned long long)index);
-   kaddr = kmap_atomic(page, KM_USER0);
-   memset(kaddr + offset, 0, size);
-   kunmap_atomic(kaddr, KM_USER0);
+   zero_user_page(page, offset, size, KM_USER0);
   }
   flush_dcache_page(page);
```

This standard patch code also replaces the call to `memset` and associated calls to `kmap_atomic` and `kunmap_atomic` with a call to `zero_user_page`. In addition, the declaration of the `kaddr` variable is removed, as it is no longer useful. In contrast to the previous standard patch, however, the call to `flush_dcache` is not removed.

While there are some differences in what changes are performed by the above two standard patches, there are also some very compelling similarities: 1) a wrapped call to `memset` and the wrapping functions are removed and replaced with a call to `zero_user_page`, 2) the arguments to the removed functions are passed to the newly inserted function call. These similarities are, however, obscured by the large size of the patch (over 300 lines), and by the small variations in the instances as described above. We would thus like to see a more high-level specification of the common changes.

Below is a semantic patch inferred by our tool. The semantic patch compactly summarizes the *common* changes mentioned.

```
@@
  expression X0, X1, X2, X3;
@@
- X3 = kmap_atomic(X2, KM_USER0);
  ...
- memset(X3 + X0, 0, X1);
  ...
- kunmap_atomic(X3, KM_USER0);
+ zero_user_page(X2, X0, X1, KM_USER0);
```

Intuitively, the semantic patch removes calls to `kmap_atomic` that are always followed by a call to `memset` that is again always followed by a call to `kunmap_atomic`. In contrast, context-free patches could only propose rewriting all instances of these three kinds of terms individually, which would be incorrect, as there are e.g. many other uses of `memset` that should not be affected by this transformation. The semantic patch furthermore leaves out the changes that are not common, i.e., it does not specify that in one case the call to `flush_dcache` is to be removed while in the other case the declaration of a variable `kaddr` should be removed.

The use of meta-variables in this semantic patch furthermore enforces some necessary relationships between the arguments of the various affected calls. In particular, the first argument to `memset` should be an addition expression where the first argument to the addition expression is syntactically equal to both the variable assigned to by the preceding call to `kmap_atomic` and the first argument to the succeeding calls to `kunmap_atomic`. This expression is not used in the generated call, but instead only the second operand of the addition is used, as the second argument.

We now describe our approach for finding semantic patches from a change set, as implemented by our `spdiff` tool. The approach is split in two steps: 1) find common patterns in functions that contain transformations, and 2) add transformation operations to the found patterns, resulting in semantic patches.

This section defines semantic patches and the subpatch relation between them.

6.2 SEMANTIC PATCHES

A semantic patch is a specification of a transformation. We consider two types of transformations: term removal, denoted by `-`, and term addition, denoted by `+`. A semantic patch is matched against the terms of the input program and transforms it according to the specified operations. The minimal element of a semantic patch is a *patch term* (*PT*), which is either a meta-variable, a wildcard (denoted _), an atom or a collection of several patch terms composed by the term constructor c. Patch terms that are unannotated or annotated with "-" are matched against the terms of the input program. The syntax of semantic patches is given below.

$$
\begin{array}{lcl}
pt & ::= & \epsilon \mid \mathrm{O}\ pt \\
\mathrm{O} & ::= & PT \mid \ldots \mid \text{-}PT \mid \text{+}PT \\
PT & ::= & \mathit{Meta} \mid \_ \mid \textsc{atom} \mid \mathfrak{a}(PT^*)
\end{array}
$$

A semantic patch *pt* consists of zero or more operations O. O is either a patch term, "...", or one of the transformation operations: -*PT* or +*PT*. The "..." operation matches a sequence of zero or more terms. A wildcard patch term matches any term in the code the semantic patch is applied to.

Previous work [11, 29, 49] has defined the application of various kinds of semantic patches. In this dissertation we omit the formal definition of semantic patch application, instead relying on the intuition developed in Section 6.1, as our main focus is on *inference* of semantic patches. Application of a semantic patch to a term is written $[\![pt]\!](\mathrm{t})$.

6.3 SEMANTIC PATTERNS

As outlined above, the algorithm for finding semantic patches starts with finding common patterns. Concretely, we look for *semantic patterns*. The grammar of semantic patterns can be given as the following subset of the grammar of semantic patches:

$$\begin{array}{lcl} pt & ::= & \epsilon \mid \mathrm{O}\ pt \\ \mathrm{O} & ::= & PT \mid \ldots \\ PT & ::= & \text{meta-variable} \mid \text{wildcard} \mid \textsc{atom} \mid \mathfrak{a}(PT^*) \end{array}$$

As can be seen, a semantic pattern is a semantic patch that contains no +'s or -'s. We will therefore refer to the left-hand side of a semantic patch as the semantic pattern obtained by removing all -'s and all +PT's from the semantic patch.

In the following we define a pattern containment relation used by pattern finding algorithm to prune the search space and reduce the number of returned patterns. In order to define the subpattern relation, we first define when a patch term pt_1 is *contained* in another patch term pt_2 denoted as $\mathrm{pt}_1 \trianglelefteq \mathrm{pt}_2$. This is the case if either:

1. $\mathrm{pt}_1 = \mathrm{pt}_2$, or
2. $\mathrm{pt}_2 = \_$, or
3. $\mathrm{pt}_1 = \mathrm{c}_1(\mathrm{pt}_1, \ldots, \mathrm{pt}_n)$ and $\mathrm{c}_2(\mathrm{pt}'_1, \ldots, \mathrm{pt}'_n)$ and $\mathrm{c}_1 = \mathrm{c}_2$ and $\forall 1 \leqslant i \leqslant n : \mathrm{pt}_i \trianglelefteq \mathrm{pt}'_i$.

6.1 Example Let the following three terms be given:

$$\begin{array}{lcl} \mathrm{t}_1 & = & \mathrm{m}(\mathrm{f}(42), \mathrm{f}(\_)) \\ \mathrm{t}_2 & = & \mathrm{m}(\mathrm{f}(\_), \mathrm{f}(100)) \\ \mathrm{t}_3 & = & \mathrm{m}(\mathrm{f}(\_), \mathrm{f}(\_)) \end{array}$$

One can verify that $\mathrm{t}_1 \trianglelefteq \mathrm{t}_3$ and $\mathrm{t}_2 \trianglelefteq \mathrm{t}_3$ and that no other containments hold.

The subpattern relationship is defined in Definition 6.2. Note that the subpattern definition provides a containment relationship between semantic patterns that can be checked by only considering the syntactic structure of the semantic patterns while the previous subpatch definition provides a containment relationship between semantic patches that requires consideration of the result of applying the semantic patches to specific terms.

6.2 Definition (**Subpattern**) *A semantic pattern* $\mathrm{p} = \langle \mathrm{e}_1, \cdots, \mathrm{e}_n \rangle$ *is a subpattern of another semantic pattern* $\mathrm{p}' = \langle \mathrm{e}'_1, \cdots, \mathrm{e}'_m \rangle$ *if and only* $\langle \mathrm{e}_1, \cdots, \mathrm{e}_n \rangle$ *is a subsequence of* $\langle \mathrm{e}'_1, \cdots, \mathrm{e}'_m \rangle$ *in which the corresponding elements are compared using the patch-term containment relation defined above. Formally,* p *is a subpattern of* p' *if there exists integers* $1 \leqslant i_1 < i_2 < \cdots < i_n \leqslant m$ *where* $\mathrm{e}_1 \trianglelefteq \mathrm{e}'_{i_1}$, $\mathrm{e}_2 \trianglelefteq \mathrm{e}'_{i_2}$, $\cdots$, $\mathrm{e}_n \trianglelefteq \mathrm{e}'_{i_n}$. *We denote this as* $\mathrm{p} \sqsubseteq \mathrm{p}'$.

6.3 *Example* Given the following three semantic patterns

$$\begin{aligned} p_1 &= \langle m(10,2), g(1)\rangle \\ p_2 &= \langle m(\_,\_), g(\_)\rangle \\ p_3 &= \langle f(\_), m(\_,\_), g(1)\rangle \end{aligned}$$

one can verify that $p_1 \sqsubseteq p_3$ and $p_1 \sqsubseteq p_2$, but not $p_2 \sqsubseteq p_3$ because $g(\_) \trianglelefteq g(1)$ does not hold.

6.4 FINDING SEMANTIC PATTERNS

We now present our algorithm for finding maximal semantic patterns. First, however, we describe the properties that we use to prune the search space.

6.4.1 *Occurrences & Pruning Properties*

A semantic pattern can match zero or more fragments of code in the left-hand side of a change set. We define the *occurrences* of a semantic pattern p with respect to a change set as follows. Let $p \vdash g$ denote that semantic pattern p matches the control-flow graph g and let $CFG(t)$ denote the control-flow graph of the term t. Then

$$Occ_{CS}(p) = \{t \mid (t, t') \in CS, CFG(t) = g, p \vdash g\}$$

The number of occurrences of a semantic pattern is the size of the set $Occ_{CS}(p)$. We elide the CS on $Occ_{CS}(p)$ when it is clear from the context. A pattern is *frequent* if the number of occurrences is larger than a user-defined minimum occurrence threshold *th*. Occurrences of a pattern follow some properties described in Properties 6.4 and 6.5 below.

6.4 Property (**Anti-monotonicity of occurrences)** Consider two patterns p_1 and p_2 and a change set CS. If $p_2 = p_1 \ldots evs$ where *evs* is also a semantic pattern, the number of occurrences of p_1 is greater or equal than that of p_2. ∎

6.5 Property (**Pattern occurrence inference)** Consider two patterns p_1 and p_2 where each patch term may contain wildcards, but not meta-variables, and a change-set CS. If p_1 and p_2 have the same set of occurrences in CS, then for any pattern *evs*, the patterns $p_1 \ldots evs$ and $p_2 \ldots evs$ have the same set of occurrences. ∎

The algorithm described below uses the above properties to prune the search space when looking for patterns with a significant number of occurrences.

6.4.2 *Algorithm*

Our approach takes as input the change set *CS* and a minimum occurrence threshold *th*. The goal is to extract all maximal patterns that are frequent. Our approach is outlined in Algorithm 1, in the procedure `Find Patterns`, below.

The semantic patterns found by Algorithm 1 all conform to the following regular expression: $PT(\ldots PT)^*$. We use $\langle p_1, \cdots, p_n \rangle$ as a short-hand notation for such patterns.

There might be many semantic patterns that are frequent in a given change set. These patterns may be overlapping and some might not match any node in the control-flow that was changed. Therefore, we find the *maximal* patterns according to the semantic pattern containment relation. There are fewer maximal patterns and by looking for maximal patterns, we can further employ a search-space pruning technique (defined in Section 6.4.1).

The algorithm proceeds in three steps: a) Infer semantic patterns where each *PT* of the pattern may contain wildcards, but not meta-variables, starting from length 1 and increasing to longer lengths (lines 1–4), b) Remove non-maximal patterns (line 5), and c) Infer meta-variable bindings on maximal patterns (line 6).

Our approach starts by first finding patterns of length 1 that are frequent in *CS* (line 1). We then grow each length 1 pattern to form longer patterns using `Grow` (line 4). The procedure `Grow` recursively grows the patterns by appending new patch terms one by one.

Several pruning strategies are employed to stop the growth process. These are based on the properties presented in Section 6.4.1. Based on Property 6.4, if a pattern does not have sufficient occurrences, there is no need to grow it further as its extension would also have insufficient occurrences. This check is made at line 10. Also, there is no need to grow a pattern if its super-pattern having the same set of instances has been considered before. This is the case as Property 6.5 dictates that all extension of the sub-pattern would have the same set of occurrences as the corresponding extension of the super-pattern. Since we are only interested in finding the maximal patterns, there is no need to keep the sub-pattern. This pruning prevents an exponential growth in the number of patterns, as would occur if all subsequences of a pattern with a sufficient number of occurrences were also occurring frequently.

For every pattern that is not pruned by the properties, `Grow` tries to grow the pattern further by making a recursive call (lines 12-15). If the pattern cannot be grown further, then we add this pattern to the candidate pattern set found so far, `R` (lines 15-16). The procedure `Grow` returns a boolean value indicating whether `R` has been updated.

Because `Grow` only tries to extend patterns at the end, the final value of `R` may contain some patterns that are not maximal and that are indeed suffixes of other, maximal, elements of `R`. We remove these non-maximal patterns at line 5. For each remaining pattern in the result R_{max}, we infer its meta-variable bindings, based on the change set *CS* (lines 6-7). The algorithm looks for the strongest bindings, i.e., the ones that convert as many wildcards as possible to the smallest number of different meta-variables, that assign a unique variable name for patch terms that always map to the same terms in the change set *CS* under consideration. The set of maximal patterns with meta-variable bindings is the output of `Find Patterns` and is fed to the next step: patch construction, described in the next section.

6.4.3 *Constructing semantic patches*

To construct patches from patterns, we first collect the individual modifications in the code, in a format that we refer to as *chunks*. Given two terms t and t′, we have implemented a simple graph differencing tool[2] that produces a list of term additions, removals, or contextual terms. From this list we construct chunks, as defined in Definition 6.6 below.

6.6 *Definition* (**Chunks**) *A chunk is a sequence of term additions, term removals, and contextual terms, as described by the following regular expression:*

$$(\texttt{+t})^*(\texttt{t} \mid \texttt{-t})(\texttt{+t}^*)$$

A chunk contains a unique term removal or contextual term, referred to as the context point. Given a chunk ch *we denote its context point as* ch.cp*. This context point* cp *is matched by a patch term pt iff there is a binding of the term pattern meta-variables resulting in the context point. We denote this as* $\texttt{cp} \mapsto \texttt{pt}$.

To make the description of the patch construction algorithm simpler, we introduce an operation *PairChunk*. This operation takes a patch *pt*, a term pattern *tp* within *pt*, and a chunk ch. It incorporates the transformation operations within *ch* into *tp*, and generates meta-variable bindings for the transformation operations within ch, resulting in a new pattern. As a simple example, let *pt* be the patch "`-a(X); ... b(X);`" and ch the chunk "`-a(42); +a(42,42)`". The result of *PairChunk*(*pt*, `-a(X)`, ch) is then the semantic patch `-a(X);+a(X,X); ... b(X);`. Notice, that the `X` in `-a(X);` has been matched with the `42` of `-a(42);` so that `42` is replaced by `X` in `+a(42,42)`.

The algorithm to construct patches from patterns is shown in Algorithm 2. It first constructs a chunk set *CHS* from the input change set *CS* (line 1). We process each pattern one by one by adding it to the working queue (line 3). For each pattern, we try to map each term pattern within it with each corresponding chunk *ch* in *CHS* (lines 6-8). If this succeeds, we create a new patch p′ by incorporating the transformation operations in the chunk ch into the current pattern (patch) under consideration p (line 9). If this patch does not appear in the output set Out and is a safe transformation with respect to *CS*, we add this patch to the output set and the working queue (lines 10-12). At line 13, non-maximal patches are removed. Maximal patches are output at line 14.

In practice the running time of the algorithm depends the structure of the programs given in the change set. If the programs have very little in common except for the locations related to the common change made, the algorithm is faster than if the programs have much in common.

2 One can use an off-the-shelf tree-differencing algorithm, sequence differencing, or graph-differencing tool.

6.5 IMPLEMENTATION

All of the algorithms mentioned have been implemented in a tool called `spdiff` using the O'Caml programming language in roughly 10.000 lines of code for (excluding the parser which is again the same as the one used in Coccinelle). In the implementation two further issues have been addressed. The first issue is the problem of noise in the input data and the second is a performance issue.

NOISY INPUT DATA We have so far assumed that the given changesets C are "ideal" in the sense that it is possible to find a patch that is safe for *all* pairs in C. When applying our tool to updates in Linux device drivers we found that this was often not the case. Rather, one patch was found safe for a number of pairs while another patch was found safe for a different set of pairs. Our tool would then say that it could find no globally safe update.

To relax the requirement that all changes have to be safe for all pairs we introduced a user-specified *threshold*. The threshold states for how many pairs a change has to be safe before it is considered safe for the given changeset C. In Section 4.5 we introduced thresholds to the definitions of transformation parts and the subsumption relation which have been implemented in our tool so that the user can specify a desired threshold. One can show that Corollary 5.9 also holds when thresholds are introduced, so the refined algorithm also works when thresholds are involved. Allowing a user given threshold is essential when dealing with input that is either buggy or expresses a collection of several incompatible collateral evolutions.

IMPROVING PERFORMANCE In the definitions given above that relate program transformations or terms, we make frequent use of equality checks on terms. A simple implementation of term equality would compare the two given terms structurally:

```
let t_equal t1 t2 = case (t1, t2) of
            | a1, a2 -> a1 == a2
            | a1(ts1), a2(ts2) ->
                 a1 == a2 && |ts1| == |ts2| &&
                 ∀ t1∈ts1, t2∈ts2: t_equal t1 t2
```

Clearly, the time complexity of `t_equal` is $\mathcal{O}(n)$ where n is the size the smaller of the two terms. Our initial implementation used the above equality check and it was not very fast. Pointer-equality is more efficient to compute than structural equality. Concretely, we have made use of a technique called *hash-consing* (or *value numbering*) as implemented in the module `Hashcons` by Conchon and Filliatre [16]. Their implementation of hashconsing works very well in our setting and allows constant time equality checks.

Procedure: Find Patterns
Inputs: *CS*: Change set
th: Minimum number of occurrences
Output: All maximal patterns obeying th

1Let P_1 = patterns without any meta-variables
with length 1 appearing $\geqslant$ th in *CS*
2: Let $R = \varnothing$
3: For each $p \in P_1$:
4: Call Grow (p, P_1, R, th, *CS*)
5: Let $R_{max} = \{p \mid p \in R \wedge (\neg \exists p' \in (R \backslash \{p\}) : p \sqsubseteq p')\}$
6: For each $p \in R_{max}$:
7: Infer strongest bindings for meta-variables in p
over all occurrences of p in *CS*.
8: Output R_{max}

Procedure: Grow
Inputs: p: Pattern being grown
P_1: Patterns of length 1
R: Candidate patterns found so far
th: Min. number of occurrences
CS: Change set
Output: Boolean value indicating whether
R is updated with new patterns

9: Let $gf = false$
10: If p matches $\geqslant$ th times in *CS*
11: If $\neg \exists p' \in R.\ p \sqsubseteq p' \wedge Occ(p) = Occ\ (p')$
12: For each $e \in P_1$:
13: Let $np = p\ \ldots\ e$
14: $gf = gf$ || Call Grow (*np*, P_1, R, th, *CS*)
15: If $\neg gf$
16: $R = R \cup \{p\}$
17: $gf = true$
18: Return *gf*

Algorithm 1: Finding maximal semantic patterns

Procedure: Construct Patches
Inputs: *PS*: semantic patterns
CS: Change set
Output: Patches corresponding to *PS*

1: Let *CHS* = all chunks from *CS*, obtained using `diff`
2: Let $Out = \emptyset$
3: Let *WorkQ* = *PS*
4: While *WorkQ* is not empty:
5: $\quad p = WorkQ.pop()$
6: $\quad$ For each term pattern $tp \in p$
not paired to a chunk
7: $\quad\quad$ For each chunk $ch \in CHS$
8: $\quad\quad\quad$ If $tp \mapsto ch.cpt$
9: $\quad\quad\quad\quad$ Let $p' = PairChunk(p, tp, ch)$
10: $\quad\quad\quad\quad$ If $p' \notin Out \wedge \forall cp \in CS : p' \preceq cp$
11: $\quad\quad\quad\quad\quad$ $WorkQ.push(p')$
12: $\quad\quad\quad\quad\quad$ $Out = Out \cup \{p'\}$
13: Remove non-maximal patches from *Out*
14: Output *Out*

Algorithm 2: Construct patches from patterns

Part III

REAL-WORLD APPLICATION

Chapter 7

7.1 EXAMPLES OF CONTEXT-FREE PATCHES

We now provide a few examples of the use of `spdiff`, based on some recent patches committed to Linux that we have identified using the `patchparse` collateral evolution mining tool [48]. For each standard patch that we have tested, we have constructed the set of pairs of terms, C, from the image of the Linux source tree just before the standard patch was applied and just after.

ADAPT TO STRUCTURE CHANGES The following commits, dated November 9, 2007, begin with the log message "convert to use the new `SPROM` structure".[1]

```
95de2841aad971867851b59c0c5253ecc2e19832
458414b2e3d9dd7ee4510d18c119a7ccd3b43ec5
7797aa384870e3bb5bfd3b6a0eae61e7c7a4c993
```

These commits comprise over 650 lines of patch code, and affect 12 files in the `drivers/net` directory or its subdirectories, at 96 locations. In the role of an expert in the affected files, we selected three files from the first commit that illustrate the set of required changes. From these files, `spdiff` infers the following context-free patch:

```
X0->sprom.r1 ⇝ X0->sprom ;
sprom->r1.X0 ⇝ sprom->X0
```

The inferred context-free patch fully updates all 12 original files in the same way the standard patches did. By careful examination of the standard patch, a person could construct the inferred context-free patch by hand. However, there would be no guarantee that the constructed patch is safe, as this is not evident in the standard patch. To check safety manually, one would have to consider 1) whether the constructed patch updates the proper locations correctly but does not update locations that were not to be modified, and 2) whether the constructed patch is only a part of the update that is to be performed to a particular file.

Furthermore, the inferred context-free patch updates some other files that were present at the time of the original patches but were overlooked. These files were in other directories and were not updated until February 18, 2008, by another developer.

1 The patches can be obtained from http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

STRUCTURE CHANGES Commit c32c2f63a9d6c953aaf168c0b2551da9734f76d2 from February 14, 2008 affects 9 files at 12 locations. The message attached to the commit is "d_path: Make seq_path() use a struct path argument". The standard patch attached to the commit is approximately 160 lines. The patch inferred by spdiff is:

```
seq_path(X1,X2->X3.mnt,X2->X3.dentry,X4)
  ⇝ seq_path(X1,&X2->X3,X4)
```

The inferred context-free patch fully updates all but one of the original files. The only file that is not fully updated is the file fs/namespace.c in which a declaration struct path mnt_path is also added.

RENAMING OF FUNCTION CALLS The following commits, dated December 20, 2007, begin with some variant of the log message "Kobject: convert drivers/* from kobject_unregister() to kobject_put()".

```
c10997f6575f476ff38442fa18fd4a0d80345f9d
78a2d906b40fe530ea800c1e873bfe8f02326f1e
197b12d6796a3bca187f22a8978a33d51e2bcd79
38a382ae5dd4f4d04e3046816b0a41836094e538
```

These commits comprise almost 800 lines of patch code, and affect 35 files at 79 locations. Based on the changes in the 17 files in the first of the above commits, spdiff derives the following context-free patch:

```
kobject_unregister(X0)⇝ kobject_put(X0)
```

The inferred context-free patch fully updates all but 3 files in the same way the standard patch did. The remaining files each include an additional change that goes beyond the collateral evolution.

MODIFYING DECLARATIONS Commit c11ca97ee9d2ed593ab7b5523def7787b46f398f and 12 others from around December 7, 2007 change 21 files at 26 locations. The log messages are "use LIST_HEAD instead of LIST_HEAD_INIT". The standard patches total almost 300 lines. The inferred context-free patch is:

```
struct list_head X0 = LIST_HEAD_INIT(X0);
  ⇝ LIST_HEAD(X0);
```

The inferred context-free patch fully updates all 21 files. The original developer, on the other hand, initially overlooked one case and had to create a second patch on the same file to correct it. Furthermore, 6 files that contained relevant declarations at the time the patches were committed were not updated by the original patches, and of those 5 files were still not updated as of several months later. All of these files are fully updated by the context-free patch.

USE `kzalloc` Over the past couple of years, around 100 patches have been committed that convert the combination of calls to `kmalloc` and `memset` to `kzalloc`. One such commit, from September 6, 2005 is `dd3927105b6f65afb7dac17682172cdfb86d3f00` which affected 6 files at 27 locations. The transformation it performs can be represented as follows.

```
x = kmalloc(size, flags);
...                          ⇝   kzalloc(x, size, flags);
memset(x, 0, size);              ...
```

Our tool is, however, not able to infer any safe context-free patch in this case because the language of context-free patches is not able to express the temporal ordering of terms or the sharing of meta-variables between disjoint code fragments.

ASSESSMENT These examples show that for a variety of collateral evolutions, `spdiff` infers context-free patches that are much more concise, and we believe much more readable, than the corresponding standard patches. In several cases, the original standard patches did not perform part of the collateral evolution in some relevant files. In this situation, a developer could use `spdiff` to infer a context-free patch from the provided standard patches to help complete the collateral evolution. Using the Coccinelle transformation system, the context-free patch can be applied everywhere in the Linux source tree.

While all of the inferred context-free patches inferred are simple enough that a person could construct them by hand by inspecting the standard patches, it would require more work to confirm that the manually constructed patch is indeed safe for all of the input files. Safety is not evident from the standard patches which contain only the code that was changed, not any similarly structured code that was *not* changed. In order to confirm safety, one would need to apply the constructed patch to all original input files and check that for each file, the constructed patch applies *correctly* to a *subset* of the locations that need to be modified in the file.

Our final example illustrates a limitation of context-free patches. The richer language of context-sensitive patches can express the properties needed to treat such examples.

7.2 EXAMPLES OF CONTEXT-SENSITIVE PATCHES

We now revisit the motivating example given in Section 6.1. Our goal is to show how `spdiff` can be used to obtain a high-level, but operational, description of the common changes in a changset, and how this result can help developers avoid potential bugs in the changes made or adapt their own code in accordance with the common changes found in the changset.

THE ORIGINAL COMMIT The commit with `git` SHA1 identification code 01f2705daf5a3620-8e69d7cf95db9c330f843af6 was the first in a series of standard patches committed starting in May 2007 to the Linux version control system that implemented a refactoring of a commonly occurring pattern for clearing pages found in filesystem code. This pattern consisted of the following operations: 1) map a page into the kernel virtual memory using `kmap_atomic`, 2)

clear this page using `memset`, 3) call `flush_dcache_page` to ensure that the cleared memory gets written to disk, and 4) unmap the memory region using `kunmap_atomic`. The refactoring introduced a new function, `zero_user_page` that does all of these operations. Core kernel locations where memory was cleared in this way were modified to use the new function. In subsequent commits, the remaining locations were updated to use the new function. These latter changes amount to collateral evolutions.

THE INFERRED SEMANTIC PATCH Suppose now that the changset that we use as input to `spdiff` consists of a subset of 8 pairs of the original and updated versions of some function that was modified by the above commit. If `spdiff` is run with the changset and a minimum support threshold of 3 we get the semantic patch already shown in Section 6.1:

```
@@
  expression X0, X1;
  struct page *X2;
  char *X3;
@@
- X3 = kmap_atomic(X2, KM_USER0);
  ...
- memset(X3 + X0, 0, X1);
  ...
- kunmap_atomic(X3, KM_USER0);
+ zero_user_page(X2, X0, X1, KM_USER0);
```

Note that the generated semantic patch does not include the call to `flush_dcache_page`. As it occurs in varying positions, i.e., sometimes before and sometimes after the call to `kunmap_atomic`, it is not included in the initial semantic pattern. Furthermore, in one case the call was not removed.

Applying this semantic patch to the files mentioned in the original commit causes it to perform a safe part of the changes that were made by hand, in all but one case. This case is represented by the following excerpt of the standard patch:

```
@@ -2108,10 +2100,8 @@ int cont_prepare_write(

- kaddr = kmap_atomic(new_page, KM_USER0);
- memset(kaddr+zerofrom, 0, PAGE_CACHE_SIZE-zerofrom);
- flush_dcache_page(new_page);
- kunmap_atomic(kaddr, KM_USER0);
+ zero_user_page(page, zerofrom, PAGE_CACHE_SIZE - zerofrom,
+                KM_USER0);
```

This standard patch code contains a small bug, that is not detected at compile time and thus only manifests itself at run-time. The error is that in the added code at the end of the standard patch, the first argument to `zero_user_page` is `page`, while, as shown by the other calls, it should have been `new_page`. The updated function can still be compiled because the variable `page` is a

parameter of the function being modified. At run-time, however a file system corruption occurs, as described in the log message associated with commit ff1be9ad61e3e17ba83702d8ed0b534e5b-8ee15c.

This error would not have happened if the change were made using the semantic patch, because the semantic patch specifies that the first argument to the newly inserted function call should be the same as the first argument to the call to `kmap_atomic`. Since in *all* other updated functions the name of the variable given as the first argument to `kmap_atomic` is indeed `page`, it seems like the bug is the result of a copy-paste error. Linux code frequently but not always uses stereotypical names for values of a given type, and thus there is a high potential for this sort of error.

The bug was eventually fixed 11 days later by a different developer.

SUBSEQUENT CHANGES On February 4, 2008 all of the calls to the function introduced 8 months earlier, `zero_user_page`, were eventually replaced with a call to one of three different functions: 1) `zero_user`, 2) `zero_user_segment`, and 3) `zero_user_segments`. 39 functions spread over 22 files were updated as part of the commit.[2] When applying `spdiff` to the changset constructed from the modified functions with a minimum support threshold of 11, two semantic patches are returned:

```
@@ struct page *X0;
   expression X1, X2;
@@
- zero_user_page(X0,X1,X2 - X1,KM_USER0);
+ zero_user_segment(X0,X1,X2);
```

and

```
@@ expression X0, X1;
   unsigned int X2;
@@
- zero_user_page(X0,X1,X2,KM_USER0);
+ zero_user(X0,X1,X2);
```

`spdiff` detects that it is unsafe to apply the second before the first, as a subtraction expression can also have type unsigned int, and thus it indicates that they must be applied in the specified order. After application of the two inferred semantic patches, three calls to the `zero_user_page` remain to be updated. In those cases, `spdiff` was not able to infer any common patterns and we have indeed verified that it is not possible for one semantic patch to update all three locations.

This shows the value of automatic inference of patches to: 1) Reduce the need for manual code updating that could be error prone, 2) Find anomalies in patches and provide feedback to programmers to aid in patch and collateral evolution understanding.

2 SHA1 identification code: `eebd2aa355692afaf9906f62118620f1a1c19dbb`

CONCLUSION

Chapter 8

In this chapter we summarize the contributions of this dissertation and give directions for future work.

8.1 SUMMARY

The main hypothesis that have been explored in this dissertation is that it is feasible and helpful to infer high-level descriptions of common changes from a representative set of changes (the changeset).

Concretely we have contributed with an abstraction definition of the notion of transformation parts. Transformations parts capture when a specification of changes is witnessed in a changeset. The definition of transformation parts is parametrized by an application function for the underlying change specification language. Furthermore, we have presented two algorithms for finding increasingly more expressive program transformations. Both of the algorithms have been implemented in the programming language OCaml. The implementations have been applied to real changesets found in the Linux repository. We have shown that for the examples tried, the inferred transformations are much more compact than what is available in the current version control system used in the development of Linux.

8.2 FUTURE WORK

The future work based on this dissertation falls in two categories: 1. evaluation and engineering work, and 2. exploration of other transformation languages.

8.2.1 *Evaluation and engineering*

The work presented in this dissertation does not include extensive evaluation of the algorithms implemented. In order to do an evaluation of our approach we would apply `spdiff` to a larger set of changes. Also, it would be interesting to apply our approach to other projects than Linux. Such an evaluation would be useful to make a more general statement about the usefulness of our approach.

In terms of the implementation it could be helpful if the user could explore the information that `spdiff` collects about the changesets. Currently, `spdiff` simply returns a list of the program transformations it inferred, but it could be useful for the user to then select a transformation and

ask whether subparts of it could be made more general–recall that we only abstract subparts if it is needed for the pattern to match enough terms. Also, in the current implementation the user can select a threshold and start the inference, and if no good results are returned, the user can try to rerun the inference with a lower threshold. However, this process could be made more efficient if the tool somehow kept a record of previous runs. Likewise, incremental analysis is not well supported by our implementation. One could imagine the user initially providing a small changeset and then when the transformation inferred seems too specific or general, add more pairs to the current changeset. In such cases the results of previous runs of the tool could be used to incrementally compute the results for the new changeset.

8.2.2 *Exploration of other transformation languages*

In this dissertation we have presented algorithms for finding common changes relative to a context-free term-rewrite language and a context-sensitive transformation language. A natural thing to do is to instantiate our transformation part framework with respect to other transformation languages–or simply to extend the presented ones to be more expressive.

CONTEXT-RESTRICTED REWRITES One suggestion is to extend the context-free rewrite language with a limited form of context-sensitivity which we call *context-restriction*. Concretely, a context-restricted term-rewrite rule rewrites all occurrences of the left-hand side of the rule within a specific context. An example context could be "all functions taking two parameters of a certain type with a specific return type". The rewrite rule would then only consider rewriting subterms within the restricted context and safety of the rule would only need to be relative to the restricted context because elsewhere the rewrite rule does not apply. The consequence is that the $p \rightsquigarrow p'$ rule could be more general and still safe. A very concrete example could be a rule such as `return 0;` $\rightsquigarrow$ `return SOMECONST;` which would change all `return 0`'s into a statement returning the constant `SOMECONST`. In general this rewrite rule should not be applied everywhere in all functions, but it may be the case that *within* functions that return a certain user-defined type, it is actually safe to apply the rewrite rule.

MORE EXPRESSIVE TRANSFORMATION LANGUAGE Recall the inferred context-free patches from Example 4.14 : $pt_1 = a \rightsquigarrow aa; b \rightsquigarrow bb$ and $pt_2 = a \rightsquigarrow aa; c \rightsquigarrow cc$. In the example we saw that the set $LCT(CS)$ contained two elements when the threshold was set to 2 $\{pt_1, pt_2\}$. One can see that the two context-free patches share the prefix which rewrites a into aa, but this sharing is not expressed in the inferred patches. We could consider, and indeed the full SmPL language supports it, allowing disjunctive transformations so that the two context-free patches could be expressed in one

$$pt_{1\cup 2} = a \rightsquigarrow aa; (b \rightsquigarrow bb \mid c \rightsquigarrow cc)$$

The above disjunctive patch specifies that first $a \rightsquigarrow aa$ must be applied and then, depending on which rule matches, either $b \rightsquigarrow bb$ or $c \rightsquigarrow cc$ should be applied. The benefit of the above

specification is that the shared part is directly evident from the patch rather than by inspection of all the inferred transformations.

COMMON PROGRAM TRANSFORMATION INFERENCE TOOL GENERATOR A final direction is to make our approach applicable to other languages than C. In our current approach we already translate the C code to an internal representation (TERMS) and all inference is performed on this internal representation. We could then make use of parser generator technology to be able to instantiate our approach to any language for which a grammar exists. In particular generation of a tool for inference of context-free patches is straight-forward given a grammar for the subject language.

BIBLIOGRAPHY

[1] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6910-1. (Cited on page 20.)

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. pages 580–592, 1998. (Cited on page 12.)

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, 2007. ISSN 0928-8910. doi: http://dx.doi.org/10.1007/s10515-006-0002-0. (Cited on page 32.)

[4] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007. ISBN 193435600X. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/193435600X`. (Cited on page 19.)

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7111-4. (Cited on pages 15 and 16.)

[6] Brenda S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, March 1992. (Cited on page 15.)

[7] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997. ISSN 0097-5397. doi: http://dx.doi.org/10.1137/S0097539793246707. (Cited on page 15.)

[8] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8779-7. (Cited on page 19.)

[9] Stefan Bellon. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2007.70725. Member-Koschke, Rainer and Member-Antoniol, Giulio and Member-Krinke, Jens and Member-Merlo, Ettore. (Cited on page 14.)

[10] Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 76(1-2):7–13, 2000. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/S0020-0190(00)00124-1. (Cited on page 26.)

[11] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Savannah, Georgia, January 2009. (Cited on pages 10 and 75.)

[12] Peter Bulchev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *3rd International Workshop on Software Clones at CSMR'2009*, March 2009. (Cited on page 22.)

[13] Peter Bulychev and Marius Minea. Duplicate code detection using anti-unification. In *Spring Young Researchers Colloquium on Software Engineering, SYRCoSE*, volume 2008, 2008. (Cited on page 22.)

[14] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4. doi: http://doi.acm.org/10.1145/233269.233366. (Cited on pages 28 and 29.)

[15] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7677-9. (Cited on pages 9 and 35.)

[16] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006. URL `http://www.lri.fr/~filliatr/ftp/publis/hash-consing2.ps`. (Cited on page 80.)

[17] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, New York, NY, USA, 2004. ACM. ISBN 1-58113-885-7. doi: http://doi.acm.org/10.1145/997817.997857. (Cited on page 21.)

[18] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006. ISBN 3-540-35726-2. (Cited on page 35.)

[19] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: http://doi.acm.org/10.1145/502034.502041. (Cited on page 11.)

[20] William S. Evans, Christopher W. Fraser, and Fei Ma. Clone detection via structural abstraction. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 150–159, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3034-6. doi: http://dx.doi.org/10.1109/WCRE.2007.15. (Cited on pages 14 and 21.)

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/24039.24041. (Cited on page 14.)

[22] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2601-2. doi: http://dx.doi.org/10.1109/ICPC.2006.16. (Cited on page 30.)

[23] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70731. (Cited on page 29.)

[24] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368132. (Cited on page 23.)

[25] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062512. (Cited on pages 34 and 35.)

[26] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976. (Cited on page 24.)

[27] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-6330-8. (Cited on page 31.)

[28] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.30. (Cited on pages 14 and 21.)

[29] Neil Jones and René Rydhof Hansen. The semantics of "semantic patches" in Coccinelle: Program transformation for the working programmer. In *Fifth ASIAN Symposium on*

Programming Languages and Systems, number 4807 in Lecture Notes in Computer Science, pages 303–318, Singapore, November 2007. (Cited on page 75.)

[30] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7): 654–670, 2002. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2002.1019480. (Cited on page 16.)

[31] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: http://dx.doi.org/10.1109/ICSE.2009.5070531. (Cited on page 34.)

[32] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.20. (Cited on pages 7 and 34.)

[33] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag. ISBN 3-540-42314-1. (Cited on page 22.)

[34] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, M. Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Autom. Softw. Eng.*, 3(1/2):77–108, 1996. (Cited on page 19.)

[35] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1. doi: http://dx.doi.org/10.1109/WCRE.2006.18. (Cited on page 18.)

[36] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1303-4. (Cited on page 23.)

[37] E.L. Lehmann and J.P. Romano. *Testing statistical hypotheses*. Springer Verlag, 2005. (Cited on page 23.)

[38] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966. (Cited on page 29.)

[39] Huiqing Li and Simon Thompson. Clone detection and removal for erlang/otp within a refactoring environment. In Germán Puebla and Germán Vidal, editors, *PEPM*, pages 169–178. ACM, 2009. ISBN 978-1-60558-327-3. (Cited on pages 14 and 19.)

[40] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association. (Cited on page 20.)

[41] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. doi: http://doi.acm.org/10.1145/1150402.1150522. (Cited on page 23.)

[42] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1095430.1081754. (Cited on page 12.)

[43] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, January 2003. Unified Format section, `http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html`. (Cited on pages 2 and 24.)

[44] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23 (2):262–272, 1976. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321941.321946. (Cited on page 15.)

[45] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2): 251–266, 1986. (Cited on page 24.)

[46] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-123-6. doi: http://doi.acm.org/10.1145/1083142.1083143. (Cited on page 30.)

[47] Yoann Padioleau. Parsing c/c++ code without pre-processing. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 109–125, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00721-7. doi: http://dx.doi.org/10.1007/978-3-642-00722-4_9. (Cited on page 40.)

[48] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, Leuven, Belgium, April 2006. (Cited on pages 2, 40, and 84.)

[49] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Eurosys 2008*, pages 247–260, Glasgow, Scotland, March 2008. (Cited on pages 2, 3, and 75.)

[50] G.D. Plotkin. A note on inductive generalization. *Machine intelligence*, 5(153-163):178, 1970. (Cited on pages 22 and 42.)

[51] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–, 2002. (Cited on page 16.)

[52] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0. (Cited on page 31.)

[53] Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995. (Cited on page 42.)

[54] Henrik Stuart, Rene Rydhof Hansen, Julia Lawall, Jesper Andersen, Yoann Padioleau, and Gilles Muller. Towards easing the diagnosis of bugs in OS code. In *4th Workshop on Programming Languages and Operating Systems (PLOS 2007)*, Stevenson, Washington, USA, October 2007. (Cited on page 10.)

[55] E. Visser. Program transformation with stratego/XT rules, strategies, tools, and systems in stratego/XT 0. 9. *Lecture notes in computer science*, pages 216–238, 2004. (Cited on page 10.)

[56] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. (Cited on page 22.)

[57] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: http://dx.doi.org/10.1109/ASE.2006.41. (Cited on page 34.)

[58] Michael J. Wise. Running Karp-Rabin matching and greedy string tiling. Technical report, Basser Department of Computer Science Technical Report, Sydney University, March 1993. (Cited on page 17.)

[59] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101919. (Cited on page 35.)

[60] Zhenchang Xing and Eleni Stroulia. API-Evolution support with diff-catchup. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2007.70747. (Cited on page 35.)

[61] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134325. (Cited on page 11.)

[62] Wuu Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21 (7):739–755, 1991. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380210706. (Cited on pages 27 and 51.)