GOETHE UNIVERSITY

Combining User-interaction and Automation to evolve Source code

DISSERTATION

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik

der Johann Wolfgang Goethe-Universitat

in Frankfurt am Main

von

Krishna Narasimhan

aus

Chennai, Indien

Frankfurt

2017

D30

Vom Fachbereich 12 der

Johann Wolfgang Goethe - Universitaet als Dissertation angenommen.

Dekan: Prof. Dr. Thorsten Theobald

Gutachter :
Dr. Christoph Reichenbach
Dr. Julia lawall

Datum der Disputation : 31-Januar-2017

Declaration of Authorship

I, Krishna Narasimhan, declare that this thesis titled, 'Combining user interaction and automation to evolve source code' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Education is what remains after one has forgotten everything he learned in school"

Albert Einstein

GOETHE UNIVERSITY

Abstract

Fachbereich Informatik und Mathematik SEPL Doctor of Philosophy

by Krishna Narasimhan

Software evolves. Developers and programmers manifest the needs that arise due to evolving software by making changes to the source code. While developers make such changes, reusing old code and rewriting existing code are inevitable. There are many challenges that a developer faces when manually reusing old code or rewriting existing code. Software tools and program transformation systems aid such reuse or rewriting of program source code. But there are significantly occuring development tasks that are hard to accomplish manually, where the current state-of-the-art tools are still not able to adequately automate these tasks. In this thesis, we discuss some of these unexplored challenges that a developer faces while reusing and rewriting program source code, the significance of such challenges, the existing automation support for these challenges and how we can improve upon them.

Reusing old code Modern software development relies on code reuse, which software developers typically realize through hand-written abstractions, such as functions, methods, or classes. However, such abstractions can be challenging to develop and maintain. An alternative form of reuse is *copy-paste-modify*, in which developers explicitly duplicate source code to adapt the duplicate for a new purpose. Copy-pasted code results in code clones, i.e., groups of code fragments that are similar to each other. Past research strongly suggests that copy-paste-modify is a popular technique among software developers. In this paper, we perform a small user study that shows that copy-paste-modify can be substantially faster to use than manual abstraction.

One might propose that software developers should forego hand-written abstractions in favour of copying and pasting. However, empirical evidence also shows that copy-paste-modify complicates software maintenance and increases the frequency of bugs. Furthermore, the developers in an informal poll we conducted strongly preferred to read code written using abstractions. To address the concern around copy-paste-modify, we propose a tool that merges similar pieces of code and automatically creates suitable abstractions. Our tool allows developers to get the best of both worlds: easy reuse together with custom abstractions. Because different kinds of abstractions may be beneficial in different contexts, our tool provides multiple abstraction mechanisms, which we selected based on a study of popular open-source repositories.

To demonstrate the feasibility of our approach, we have designed and implemented a prototype merging tool for C++ and evaluated our tool on a number of clones exhibiting some variation, i.e near clones, in popular Open Source packages. We observed that maintainers find our algorithmically created abstractions to be largely preferable to existing duplicated code.

Rewriting existing code Rewriting existing code can be considered as a form of program transformation, where a program in one form is transformed into a program in another form. One significant form of program transformation is data representation migration that involves changing the type of a particular data structure, and then updating all of the operations that has a control or data dependence on that data structure according to the new type. Changing the data representation can provide benefits such as improving efficiency and improving the quality of the computed results. Performing such a transformation is challenging, because

it requires applying data-type specific changes to code fragments that may be widely scattered throughout the source code connected by dataflow dependencies. Refactoring systems are typically sensitive to dataflow dependencies, but are not programmable with respect to the features of particular data types. Existing program transformation languages provide the needed flexibility, but do not concisely support reasoning about dataflow dependencies.

To address the needs of data representation migration, we propose a new approach to program transformation that relies on a notion of semantic dependency: every transformation step propagates the transformation process onward to code that somehow depends on the transformed code. Our approach provides a declarative transformation specification language, for expressing type-specific transformation rules. We further provide scoped rules, a mechanism for guiding rule application, and tags, a device for simple program analysis within our framework, to enable more powerful program transformations.

We have implemented a prototype transformation system based on these ideas for C and C++ code and evaluate it against three example specifications, including vectorization, transformation of integers to big integers, and transformation of array-of-structs data types to struct-of-arrays format. Our evaluation shows that our approach can improve program performance and the precision of the computed results, and that it scales to programs of at least 3700 lines.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Dr. Christoph Reichenbach for the continuous support of my Ph.D. study and research, for his patience, motivation, enthusiasm, and immense knowledge. I am very grateful for having had an advisor who more than just a mentor figure, but also a good friend.

Besides my advisor, I would like to mention special thanks to Dr. Julia Lawall from Inria, Paris for her support with the research, coming up with ideas to pursue and her detailed feedback on the papers we have written together and the thesis.

My sincere thanks also go to Dr. Visvanathan Ramesh and Dr. Andreas Zeller for their career and personal advice. I consider myself extremely lucky to have had access to such vastly experienced academics who are not only skilled at guiding students, but also keep themselves extremely accessible.

I thank my friends Bharath and Naresh for their support with my personal life during the last 3+ years. I would like to express my sincerest gratitude to Heinrich, Bayram, and Nisa who have provided the right suggestions at the right time on handling my professional career and the Ph.D.

I would like to express my sincerest thanks to my Fiance, Priya for being there with the ups and downs during the last few months, always reminding what is important at what point in life, and also for her invaluable effort in proofreading the thesis.

It is not the ideal scenario to be far away from home during a Ph.D. So, I would like to mention special thanks to the Grouvens (Radhika, Joachim, and Isha) for providing me a family away from home. I would like to mention special thanks to Radhika for not just being a mother, but also a friend, guide, and mentor in the most important aspects of life and more.

Last but not the least, I would like to thank my parents Uma and Murali, for giving birth to me at the first place and supporting my decisions in every stage of my life, without which who I am today will not be possible.

Contents

Declaration of Authorship ii

Abstract iv

Reusing old code v
Rewriting existing code v

Acknowledgements vii

List of Figures xii

List of Tables xiv

1 Introduction 1
The need for software evolution. 2
1.1 Software evolution and source code 2
1.2 Contribution 4
Cleaning up copy-pasted clones. 4
Data representation migration. 4
1.2.1 Source code reuse - Cleaning up copy paste clones with interactive merging 5
1.2.2 Software Reuse - Merging clone instances into existing abstractions 6
1.2.3 Source code rewriting - Data Representation Migration . . . 7

I Source code Reuse 9

2 Source code Reuse - Cleaning up copy-pasted clones with interactive merging 10
2.1 Resolution Patterns for Merging Method-Level Code Clones 12
2.2 User Study and Informal Poll 17
2.2.1 Benefits of Copy-Paste-Modify 18
2.2.2 Copy-Paste-Modify versus Manual Abstraction 19

2.3 Merging Algorithm . . . 20
2.3.1 Abstract Syntax Trees . . . 21
2.3.2 RTED . . . 22
Recalculating positions of nodes to accommodate insertions. . . . 23
Overview of the algorithm. . . . 23
2.3.3 Algorithm . . . 25
2.3.3.1 Identifying the conflict nodes . . . 26
2.3.3.2 Constructing the merge tree . . . 27
2.3.3.3 Applying resolution patterns . . . 28
Merge-Substitution. . . . 28
Fix Up. . . . 30
2.3.3.4 Pattern: Switch Statement with Extra Parameter . . . 31
2.3.3.5 Pattern: Extra Parameter for Literal Expressions . . . 33
2.3.3.6 Pattern: Templates for Type Expressions . . . 34
2.3.3.7 Pattern: Extra Parameter for Identifiers . . . 36
2.3.4 Optimizations . . . 38
2.4 Implementation . . . 40
Libraries and frameworks used in our implementation 40
Availability . . . 41
2.5 Informal discussion on behaviour preservation of the algorithm . . . 41
Merging existing parameters of near-clone methods . 43
2.6 Evaluation . . . 43
2.6.1 Identifying and Merging Clone Groups . . . 44
2.6.2 Results . . . 46
2.6.3 **Analysis of Rejected and Pending Results** . . . 48
2.6.3.1 Pending results . . . 48
2.6.3.2 Rejected results . . . 48
2.7 Full Repository Evaluation - GIT . . . 49

3 Source code Reuse - Merging clone instances back into existing abstractions 64
3.1 Motivation . . . 65
3.2 Algorithm . . . 69
Requirements for merging clone instances with an existing abstraction . . . 69
Summary of steps in CloneMergeExt: . . . 69
3.3 Use Cases - Algorithm walk through . . . 76
3.3.1 Cocos 3d . . . 76
3.3.2 Facebook - ROCKSDB . . . 78
3.3.3 RethinkDB . . . 80
3.4 Validation . . . 80

II Source code Rewriting 84

4 Source code Rewriting - Aiding program transformation using the program dependence graph 85
4.1 Motivating Example . . . 87
4.2 Language . . . 89
4.2.1 Rules . . . 90
4.2.1.1 Top-level rules . . . 90
4.2.1.2 Scoped rules . . . 91
4.2.1.3 Tags . . . 93
4.2.1.4 Scoped rules with parameters . . . 95
4.2.2 Assessment . . . 96
4.3 Semantics . . . 96
4.3.1 Program model . . . 97
4.3.2 Algorithm . . . 98
4.3.2.1 Finding a node at which to apply a rule . . . 99
4.3.2.2 Substituting the Right-Hand-Side of a Rule . . . 100
4.3.2.3 Applying a Scoped Rule . . . 101
4.3.2.4 Dependencies . . . 103
Automatic Renaming . . . 103
4.3.2.5 User Interactivity . . . 103
4.3.2.6 Formal properties . . . 103
Termination. . . . 104
Confluence. . . . 104
Determinism. . . . 104
Complexity. . . . 105
4.3.3 Limitations . . . 105
4.4 Case Studies . . . 105
4.4.1 Vectorization . . . 107
4.4.1.1 Guidance Rule . . . 109
4.4.2 GMP Specification . . . 110
4.4.3 Array of Struct vs Struct of Arrays . . . 111
4.5 Implementation and Results . . . 113
4.5.1 Vc . . . 113
4.5.2 GMP . . . 114
4.5.3 Array of Struct vs Struct of Array . . . 114
4.6 Furter Evaluation . . . 115
4.6.1 Scalability . . . 115
4.6.2 Generality . . . 116
Partial transformation of code. . . . 117

III Related work and conclusions 121

5 Related work 123

5.1 Software reuse . . . 123
5.1.1 Clone detection . . . 123
5.1.1.1 Text/String based clone detection techniques . . . 124
5.1.1.2 Token based clone detection techniques . . . 125
5.1.1.3 Tree based clone detection techniques . . . 126
5.1.1.4 Metric based clone detection techniques . . . 126
5.1.1.5 Program analysis based clone detection techniques 127
Hybrid approaches . . . 127
5.1.1.6 Studies on clone detection . . . 128
5.1.2 Clone management . . . 128
5.2 Software rewriting . . . 129
5.2.1 Related features . . . 130
5.2.1.1 Coccinelle . . . 130
5.2.1.2 Stratego . . . 132
5.2.1.3 TXL . . . 133
5.2.1.4 Rascal . . . 135
5.2.1.5 IDE Refactorings . . . 135
5.2.2 Other program transformation related work . . . 136
5.3 Software refactoring . . . 137
5.4 Software Evolution . . . 139

6 Conclusion **141**
6.1 Reusing old code . . . 141
6.2 Rewriting existing code . . . 142
6.3 Overall contributions and future work . . . 142

A Coding Tasks and Programmer Poll **144**

Bibliography **146**

List of Figures

1.1 Distribution of maintenance effort . . . 3

2.1 An example of merging two functions by introducing a boolean parameter and an **if** statement. . . . 20
2.2 Example of a three-way merge supported by our tool. . . . 21
2.3 Three-way merge in AST form . . . 21
2.4 Outline of the recalculation algorithm to accommodate insertions . 24
2.5 Example map (MAP_{cn}) generated by the common difference identification phase . . . 26

3.1 Example clone group merge . . . 67
3.2 Outline of the algorithm to merge clone instance into an existing abstraction . . . 75
3.3 Cocos 3d - Abstraction . . . 77
3.4 Facebook RockSDB - Abstraction . . . 79
3.5 RethinkDB - Abstraction . . . 81
3.6 Validation of merging clone instance with merged methods . . . 83

4.1 Vectorization using the Vc library . . . 88
4.2 Struct definition before and after applying scope . . . 93
4.3 Example usage of tags . . . 95
4.4 Dependency links . . . 98
4.5 Outline of the transformation algorithm . . . 100
4.6 Case Study Vc Specification . . . 106
4.7 Integer to big integer conversion using GMP . . . 110
4.8 Case Study GMP Specification . . . 110
4.9 Going from Array of Struct to Struct of Array . . . 112
4.10 AoS to SoA Specification . . . 112
4.11 Vc runtimes, as box plots summarizing 99 runs. . . . 114
4.12 Results of usability study of GMP specifications. Lines/Sites list lines before/lines after/transformed sites. . . . 118
4.13 Function to find the largest number in an array of integers . . . 119
4.14 Function to find the largest number in an array of big integers . . . 119

5.1 Grammar for numerical expression evaluator . . . 133
5.2 Transformation rules for numerical expression evaluator . . . 134

A.1 Amount of time used for extending functionality. x-axis = time taken, y-axis = user, red triangle = copy-paste, blue triangle = abstraction 145
A.2 Preferred results after extending functionality. Out of the 20 answers we received, 3 were undecided, 5 preferred copy-pasted code, and 12 preferred abstracted code. 145

List of Tables

2.1 Available resolution patterns as options presented to the user 30
2.2 Repositories with their pull request URLs. Each clone group represents one abstraction. We encourage readers who choose to look at the pull requests to go through the comments. While some of the pull requests do not explicitly have their status listed as 'merged', as with the OracleDB and the MongoDB repositories, the code has actually been merged, as indicated by the maintainer comments. . . 45
2.3 Phase 1 results summary 47
2.4 Phase 2 results summary 47

A.1 Student experience levels (self reported). 144

To my parents

Chapter 1

Introduction

Software evolution deals with the process by which programs are modified and adapted to their changing environment [1]. Software evolution can also be seen as a result of software maintenance. As a software is maintained, it undergoes evolution. To understand the connection between software evolution and maintenance, let us look at the types of software maintenance as defined by the ISO/IEC 14764:2006 [2]:

- **Corrective maintenance**: Reactive modification of a software product performed after delivery to correct discovered problems.
- **Adaptive maintenance**: Modification of a software product performed after delivery to keep the software product usable in a changed or changing environment.
- **Perfective maintenance**: Modification of a software product after delivery to improve performance or maintainability.
- **Preventive maintenance**: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

Adaptive, Perfective and Preventive maintenance can be seen as modifications that arise due to evolving needs of the software and therefore be categorized under 'Software Evolution.

The need for software evolution. Software evolution is inevitable. In his acclaimed book **Software Engineering, 7th Edition** [3], Ian Sommerville states 'The majority of the software budget in large companies is devoted to evolving existing software rather than developing new software'. Fred Brooks, in his book **The mythical man-month** [4] claims that any successful piece of software will inevitably undergo maintenance. He estimates that more than 90% of the costs of a typical system arises during the maintenance phase. The need for software evolution is further emphasized by the first of Lehman's 8 laws of software evolution, which states 'software systems must be continually adapted or it becomes progressively less satisfactory' [5].

1.1 Software evolution and source code

Software evolution and maintenance are realized by developers by making changes to the program source code. In order to understand the the kinds of changes a developer may need to perform to support software evolution and maintenance, consider the distribution of maintenance effort in Figure 1.1 (as classified by Ian Sommerville [3]). The majority of the maintenance effort is spent on 'addition or modification of features'(65%), followed by effort spent on 'adapting software'(18%) and 'repairing faults'(17%).

- *Fault repair* typically occurs due to bugs in the code. For example, a bug is introduced when a developer forgets to add a check for minimum age requirements while implementing a program to gather the list of citizens who can vote in an election. This bug results in faulty executions which need to be repaired.

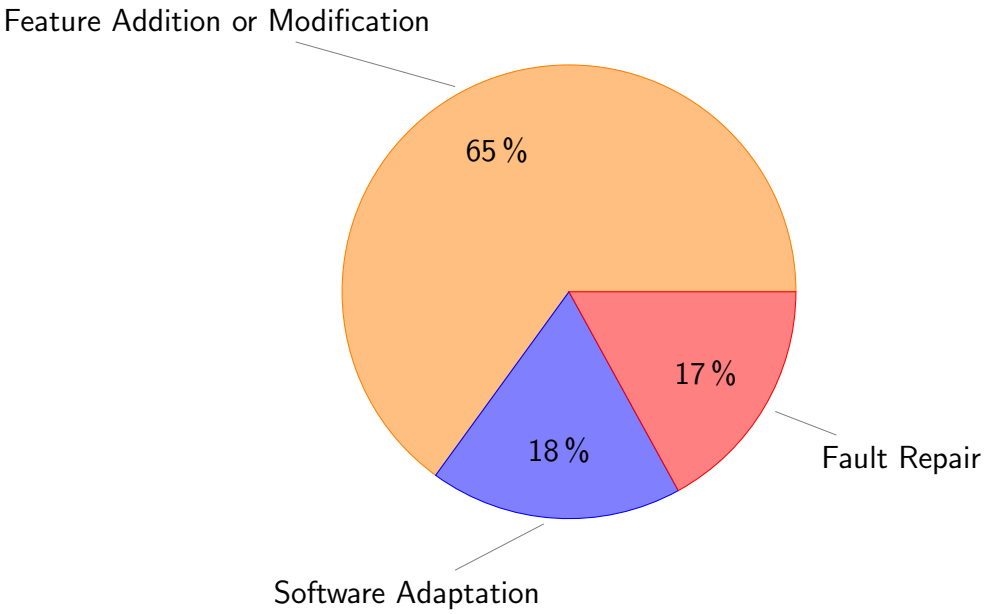


FIGURE 1.1: Distribution of maintenance effort

- *Feature addition* occurs for a various number of reasons. It typically occurs when a customer arrives with a new requirement. For example, a customer from the banking sector may want the software to support a new way of calculating interest.

- *Feature modification* can happen due to external customer requirement changes, or internally to perform optimization or tweak to the software. For example, the customer of a graphical software may need the interface to be updated with a new design.

- *Software adaptation* occurs in scenarios when the software needs to be adapted to a changing environment. Typical forms of adapting software involve migrating from one library to another, migrating from one data representation to another, performing large scale optimization etc. For example, a software having gotten access to sophisticated hardware may need to adapt its code to make use of the new hardware.

All the above mentioned tasks require some sort of code analysis, transformation and generation.

1.2 Contribution

In this work, we look at a few unexplored challenges involved with developing analyses and transformations that support source code evolution. We establish their significance with user studies and related work. We come up with program analysis and transformation-based approaches to address these challenges. We create prototype tools based on these approaches. We evaluate the effectiveness, usability and feasibility of our approach. We report on the results and insights.

Our primary focus is on the feature addition/modification and software adaptation parts of maintenance, as they are central to software evolution. Fault repair falls under the category of 'corrective maintenance' which we recall was not categorized as evolutionary maintenance. The core hypothesis of our thesis is that software evolution tasks are complicated enough that they need to be automated. But the tools built to automate them should essentially aid developers, and therefore the developers must have the ultimate say in how the evolution happens. We propose to arrive at program analysis and transformation frameworks that aid software evolution by developing automation that is guided by user interaction. In particular, we are concerned with two tasks that arise during software evolution that are yet to be deeply explored.

Cleaning up copy-pasted clones. Feature addition happens by introducing new functionality. New functionality is sometimes very similar to existing functionality and developers re-use existing functionality. We look at the two prominent methods of accomplishing such code re-use, i.e function abstraction and copy-paste-modify. We conduct user studies to compare these methods of re-use to establish our hypothesis that copy-paste-modify is easier, but abstraction is preferred. We propose that tool support is required to clean up code clones that result from copy-paste-modify using well chosen abstractions.

Data representation migration. Software adaptation typically involves code transformations. A myriad of tools and frameworks exist to help developers with

automated code transformations. The two primary category of tools that aid code transformations are **Refactoring tools** and **Transformation Languages**.

Refactoring tools provide simple transformations like **rename variable** or **extract method**. Refactoring tools have the advantage of tracking dependencies of transformation in one code location and propagating the change into all of its dependencies, thereby allowing users to selectively apply the transformations. But refactoring tools come with a specific set of built-in transformations. An alternative to such transformations are **Transformation Languages**, which support customizable rules, where the developer can create any number of transformation rules and apply them over her code to support feature adaptation. Current transformation languages do not support selective application of rules.

We hypothesize that a specific kind of software adaptation, i.e migrating the software from one data representation into another, requires both the customizability of **Transformation Languages** and the interactive, selective transformation of **Refactoring**. We propose that tool support is required to support both these features.

In the following subsections, we introduce discuss the significance and our contribution to the above mentioned challenges in detail.

1.2.1 Source code reuse - Cleaning up copy paste clones with interactive merging

As software developers add new features to their programs, they often find that some new feature is very similar to an existing one. The developer then faces a choice: they can either introduce a (possibly complex) abstraction into the existing, working code, or copy and paste the existing code and modify the result. Introducing the *abstraction* produces smaller, more maintainable code but alters existing functionality on the operational level, carrying the risk of introducing inadvertent semantic changes. Copying, pasting, and modifying introduces duplication in the form of *near clones* (type-3 clones in the terminology of Koschke et

al. ([6]), i.e., clones with nontrivial differences), which tends to decrease maintainability, but avoids the risk of damaging existing functionality [7].

Duplication is widespread [8, 9], especially if we count near clones. However, duplication is unpopular in the practitioner literature [10] and "*can be a substantial problem during development and maintenance*" [11] as "*inconsistent clones constitute a source of faults*". Similarly, the C++ developers taking part in our user study (Section 2.2.1), preferred to read code using abstraction rather than duplication. This suggests that there is a discrepancy, which we refer to as the *reuse discrepancy*, between what developers want and what they do.

Kapser and Godfrey ([7]) offer one possible explanation: they claim that "*code cloning can be used as an effective and beneficial design practice*" in a number of situations, but observe that existing code bases include many clones that do not fit their criteria for a 'good clone'. We suggest an alternative explanation, namely that developers view cloning as an implementation technique rather than a design practice: they would prefer abstraction, but find copy-paste-modify faster or easier to use. In an informal poll, we found evidence that supports this idea.

We propose a novel solution to the reuse discrepancy that offers all the speed and simplicity of copy-paste-modify together with the design benefits offered by abstraction, by using a refactoring algorithm to merge similar code semi-automatically. In chapter 2, we will discuss the details of the significance of the discrepancy, the motivation behind our approach, the design and implementation of a prototype tool and the supporting experiments and evaluation.

1.2.2 Software Reuse - Merging clone instances into existing abstractions

Our initial approach merges clones into abstractions. But, it may be desirable for developers to introduce clones without the awareness of existing abstractions and have tool support that will detect and merge the newly introduced clones into existing abstractions if they are available. The approach used to merge clones

into abstractions does not suffice to merge clones into existing abstractions. In chapter 3, we discuss an extension to our clone merging tool that also supports detecting possible abstractions for newly introduced functionality and merging the functionality into the existing abstraction in .

1.2.3 Source code rewriting - Data Representation Migration

A frequently useful type of software adaptation is a migration between data representations. For example, converting integers to bigints enables computations on larger values, converting an array of structures to a structure of arrays can improve locality, and vectorization at the source-code level can better exploit the capabilities of a machine that provides vectorized instructions. Such evolution may require disparate but interconnected transformations all over a program.

Consider the problem of vectorization. Vectorization enables software to simultaneously perform various arithmetic operations on adjacent array elements, rather than individually on scalar values. The Vc API [12] provides a collection of functions and datatypes that ensure explicit vectorization of C++ code at the source level. Unlike compiler-level vectorization, which may or may not succeed, source-level vectorization guarantees vectorized execution [13]. The Vc API provides custom types, such as float_v representing a float vector, having the capability to hold more than one float, where the exact number depends on the underlying machine. Vectorization with the Vc API thus requires users to change type declarations, operations that use values of these types, custom data structures that hold the results of these operations, functions that receive as arguments or return as results the new types of values, and so on. Performing multiple such scattered changes quickly becomes tedious and error-prone. We therefore argue for the need for tool support for such changes. To address the needs of various kinds of data types, such a tool should be easily extensible.

Existing tools for automated program transformation can support such migration to some degree: users can write a specification for e.g. Stratego [14] or Coccinelle [15] and ask these tools to apply the specification to the entire program. Unfortunately, this process does not consider the issue that a user may want a transformation to be applied selectively, to only specific instances of a data type, for semantic or performance reasons. For example, consider migrating from `int` to a `bigint` type in order to scale an arithmetic module to support larger numbers. In a C, C++, or Java program, this is not a change that we would want to automatically apply to all `int` variables in the entire program. Instead, we will typically want to be able to select a subset of the integer variables in the program, and their uses, to transform.

We propose a tool that supports semi-automated migrations of data representations. Our tool provides a rule-specification language, coupled with a transformation engine that applies the rules to C or C++ code. In chapter 4, we will discuss the motivations behind the features we provide in our language, the design of the features, a prototype implementation and use case studies.

Part I

Source code Reuse

Chapter 2

Source code Reuse - Cleaning up copy-pasted clones with interactive merging

Chapter Overview

In this chapter, we look at popular open-source repositories, gather clone groups that are representative of copy-pasted code and study them to motivate the potential mechanisms for abstracting copy-pasted code. With our approach, developers reuse code by copying, pasting, and modifying it, producing near clones. Developers then invoke our tool to merge two or more near clones back into a single abstraction. Since there may be multiple ways to introduce an abstraction, our tool may ask the developer to choose which abstractions he prefers during the merge. Our tool is easily extensible, so that developers may add support for their own abstractions (e.g., project-specific design patterns). Conceptually, the approach we propose and evaluate can be applied at any code granularity, but we have chosen method-level clones as the focus because methods represent well-recognized units of reuse. The abstractions produced by our approach are also represented as methods.

We find that our approach is not only effective at solving the aforementioned reuse discrepancy, but also produces code that meets the quality standards of existing open-source projects. Moreover, our approach can improve over manual abstraction in terms of correctness: as with other automatic refactoring approaches, ensuring correctness only requires validating the (small number of) constituents of the automatic transformation mechanism [16, 17], as opposed to the (unbounded number of) hand-written *instances* of manual abstractions that are required without a tool. Our contributions are as follows:

- We describe the results of our manual study of various clone groups in popular open-source repositories, and identify abstraction mechanisms that may be relevant to them.
- We present a small user study that we have carried out, in which C++ programmers were asked to use copy-paste-modify or abstraction in a set of coding tasks. We also present the results of a poll among these C++ programmers on their opinion of the desirability of the various forms of produced code. While our study and poll have only a small sample size, they suggest that developers prefer to use copy-paste-modify for development, but find the results of abstraction to be preferable.
- We describe an algorithm that can automatically or semi-automatically merge near-clones and introduce user-selected abstractions. The abstraction mechanisms supported in our implementation of this algorithm are motivated by our manual study of clone groups.
- We report on initial experiences with our algorithm on popular C++ projects drawn from open-source repositories. We find that the generated merged code is of sufficiently high quality to be accepted as a replacement to unmerged code in most cases.

The rest of this chapter is organized as follows. Section 2.1 present our user study and poll that further guide the design of our approach. Section 2.2 describes our experiences with a manual study of clone groups from popular open-source

repositories. Section 2.3 describes our merge algorithm, and Section 2.4 gives an overview of its implementation. Section 2.5 provides an informal proof of why our algorithm, along with the choice of resolution mechanisms, preserves the semantics of the code. Section 2.6 presents our evaluation on various open-source software projects, and Section 2.7 presents a larger example in detail.

2.1 Resolution Patterns for Merging Method-Level Code Clones

To understand the commonly occurring differences in such clones in C++ code and to motivate a potential approach to merging method-level clones, we conducted a study of clone groups from top trending open source GitHub repositories. In this section, we will describe the set-up of the study and report on our findings and insights.

As we were not able to find a method-level clone detector for C++, we chose to implement a simple clone detector of our own based on the robust tree edit distance algorithm RTED [18]. The tree edit distance represents a metric for the amount of edits required to transform one tree into another, which intuitively represents a basis for a metric of similarity for copy-paste-modified code. We adapted the existing implementation,[1] which works on trees of strings, to support the AST nodes of the Eclipse CDT.[2] RTED computes the nodes that we need to add to or remove from one AST to obtain another; its output is an *edit list*, i.e., a list of *delete*, *insert* or *relabel* operations:

- **delete** a node and connect its children to its parent, maintaining their order.
- **insert** a node between an existing node and a sub-sequence of its neighbouring siblings of this node.
- **relabel** a node, essentially replacing one node by another.

[1] http://tree-edit-distance.dbresearch.uni-salzburg.at/
[2] https://eclipse.org/cdt/

Based on the results of the RTED algorithm, we calculated the edit distance, which is the size of the edit list, between every possible pair of methods in the top 6 trending C++ repositories in Github in the month of December 2014. The repositories were:

1. ForestDB, a key-value store, developed by Couchbase:
   https://github.com/couchbase/forestdb
2. Google Protobuf, a library for data interchange:
   https://github.com/google/protobuf
3. Open CV, a computer-vision library, originally from Intel:
   https://github.com/Itseez/opencv
4. Facebook's HHVM, a virtual machine supporting Hack and PHP:
   https://github.com/facebook/hhvm
5. Facebook's RocksDB, a persistent key-value store for fast storage environments:
   https://github.com/facebook/rocksdb
6. Tiled, a map editor:
   https://github.com/bjorn/tiled

We ran our tree edit distance calculator with a threshold of 0.5. The threshold between two methods in a pair is defined as:

$$\frac{\text{difference between the number of nodes in the two methods}}{\text{the number of nodes in the method with the smaller tree}}$$

This setting resulted in 111 clone pairs. We further filtered our sample set by picking the top 10 clone pairs that were closest (from both above and below) to the thresholds of 0.1, 0.2, 0.3, 0.4 and 0.5. We also included thresholds of 0.1 and 0.2 where the methods are not very similar for exhaustiveness of our study. This left us with 50 clone pairs. We manually analyzed the differences between the individual clone pairs and identified potential ways of merging them. For example,

if the difference was between two literal expressions (constants) we could merge them by abstracting the literal as a global variable or an extra parameter. If the difference was between two types, we could introduce a template type argument. Some kinds of differences could also be resolved, e.g., using a delegate [19], but for this thesis we are focusing on ways of merging that are non-intrusive, in the sense that they do not require introducing new classes.

In the following, we describe the top three types of differences we observed, using real examples from our sample set and the methods of merging we propose for those differences. For readability, we illustrate each case using very small examples that would probably not be worthwhile to merge in practice. We present more realistic examples in our evaluation in Sections 2.6 and 2.7. In the examples that follow, the code differences and the manually generated parameters and code segments are highlighted with a gray background.

Difference type 1 — Constants. Consider the following functions from Google's Protobuf:[3]

```
// Return the name of the AssignDescriptors()
// function for a given file.
string GlobalAssignDescriptorsName(const string& filename) {
  return "protobuf_AssignDesc_" + FilenameIdentifier(filename);
}

// Return the name of the ShutdownFile()
// function for a given file.
string GlobalShutdownFileName(const string& filename) {
  return "protobuf_ShutdownFile_" + FilenameIdentifier(filename);
}
```

These functions differ only in the string, `"protobuf_AssignDesc_"` or `"protobuf_ShutdownFile_"`, used to make up the beginning of the return value. As both strings are constants, we can create a merged version of these functions, in which the differences are resolved using a global variable or an extra parameter. The following code shows the result when using a global variable. The code includes

[3] https://github.com/google/protobuf/blob/6ef984af4b0c63c1c33127a12dcfc8e6359f0c9e/src/google/protobuf/compiler/cpp/cpp_helpers.cc

both the merged function and the original functions that have been modified to use the new merged version.

```
string globalvar = "";
string mergedFunction(const string &filename) {
  return globalVar + FilenameIdentifier(filename);
}

string GlobalAssignDescriptorsName(const string& filename) {
  globalVar = "protobuf_AssignDesc_" ;
  return mergedFunction(filename);
}

string GlobalShutdownFileName(const string& filename) {
  globalVar = "protobuf_ShutDownFile_" ;
  return mergedFunction(filename);
}
```

The following code likewise shows the result when using an extra parameter:

```
sting mergedFunction(const string &filename, string extraParam) {
  return extraParam + FilenameIdentifier(filename);
}

string GlobalAssignDescriptorsName(const string& filename) {
    return mergedMethod(fileName, "protobuf_AssignDesc_");
}

string GlobalShutdownFileName(const string& filename) {
  return mergedMethod(fileName, "protobuf_ShutdownFile_");
}
```

Difference type 2 — Types. The following code shows another pair of functions from Google's Protobuf[4] that differ in a parameter type:

```
string TextFormat::FieldValuePrinter::PrintInt32(int32 val) const {
  return SimpleItoa(val);
}

string TextFormat::FieldValuePrinter::PrintUInt32(uint32 val) const {
  return SimpleItoa(val);
}
```

[4] https://github.com/google/protobuf/blob/6ef984af4b0c63c1c33127a12dcfc8e6359f0c9e/src/google/protobuf/text_format.cc

The code below merges these functions using a template argument:

```
template <typename T>
string TextFormat::FieldValuePrinter::PrintInt(T val) const {
  return SimpleItoa(val);
}

string TextFormat::FieldValuePrinter::PrintInt32(int32 val) const {
  return PrintInt<int32>(val);
}

string TextFormat::FieldValuePrinter::PrintUInt32(uint32 val) const {
  return PrintInt<uint32>(val);
}
```

Difference type 3 — Statements. The following code shows a pair of functions from Facebook's RocksDB[5] that differ in various aspects of a statement containing a function call. It is to be noted that statement level differences may be considered as differences at a sub-fragment of the statement. In our example below, the difference between the two function calls to the same function **Merge** could also be interpreted as just an extra argument. The ability to merge statement level differences is useful when handling scenarios where no other methods of merging the differences in the sub-fragment are available.

```
void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
  b->rep.Merge(Slice(key, klen), Slice(val, vlen));
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
  b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
}
```

We propose two ways to merge statement level differences, using either conditionals or a switch statement. We show only the result using conditionals, below:

```
void abstractedFunction(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
```

[5] https://github.com/facebook/rocksdb/blob/767777c2bd7bf4be1968dbc35452e556e781ad5f/db/c.cc

```
    const char* key, size_t klen, const char* val, size_t vlen,
    int functionID) {
  if(functionId == 1) {
    b->rep.Merge(Slice(key, klen), Slice(val, vlen));
  }
  else if(functionId == 2) {
    b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
  }
}

void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
  abstractedFunction(b, null, key, klen, val, vlen, 1);
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
  abstractedFunction(b, column_family, key, klen, val, vlen, 2);
}
```

In summary, we gather the most common type of code clone differences in practice, i.e, **constants**, **types** and **statements** based on our analysis of a number of near clone methods from popular open source repositories. We also use the insights gathered by manually coming up with potential ways of merging the clones to motivate the algorithm we propose will resolve these differences automatically. We will discuss our how algorithm makes use of these insights in Section 2.3.

2.2 User Study and Informal Poll

Past work on clone detection has found that clones are widespread [8, 6, 9]. We hypothesize that a key cause for this prevalence of clones is that *copy-paste-modify makes software developers more productive*, at least in the short term. To explore this hypothesis, we conducted a preliminary, exploratory experiment with a group of graduate student volunteers.

2.2.1 Benefits of Copy-Paste-Modify

For our user study, we selected five pairs of C++ methods from Google Protobuf,[6] Facebook HHVM,[7] and Facebook RocksDB,[8] randomly choosing from the set of near-clones reported by our own clone detector, described in Section 2.1. We then removed one of the methods and asked five graduate students with 2 months, 3 months, and 1, 4, and 10 years of (self-reported) C++ programming experience, respectively, to implement the missing functionality. We asked the students with 3 months and 4 years of experience to modify the existing method to support both the existing and the new functionality (i.e., to perform *manual abstraction*), and the remaining students to use *copy-paste-modify*. All students worked on all five tasks.

We found that the students using copy-paste-modify were almost universally faster in completing their objectives (2–15 minutes) than the students who performed manual abstraction (7–55 minutes, with three tasks left incomplete). We found only one exception, where the best-performing student using manual abstraction completed the task in the same time as the worst-performing student using copy-paste-modify. Since the three students using copy-paste-modify finished first and had a lot of their allocated time still left, we asked two of the copy-paste groups to abstract two of the tasks, and the users in the third copy-paste group to abstract one of the tasks. Despite their familiarity with the code, they consistently performed worse (taking more than twice as long as before) when completing the same task again with manual abstraction. However, the same developers showed a preference for *having* abstractions as a result (in 12 cases, vs. 5 for copy-paste-modify, out of 20 responses, cf. Appendix A).

While our numbers are too small to be statistically significant, they provide evidence that copy-paste-modify can be more effective than manual abstraction at accomplishing reuse at the method level.

[6] https://github.com/Google/protobuf
[7] https://github.com/Facebook/hhvm
[8] https://github.com/Facebook/rocksdb

2.2.2 Copy-Paste-Modify versus Manual Abstraction

To understand *why* copy-paste-modify might be easier, consider the function `costFunction1` from Figure 2.1. This function (adapted from the OpenAge[9] project) computes the Chebyshev distance of two 2-dimensional coordinates. The implementation consists of a function header with formal parameters, a computation for the intermediate values `dx` and `dy`, and finally a computation of the actual Chebyshev distance from `dx` and `dy`.

At some point, a developer decides that a different distance function is needed, describing the beeline distance between two points (i.e., $\sqrt{\mathsf{dx}^2 + \mathsf{dy}^2}$). Computing this distance requires almost the same steps as implemented in `costFunction1`, except for calling the standard library function `std::hypot` instead of `std::max`. At this point, the developer faces a choice: she can copy and paste the existing code into a new function (requiring only a copy, paste, and rename action) and modify the call from `std::max` to `std::hypot` (a trivial one-word edit), or she can manually transform the function `costFunction1` into a more abstract version, such as `costFunctionM` (depicted on the bottom right in **Figure 2.1**).

This transformation from the copy-pasted code to an abstracted code requires introducing a new parameter, introducing an **if** statement, adding a new line to handle the new case, and updating all call sites with the new parameter (perhaps using a suitable automated refactoring). Intellectually, the developer must reason about altering the function's control flow, formal parameters, and any callers that expect the old functionality, whereas with copy-paste-modify, they only need to concern themselves with the exact differences between what already exists and what they now need.

We observe the need to devise an algorithm that takes the definitions of `costFunction1` and `costFunction2` and abstracts them into a common `costFunctionM`, taking care that any callers still continue to work correctly. Note that there are multiple possible strategies for `costFunctionM`. For example, we could pass `std::hypot`

[9] http://openage.sft.mx/

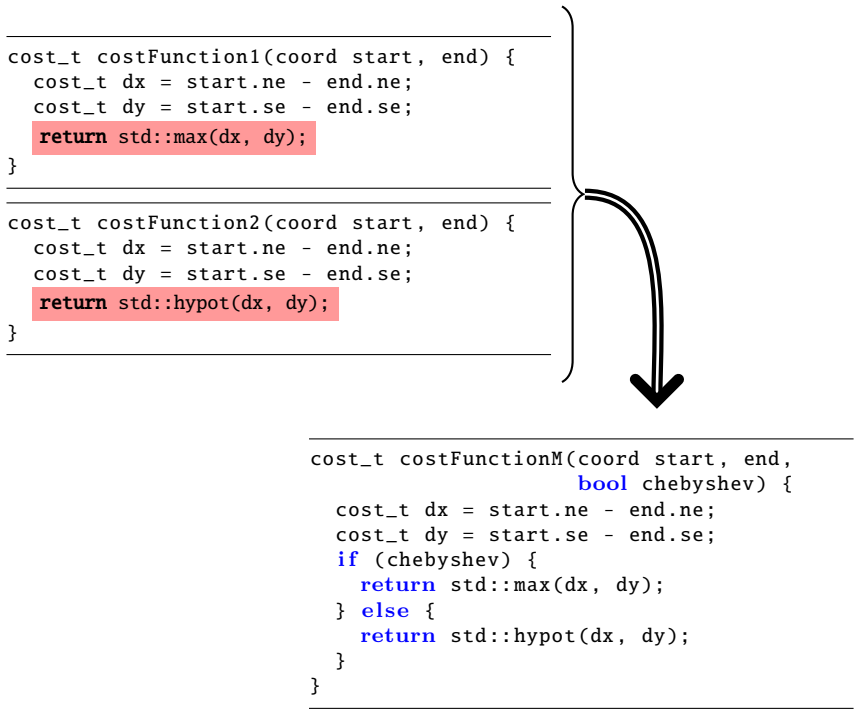


FIGURE 2.1: An example of merging two functions by introducing a boolean parameter and an **if** statement.

or `std::max` as function parameters or pass an enumeration parameter to support additional metrics within this one function. The 'best' abstraction mechanism may depend on style preferences, performance considerations, and plans for future extension; we thus choose to provide the user with the means to choose the most appropriate abstraction mechanism for a given situation.

2.3 Merging Algorithm

To illustrate how our algorithm merges a collection of near-clone functions into an abstracted function, we will use the three functions at the top of **Figure 2.2** as a running example. These synthetic functions are unlikely merge candidates, since they are both small and rather dissimilar, but they illustrate special cases in our algorithm and show that our approach works even for code with a large degree of variation.

```
 void function1() {
   b(c,k(d));
   y = f1;
   x(z);
 }
```

```
void function2() {
  b(c,k(e));
  y = f2;
  x(z);
}
```

```
void function3() {
  b2();
  y = f3;
  n();
  x(z);
}
```

```
void fnMerged(int functionId, int fValue, int bParam) {
  if(functionId == 12) {
    b(c, k(bParam));
  }
  else if(functionId == 3) {
    b2();
  }
  y = fValue;
  if(functionId == 3){
        n();
  }
  x(z);
}
```

FIGURE 2.2: Example of a three-way merge supported by our tool.

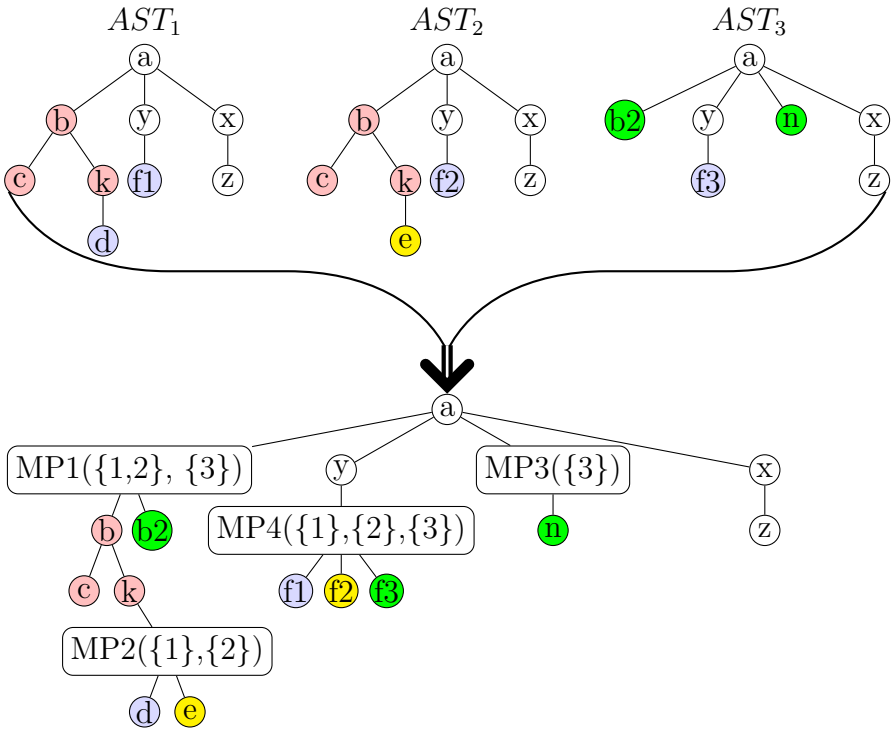


FIGURE 2.3: Three-way merge in AST form

2.3.1 Abstract Syntax Trees

Our algorithm works at the Abstract Syntax Tree (AST) level. **Figure 2.3** shows simplified ASTs, omitting operators for conciseness, for the functions in **Figure 2.2**.

In our ASTs, each node contains a *label* and a *position*. The *label* of a node is the type of the node along with any content the node contains. For example, in an AST representing the declaration **int x = 10;**, the node corresponding to **int** would have type **Type** and content **int**, and the node corresponding to **10** would have type **Literal Expression** and content **10**. The *position* of a node is the traversal path to this node from the root of the AST [20]. The position is a list of numbers, with each number representing the *offset*, starting with 1, from the leftmost child of each node's parent to the node. In **Figure 2.3**, the *position* of the node 'y' in all of the ASTs is (1, 2). The *position* of the node 'e' in AST_2 is (1, 1, 2, 1).

2.3.2 RTED

Our algorithm relies on the RTED algorithm to identify common nodes and inserted nodes. As described in Section 2.1, RTED computes the edit distance between two trees, *i.e.*, the number of edit operations that are required to transform one AST into another.

Each element of an edit list is a pair (n_a, n_b), describing a single edit operation, transforming node n_a in AST_A into node n_b in AST_B. The edit list is computed based on the tree structure and the node content, but independently of the node position. The edit lists produced by RTED are completely symmetric, *i.e.* the result of applying RTED to a pair of ASTs (AST_x, AST_y) is simply the reverse of the result of applying RTED to (AST_y, AST_x). At most one component of an element of an edit list can be 0, indicating that the node in the other component is inserted into its corresponding tree. For example, $(a, 0)$ indicates that node a is inserted into tree AST_A.

The edit lists of each pair of ASTs in our example in Figure 2.3 are:

- Edit List of (AST_1, AST_2) : (d, e), (f1, f2)
- Edit List of (AST_2, AST_3) : (b, b2), (c, 0), (k, 0), (e, 0), (f2, f3), (0, n)

- Edit List of (AST_1, AST_3) : (b, b2), (c, 0), (k, 0), (d, 0), (f1, f3), (0, n)

Based on the results of RTED, our algorithm identifies nodes that do not appear in any edit list as being common to all ASTs. Note that the set of common nodes does not necessarily include all subtrees that look alike. For example, consider the trees A = **(a(b(treex),c))** and B = **(a(b,c(treex)))**. RTED could consider 'a', 'b', and 'c' to be common, or it could consider 'a' and 'treex' to be common, but it cannot consider all of them to be common at once, due to conflicting structural constraints. In the former case, for example, 'treex' would be considered to be inserted as the child of node 'b' in tree A and as the child of node 'c' in tree B.

Recalculating positions of nodes to accommodate insertions. Our merging algorithm (Section 2.3.3) compares nodes based on the notion of equality of nodes, which depends on both the content and the position of the nodes. Our algorithm relies on the positions of the nodes in order to construct the merged AST from the input ASTs. We first normalize the positions across all trees, to account for inserted nodes in order to simplify the merging process. To understand the need for such a normalization step, consider the various occurrences of the node 'x' in Figure 2.3. The position of 'x' in AST_1 and AST_2 is (1, 3) and the position of 'x' in AST_3 is (1, 4). This is because the node 'n' is inserted before the node 'x' in the edit list of (AST_1, AST_3) or (AST_2, AST_3). The normalization step will change the position of 'x' in all ASTs from (1, 3) to (1, 4).

Overview of the algorithm. Since there are a lot of details in the description of the algorithm that follow this paragraph, we begin with an overview of the algorithm. The algorithm simply looks at the edit list of every pair of ASTs A and B, and for every insertion in A with respect to B, the algorithm shifts the position of all nodes in B that occupy the position of the inserted node in A by adding one to the relevant offsets. The algorithm also needs to shift all the nodes that are impacted by the previous shifting. For this purpose the algorithm uses

Procedure RecalculatePosition
Input: $ASTs \leftarrow$ input ASTs, $editList_{x,y} \leftarrow$ RTED(AST_x, AST_y) for each pair of ASTs
Side effect: **Position** for every *node*
in each AST that was impacted by an insertion
1 - MAP$_{mp}$: AST$\rightarrow$ Position Set := {}
2 - foreach pair $(AST_x, AST_y) \in ASTs \times ASTs$ where $x < y$
3 - if ($AST_y \notin$ dom(MAP$_{mp}$)):
4 - MAP$_{mp}[AST_y]$:= {}
5 - foreach $(n_a, 0) \in editList_{x,y}$ /* n_b component is zero */
6 - if($n_a.position \notin$ MAP$_{mp}[AST_y]$)
7 - iLastOffset $\leftarrow n_a.position.size$
8 - foreach $node \in AST_y$ where
. - ($node.position.size == n_a.position.size \wedge$
. - $node.position$[iLastOffset]$\geq n_a.position$[iLastOffset])
9 - $node.Position$[iLastOffset]= $node.position$[iLastOffset] + 1
10 - foreach $node_{sub} \in$ subtree rooted at *node*
11 - $node_{sub}.Position$[iLastOffset]= $node_{sub}.position$[iLastOffset] + 1
12 - MAP$_{mp}[AST_y] \leftarrow$ MAP$_{mp}[AST_y] \cup \{n_a.position\}$

FIGURE 2.4: Outline of the recalculation algorithm to accommodate insertions

the *iLastOffset*. The algorithm uses a Map (MAP$_{mp}$) to ensure that insertions in the same position for a particular tree do not update the offsets twice.

We describe details of the Algorithm 2.4 below.

The procedure **RecalculatePosition** (Figure 2.4) normalizes the trees by recalculating the positions of all nodes to the right of an inserted node. **RecalculatePosition** takes as input the set of ASTs and the edit list of each pair of ASTs, as obtained from RTED.

RecalculatePosition outputs a data structure **Position**, which is the position list of every node in every AST. **RecalculatePosition** recalculates the position of every node that was impacted by an insertion in each AST.

RecalculatePosition needs to make sure the recalculation does not happen more than once for the same insertion when comparing one AST with multiple ASTs. Consider the three ASTs AST_1 = (a(b)(c)(d)), AST_2 = (a(b)(d)), and AST_3 = (a(b)(d)). The node 'c' is inserted when AST_1 is considered with both AST_2 and AST_3. The shifting of nodes in AST_1 must happen only once. If a node is inserted in an AST when considering it with two trees, we do not want to shift everything to the right twice. For this purpose, we rely on the data structure

$\mathbf{MAP}_{mp}$ which maps each AST to the set of positions that have already been considered for recalculation.

We now walk through the procedure **RecalculatePosition** line by line. **RecalculatePosition** considers every pair of ASTs, AST_x and AST_y (Line **2**). **RecalculatePosition** initializes an empty set as a value for any key AST_y that does not have a value in the $\mathbf{MAP}_{mp}$ (Lines **3, 4**). **RecalculatePosition** considers each entry in every edit list of every pair of ASTs, AST_x and AST_y (Line **5**). Every entry that has the right (n_b) component as zero implies that the node on the left hand side n_a is inserted into AST_x when migrating from AST_y (Line **5**). **RecalculatePosition** uses the map to test whether there are already insertions at the position of n_a for AST_y. **RecalculatePosition** continues with the actual repositioning (Lines **7**–**9**) only if the position of n_a does not exist in the value set for the key AST_y in the map $\mathbf{MAP}_{mp}$ (Line **6**).

RecalculatePosition updates the **Position** of every node in AST_y that occupies the position of n_a (or to the right of it) by one position to the right (Lines **7**–**9**). For every *node* updated in the last step, this section of the procedure also updates the corresponding offsets (denoted by **iLastOffset**) of all the nodes that are part of the subtree rooted at *node* (Lines **10**, **11**). For example, in AST_1 and AST_2, the position of the node 'x' and the nodes that are part of subtree rooted at 'x', i.e 'z' are (1, 3) and (1, 3, 1) respectively before recalculation. After recalculation, the position of node 'x' and 'z' in AST_1 and AST_2 would be (1, 4) and (1, 4, 1), to be consistent with the positions of these nodes in the other AST, AST_3. Finally, **RecalculatePosition** updates the map $\mathbf{MAP}_{mp}$ by adding the position of n_a to the set of positions for the entry corresponding to AST_y (Line **9**). The recalculated **Position**s will be used by the rest of the merging algorithm that follows.

2.3.3 Algorithm

Our merge algorithm is split into three high-level steps:

1. Identifying the *conflict nodes*

$ASTs = \{\mathsf{AST}_1, \mathsf{AST}_2, \mathsf{AST}_3\}$

position	content	Source ASTs
(1, 1, 2, 1)	d	AST_1
(1, 1, 2, 1)	e	AST_2
(1, 2, 1)	f1	AST_1
(1, 2, 1)	f2	AST_2
(1, 2, 1)	f3	AST_3
(1, 3)	n	AST_3
(1, 1)	b	$\mathsf{AST}_1, \mathsf{AST}_2$
(1, 1)	b2	AST_3
(1, 1, 1)	c	$\mathsf{AST}_1, \mathsf{AST}_2$
(1, 1, 2)	k	$\mathsf{AST}_1, \mathsf{AST}_2$

FIGURE 2.5: Example map (MAP_{cn}) generated by the common difference identification phase

2. Constructing the merge tree

3. Applying the resolution patterns

2.3.3.1 Identifying the conflict nodes

After normalizing the positions of the common nodes shared between all ASTs, our algorithm needs to identify the nodes which differ from each other and need to be merged. For this purpose, we introduce the notion of a *conflict node* which is a node where a merge happens. Our algorithm identifies the conflict nodes along with the node's position and the corresponding ASTs in which the node is found.

Our algorithm begins by collecting the common nodes, as defined previously in Section 2.3.2. In our example, the common nodes are 'a', 'x', 'y', 'z'. Our algorithm then maps the remaining nodes to their respective source ASTs (MAP_{cn}), as shown in Figure 2.5. A node, as we recall, comprises its content and its position. For example, the node with the content 'd' and position (1, 1, 2, 1) has the source AST AST_1. The node with content 'b' and position (1, 1) has the source ASTs, AST_1 and AST_2.

We use the term *merge point* to refer to a position where conflict nodes are placed. We identify the merge points based on the positions in the Map_{cn}. For example, at the position (1, 1), there will be a merge point between the node **b** from (AST_1, AST_2) and node **b2** from (AST_3), and there will be a merge point at position (1, 2, 1) between **f1** from AST_1, **f2** from AST_2 and **f3** from AST_3. Every merge point has a set of at one or more choices, with each choice consisting of a conflict node along with the source ASTs that contain the node. The merge point at (1, 1) has two choices. The first choice branches to the node **b** from $\{AST_1, AST_2\}$ and the second choice branches to node **b2** from $\{AST_3\}$. Merge points with one choice, eg (1, 3), can occur only in the case of a single node insertion.

2.3.3.2 Constructing the merge tree

Now that our algorithm has identified the positions where conflict nodes can occur, it constructs the *merge tree*, which represents the abstraction of the near-clone trees that were provided as inputs. Each identified merge point from the previous step also contains information about the individual nodes that are part of the conflict nodes for this merge point, along with their source ASTs. Our algorithm initially creates an empty merge tree. It then places nodes into this tree level by level and for each level, it places the nodes from left to right. The algorithm begins by placing the common nodes in their respective positions. The positions under use currently are the positions that were recalculated as part of the normalization step proposed in Section 2.3.2.

For every position of every node in the domain of the map MAP_{cn}, our algorithm places an identified merge point, with one choice each for every node in that position along with the node's corresponding source ASTs. Merge points are denoted as **MP*x*(List of sets of ASTs)**, where each set of ASTs inside the merge point represents a choice. For the position (1, 1) discussed previously, the merge point would be $\mathbf{MP_1(\{1,2\}, \{3\})}$ in Figure 2.3.

Our algorithm does not create merge points for positions where the nodes under consideration are part of a subtree rooted at a previously formed merge point and

are part of the same source trees. For example, the node 'c' at position (1, 1, 1) will not be part of a merge point, as 'b' at position (1, 1) is an ancestor to 'c' , 'b' is already part of a merge point, and both 'c' and 'b' arise from the same source trees as 'b', i.e $\{AST_1, AST_2\}$.

2.3.3.3 Applying resolution patterns

Finally, we eliminate each merge point by applying a *resolution pattern*. A resolution pattern is a code transformation pattern to resolve the merging of specific types of nodes at a given merge point. Recall from the introduction, that our approach works on near-clone C++ methods. So, the resolution pattern in our approach constructs a concrete node that corresponds to a C++ code fragment that is to be inserted at the merge point. We use the term *merge-substitution* to describe the algorithm that generates the merged node that is to be inserted into the merged methods. This node generated by the merge-substitution algorithm replaces the merge point in the AST.

Additionally, the algorithm handles call sites of the merged near clone methods by generating calls to the merged method from each of the near-clone methods. These generated calls replace the bodies of the existing near-clone methods. Alternatively, our approach can also handle call sites by replacing all the calls to the near-clone methods. For the purpose of generating corresponding calls, we complement our algorithm with a *fix-up* mechanism. The *fix-up* mechanism generates calls to the merged method from individual near-clone methods and introduces appropriate parameters. We begin by discussing first the *merge-substitution* and then discuss the *fix-up* mechanism.

Merge-Substitution. The *merge-substitution* comprises a *selector* and *selection mechanism*. The selector is the device that the caller of an abstraction uses to choose between the various alternatives. The selection mechanism translates the selector into one of the alternatives within the abstraction, in the body of the

called code. To illustrate the concept of selector and selection mechanism, consider the following example code:

```
void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
  b->rep.Merge(Slice(key, klen), Slice(val, vlen));
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
  b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
}
```

and the following merged version of the code:

```
void abstractedFunction(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen,
    int functionID) {
  if(functionId == 1) {
    b->rep.Merge(Slice(key, klen), Slice(val, vlen));
  }
  else if(functionId == 2) {
    b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
  }
}

void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
  abstractedFunction(b, null, key, klen, val, vlen, 1);
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
  abstractedFunction(b, column_family, key, klen, val, vlen, 2);
}
```

In this merged function **abstractedFunction**, the selector is the added parameter **int functionID** to the function **mergedFunction** and the selection mechanism is the testing of **functionId** to choose which statement to execute. The various alternative values for the *selector* can be supplied using the actual selector, which the caller supplies. In this case, the actual selectors are **1** from the function

TABLE 2.1: Available resolution patterns as options presented to the user

Type of Node	Selector	Selection Mechanism
Statement	switch conditional	actual parameter global variable
Literal	formal parameter	actual parameter global variable
Type	template parameter	type value
Identifier	formal parameter function pointer	variable reference actual parameter

rocksdb_writebatch_merge and **2** from the function **rocksdb_writebatch_merge_cf**.

Table 2.1 lists the resolution patterns that our prototype supports, in terms of the node types to which they are applicable and the corresponding selector and selection mechanisms. For example, if the nodes under consideration in a particular position are all literals, we can introduce a formal parameter (selector) of the type of the literal and pass the literal as an actual parameter (selection mechanism) value. Another possibility, although arguably less elegant, would be to introduce a global variable that could be assigned the literal value.

Fix Up. After we merge method definitions, we replace the existing bodies of the definitions with calls to the merged method, as in **Figure 2.2**. Since the merging process involves creating a new merged method and introducing calls to the merged method from the near-clone methods, we need a *fix-up* mechanism to generate a merged version of the parameters for each near-clone method. A merged version of the parameters is simply a set union of set of the parameters of the each individual individual near-clone method. In order to define set-union we need a notion of equality. Two parameters are termed as equal if their types, names and type-qualifier specifiers are equal. We distinguish parameters with different names because alpha-renaming is not part of the merging process. If only the parameters' names are equal and their types are different, the parameters are still considered different. We handle the differing types using our existing pattern which resolves type differences using a template parameter. We maintain a map of the individual

parameters to their positions in the merged parameters to generate appropriate calls.

Below, we discuss the resolution patterns that we have implemented to evaluate our approach. For each resolution pattern, we describe the merge-substitution and the fix-up mechanism. We illustrate our resolution patterns with examples taken from open source projects hosted at GitHub. We picked these four patterns based on the dominant kinds of differences found using our earlier study in Section 2.1. We found these patterns to be sufficient for the examples that we had randomly selected for evaluation. The examples presented here are abbreviated for space reasons. In the examples, the nodes highlighted in red (light gray in a black and white view) indicate the unique nodes in each function and the nodes highlighted in blue (dark grey in a black and white view) indicate the nodes produced by our merge resolution.

2.3.3.4 Pattern: Switch Statement with Extra Parameter

This resolution pattern can be applied if the nodes to be merged are all statements. We then construct the following **switch** statement:
Merge-Resolution:

switch (*choice*) {
  case 1: $stmt_1$; **break;**
  ...
  case k: $stmt_k$; **break;**
}

where *choice* is a fresh method parameter, $stmt_i$ is one statement alternative taken from the individual cases of the switch statement, and i is a unique number identifying the ASTs in the filtered map. We add *choice* as an additional formal parameter(selector) to the surrounding method or function.

Fix-up: We modify the corresponding call sites to supply their own unique AST identifiers as actual parameters. Consider the function snippets

```
jobject function_openROnly__JLjava(JNIEnv* env, jobject jdb,...) {
  rocksdb::DB* db = nullptr;
  rocksdb::Status s;
  /* About 50 lines of common code */
  s = rocksdb::DB::OpenForReadOnly(*opt, db_path, column_families, &handles, &db);
  return null;
}

jobject function_open__JLjava(JNIEnv* env, jobject jdb,...) {
  rocksdb::DB* db = nullptr;
  rocksdb::Status s;
  /* About 50 lines of common code */
  s = rocksdb::DB::Open(*opt, db_path, column_families, &handles, &db);
  return null;
}
```

Our pattern merges these snippets by introducing a switch statement to choose between the two options. Modulo variable renaming and indentation, resolution produces the following output (with the generated switch statement in lines 14–20):

```
1  jobject function_openROnly__JLjava(JNIEnv* env, jobject jdb,...) {
2    return function_open_Merged__JLjava(env, jdb,..., 1);
3  }
4
5  jobject function_open__JLjava(JNIEnv* env, jobject jdb,...) {
6    return function_open_Merged__JLjava(env, jdb,.., 2);
7  }
8
9  jobject function_open_Merged__JLjava(JNIEnv* env, jobject jdb, ...,
10                                      int openType) {
11   rocksdb::DB* db = nullptr;
12   rocksdb::Status s;
13   /* About 50 lines of common code */
14   switch(openType) {
15   case 1:
16     s = rocksdb::DB::OpenForReadOnly(*opt, db_path, column_families, &handles,&db);
17     break;
18   case 2:
19     s = rocksdb::DB::Open(*opt, db_path, column_families, &handles,&db);
20     break; }
21   return null;
22 }
```

2.3.3.5 Pattern: Extra Parameter for Literal Expressions

This resolution pattern can be applied if the nodes to be merged are all literal expressions. Literal expressions are nodes that have a constant value. We require that all of these constants have the same type. The merge-resolution is an identifier that serves as a placeholder for the corresponding constants based on the values passed to a fresh parameter, *value*. An identifier expression is an expression that corresponds to names in C++. We resolve this pattern with an identifier expression, which is a node that contains a reference to the newly generated parameter. We add *value* as an additional formal parameter to the surrounding method or function. The selection mechanism would be any constant that has the same type as **value**. If **value** is an 'int', then an example of a valid selection mechanism would be **10**.

Merge-Resolution: *value*

Fix-up: We modify existing call sites to supply their own constants as the actual parameter input. Consider the following function, taken from the Oracle's Node-OracleDB project:[10]

```
Handle<Value> Connection::GetClientId (Local<String> property,
                                       const AccessorInfo& info) {
  ...
  if(!njsConn->isValid_)
    ...
  else
    msg = NJSMessages::getErrorMsg(errWriteOnly, "clientId");
  NJS_SET_EXCEPTION(msg.c_str(), (int) msg.length());
  return Undefined();
}

Handle<Value> Connection::GetModule (Local<String> property,
                                     const AccessorInfo& info) {
  ...
  if(!njsConn->isValid_)
    ...
  else
    msg = NJSMessages::getErrorMsg(errWriteOnly, "module");
  NJS_SET_EXCEPTION(msg.c_str(), (int) msg.length());
  return Undefined();
}
```

[10] https://github.com/oracle/node-oracledb/

```
Handle<Value> Connection::GetAction(Local<String> property,
                                    const AccessorInfo& info) {
  ...
  if(!njsConn->isValid_)
    ...
  else
    msg = NJSMessages::getErrorMsg(errWriteOnly, "action");
  NJS_SET_EXCEPTION(msg.c_str(), (int) msg.length());
  return Undefined();
}
```

Our tool would identify that the calls to GetClientId, GetModule and GetAction are mergeable using an extra parameter. Modulo variable renaming and indentation, this produces the following output:

```
1  Handle<Value> Connection::GetProperty(Local<String> property,
2                                        const AccessorInfo& info,
3                                        string errorMsg)
4  {
5    ...
6    if(!njsConn->isValid_)
7      ...
8    else
9      msg = NJSMessages::getErrorMsg(errWriteOnly, errorMsg);
10   NJS_SET_EXCEPTION(msg.c_str(),  (int) msg.length());
11   return Undefined();
12 }
13
14 Handle<Value> Connection::GetClientId(Local<String> property,
15                                       const AccessorInfo& info)
16 {
17   return Connection::GetProperty(property, info, "clientId");
18 }
19
20 /* The methods getModule and getAction are constructed to be analogous to
21    GetClientId */
```

2.3.3.6 Pattern: Templates for Type Expressions

We can apply this resolution pattern if the nodes to be merged all represent types. The tool introduces a fresh variable for a template, *type*. We also convert the method into a template method if it is not already one.

Merge-Resolution: *type*. We also introduce a new formal template type parameter (selector) *type* to the function definition. Any type (int, char..,) is a valid selection mechanism.

Consider the following functions taken from the RethinkDB project[11]

```
cJSON *cJSON_CreateIntArray(int *numbers,int count) {
  ...
  for (int i=0;a && i<count;i++) {
    ...
  }
  a->tail = p;
  return a;
}

cJSON *cJSON_CreateDoubleArray(double *numbers,int count) {
  ...
  for (int i=0;a && i<count;i++) {
    ...
  }
  a->tail = p;
  return a;
}
```

Our tool would identify that it can merge the calls to CreateIntArray and CreateDoubleArray by introducing a template type parameter. Modulo variable renaming and indentation, our tool produces the following output:

```
template<typename T> cJSON *cJSON_CreateNumArray(T *numbers,int count) {
  ...
  for (int i=0;a && i<count;i++) {
    ...
  }
  a->tail = p;
  return a;
}

cJSON *cJSON_CreateIntArray(int *numbers,int count) {
  return cJSON_CreateNumArray<int>(numbers, count);
}

cJSON *cJSON_CreateDoubleArray(double *numbers,int count) {
  return cJSON_CreateNumArray<double>(numbers, count);
```

[11] https://github.com/rethinkdb/rethinkdb/

```
}
```

2.3.3.7 Pattern: Extra Parameter for Identifiers

This resolution pattern can be applied if the nodes to be merged are all variable identifiers (identifier expression nodes). We require that all of these variables be of the same type. The *merge-resolution* is a simple identifier expression that switches between the corresponding variable names based on the values passed to the '*value*', a fresh parameter.

Merge-Resolution: *value*

The resolution here is very similar to the pattern for literals, except that our algorithm promotes L-values to pointer-typed parameters whenever the differences involve identifiers which are left hand side of assignment expressions. We opted for pointers instead of references because then the code would also work for C. Analogous to the resolution pattern for literals, the selector is the formal parameter, **value** and selection mechanism is any identifier that has the same type as **value**. Consider the following example:

```
int x, z;
void fn1() {
  x = 10;
  int y = x + 1 + 25;
}
void fn2() {
  z = 10;
  int y = z + 1 + 45;
}
```

Our algorithm handles this case by identifying, among other merge points, two different merge points each for the identifiers x and z. Our algorithm creates a pointer parameter to switch between x and z, and passes references to each of these identifier(&x and &y). We add *value* as an additional formal parameter of the type of the identifiers being merged. A merged version of the functions described previously would look something like this:

```
int x, z;
```

```
void fnMerged(int *ptr, int constant){
        *ptr = 10;
        int y = *ptr + 1 + constant;
}
void fn1() {
        fnMerged(&x, 25);
}
void fn2() {
        fnMerged(&z, 45);
}
```

Fix-up: We modify the corresponding call sites to supply their own identifiers as actual parameters. Consider the function snippets taken from Facebook's HHVM project:[12]

```
Type typeDiv(Type t1, Type t2) {
  if (auto t = eval_const_divmod(t1, t2, cellDiv))
    return *t;
  return TInitPrim;
}

Type typeMod(Type t1, Type t2) {
  if (auto t = eval_const_divmod(t1, t2, cellMod))
    return *t;
  return TInitPrim;
}
```

Our tool would identify that the calls typeDiv and typeMod can be merged by introducing an extra parameter. Modulo variable renaming and indentation, this produces the following output:

```
template<class CellOp> Type typeModDiv(Type t1, Type t2, CellOp fun) {
  if (auto t = eval_const_divmod(t1, t2, fun))
    return *t;
  return TInitPrim;
}

Type typeDiv(Type t1, Type t2) { return typeModDiv(t1, t2, cellDiv); }
Type typeMod(Type t1, Type t2) { return typeModDiv(t1, t2, cellMod); }
```

[12] https://github.com/facebook/hhvm/

2.3.4 Optimizations

Although our core algorithm is generic enough to handle most cases of merging, our algorithm contains a few optimizations to improve the resulting merged code.

Going up the parent node

Our resolution patterns are restricted to a finite number and so not all kinds of differences can be resolved at the level at which they occur. Consider the following pieces of code:

```
x = y + z; //A
x = y - z; //B
```

Even though the difference is only the operator in the binary expression on the right hand side of the assignment, our algorithm currently supports no resolution pattern that can resolve such a case. Our algorithm considers the parent of the node, when differences cannot be resolved using an existing pattern. This step is used to make sure that the algorithm never aborts when no resolution pattern exists to resolve the kinds of differences in the nodes under consideration.

In this case, our algorithm would move up one parent level in both the clones. In which case, our algorithm would end up with two binary expressions where there is still no resolution pattern. After considering the parents for two levels, our algorithm reaches the assignment statement, which can be resolved either using a switch statement or a conditional branch. Because of the measure we just described, our algorithm never aborts because of the existence of the statement level resolution pattern and the fact that all methods bodies are compound statements.

In our earlier study of near-clones, we encountered only two kinds of expressions as differences, i.e literals and identifiers. We have chosen to handle only these two kinds of differences as they seem to occur dominantly in our study.

Sequence of line differences

Our algorithm does not handle the case where differences between trees happen in contiguous neighbor nodes. Consider the following near-clones.

```
//Clone 1
commonStat1();
statement1();
statement2();
commonStat2();

//Clone 2
commonStat1();
statement3();
statement4();
commonStat2();
```

Our algorithm would identify two merge points, one for the difference between **statement1** and **statement3**, and one for the difference between **statement2** and **statement4**. Applying the statement level resolution pattern twice would result in two conditions, or two switch statements, both of which always behave in the same way. Our algorithm performs an optimization to resolve such contiguous differences and treat them as one block to avoid multiple resolutions for contiguous statements. Our algorithm considers adjacent siblings of the same block which have a merge point at the statement level as contiguous.

Format Strings

Format string like **"string %s %d"** become invalid when they are passed as arguments or set to global variables. So, our algorithm treats this scenario as an unresolvable difference and continues by going up the parent one level at a time till it bumps into nodes that have types that are resolvable.

2.4 Implementation

In this section, we will look at the basics of how we implement our approach. We begin by discussing the libraries and frameworks we used and adapted in order to implement our tool followed by a discussion on the public availability of our tool.

Libraries and frameworks used in our implementation We have adapted an existing implementation of RTED[13] to fit our CDT AST representation. The existing implementation worked on in-order representations of trees in which nodes are labeled with strings. We adjusted the representation of nodes to contain information about AST node types and content.

Our merging tool is available to the user as an Eclipse plug-in in which we have extended the Refactoring menu to offer the merging tool as an option. Our tool currently works only on near-clone methods in the same source file, as we wanted to avoid the issue of where to place the merged method if the near-clones belong to different source files. Our tool presents all the available functions in the file in the currently active window using an input selection form. The user marks the near-clone methods he wishes to abstract using our tool's input form. The merging tool then produces a merged function and replaces the bodies of the existing functions with calls that invoke the merged function with appropriate arguments. Our approach also supports the option of changing all the call sites of the near-clone methods (even if they are in other files) in addition to that of replacing existing near-clone method bodies.

We have implemented the distance calculator, the algorithm and the framework on top of Eclipse CDT.[14]

[13] http://www.inf.unibz.it/dis/projects/tree-edit-distance/download.php
[14] https://eclipse.org/cdt/

Availability We have made our prototype publicly available.[15] Since October 2015, the version of our prototype that is available as a plugin in the Eclipse Marketplace has had 86 click-throughs and 36 installs, with no reported installation failures.[16]

2.5 Informal discussion on behaviour preservation of the algorithm

We now give an overview of why our algorithm with the current set of resolution patterns preserves behavior.

Recall that our algorithm performs the following transformations as part of the merging process:

- Generating a new merged method with a merged set of parameters (formal selectors), given as input any number of near-clone methods.
- Replacing definitions of the original near-clone methods using generated calls to the merged method with appropriate values for parameters (actual selectors). Alternatively, our algorithm replaces the call sites of the original near-clone methods with the appropriate generated calls. In both cases, the generated calls and the appropriate parameters are the same.

So, it is enough to discuss why each generated call to the merged method has the same behavior as the corresponding original near-clone method. We consider the formal and actual selectors generated for each resolution pattern. Then, we discuss how the selection mechanism uses the actual selector to reproduce the old code and discuss how the order of execution is preserved even in the presence of insertions. Finally, we summarize how our algorithm merges the parameter lists of all of the near-clone methods and maps the freshly generated parameters accordingly.

[15] http://sepl.cs.uni-frankfurt.de/~krishnanm86/clonemergeindex.html
[16] https://marketplace.eclipse.org/content/clone-abstractor-c-methods-0/metrics

Let us begin by considering what replaces the original function calls, for each possible type of selector. For switch statements and conditionals as selectors, the function signature will contain an extra parameter and the near-clone methods will pass their identifiers as actual selector to this parameter. For literals or variable identifiers that use an extra direct parameter as a selector, the function signature is updated with an extra parameter and the individual near-clones pass the actual literals or pointers to these identifiers as the actual selector. For template selectors, the function signature will contain an extra template parameter and the near-clone functions will pass the type as the actual selector. Alternatively, for all the cases other than the template selector, our algorithm also provides the option of using a global variable as the formal selector. Using global variables as selectors is only correct if the relevant method does not contain recursive calls because recursive calls may alter the value of global variables. The rest of the section discusses the behavior preservation of cases that donot involve global variables.

We have discussed the replacement of actual selectors with selection mechanisms in Section 2.3.3.3. The special cases that require further discussion are the treatment of replacing a constant by a variable introduced as a parameter and replacing an L-values of an assignment expression with a reference. We replace constants by freshly introduced function parameters having the type of the constant. In case of pointers, we use a '&' or a '*' when applying the selection mechanism, depending on whether the use of the variable is on the left hand side or the right hand side of an expression. The selection mechanism produces the appropriate statement, constant or variable identifier when the appropriate selector is applied given how we produce the selectors and the selection mechanism in each case.

Observe that we preserve the order of statements in the AST in all cases. Whenever there is an inserted statement, the relevant statements are guarded by a selection mechanism. When there is a node 'n' that is inserted when moving from AST_x to AST_y, then there is repositioning of nodes in AST_y. But, the selection mechanism ensures that the positions of these nodes are retained based on the value of the actual selector passed by the near-clone methods. We refer the reader to the detailed discussion of the algorithm in Section 2.3.

Merging existing parameters of near-clone methods Our algorithm constructs the parameter list of the merged method as the union of the parameters of the near-clone methods. Two parameters are considered equal if their names and types are the same. Two parameters with the same name and different types are resolved using the **type difference** resolution pattern and a newly generated template type is used to resolve the difference. For example, consider the processing of the **RethinkDB** clone group in Section 2.3.3.6. Our algorithm merged the parameter list and produced the following merged method signature:

```
template<typename T> cJSON *cJSON_CreateNumArray(T numbers,int count) ;
```

Our algorithm also generates dummy values for the parameters in the generated merged method that have no equivalent from a corresponding near-clone method. Dummy values are chosen based on the type of the parameters, *e.g.*, 0 for an integer-typed parameter. As these arguments will not be used in cases where a dummy value is provided, the value is not important.

2.6 Evaluation

We have evaluated our approach by exploring the following research question:

RQ: Are the abstractions performed by our algorithm of sufficient quality for production level code?

In order to evaluate this question, we first looked for clone group candidates to merge. Our initial study in Section 2.1 was performed at an earlier month than the evaluation performed here and therefore, we had to re-run our clone detector to choose groups to merge. We explored top trending Github repositories, identified potential candidates for merging using our RTED inspired clone detector, and abstracted the identified candidates using our approach.We finally submitted the abstracted code back to the developers through pull requests, to see how many of them were of sufficient quality to be introduced back into production code. We performed a total of 18 abstractions of clone groups from the top trending GitHub

repositories that we identified previously and sent pull requests to the repositories from which we got the code. **Table 2.2** lists the repositories that we considered in our evaluation along with our pull request URLs, the number of clone groups abstracted per repository, and the status of the pull requests.

2.6.1 Identifying and Merging Clone Groups

The clone group candidates for our approach are those with near clones. We started with the repositories in **Table 2.2** and collected all method pairs belonging to the same source file.

We began by computing the edit distance of each pair. We called a function pair a near-clone function pair if the number of nodes in the smaller of the two functions (*#fnSmaller*) was greater than a customizable $threshold_n$ and if the ratio of the edit distance to *#fnSmaller* was less than a customizable $threshold_r$, where $threshold_n$ is a positive integer and $threshold_r$ is a positive float between 0 and 1.

We collect the near-function pairs into sets such that every function in each set forms a near-clone function pair with every other function inside the set. We call such sets of methods whose bodies are closely reated to each other 'clone groups'. We then randomly picked clone groups. Each clone group we picked contained 2–4 functions. We then merged the clone groups, using a predetermined resolution pattern for each node type, and submitted pull requests. We chose the following resolution patterns for specific node type differences:

- We resolved differences in statements using a switch and as a selector, an extra method parameter specifying the switch branch to choose (Pattern - Statement level differences). We could have chosen to try the conditional pattern in cases where there are only two choices. We believe a switch statement offers developers the opportunity to expand the merged method into more cases in the future more easily than adding more branches to an existing conditional.

TABLE 2.2: Repositories with their pull request URLs. Each clone group represents one abstraction. We encourage readers who choose to look at the pull requests to go through the comments. While some of the pull requests do not explicitly have their status listed as 'merged', as with the OracleDB and the MongoDB repositories, the code has actually been merged, as indicated by the maintainer comments.

Repository	**Phase**	**Clone Groups**	**Status**
ideawu/ssdb	2	1	
https://github.com/ideawu/ssdb/pull/609			Rejected
facebook/rocksdb	1	1	
https://github.com/facebook/rocksdb/pull/440/			Pending
openexr/openexr	1	3	
https://github.com/openexr/openexr/pull/147			Pending
facebook/hhvm	1	1	
https://github.com/facebook/hhvm/pull/4490			Rejected
google/protobuf	1	2	
https://github.com/google/protobuf/pull/128			**Accepted**
https://github.com/google/protobuf/pull/126			Rejected
SFTtech/openage	1	1	
https://github.com/SFTtech/openage/pull/176			Rejected
oracle/node-oracledb	2	3	
https://github.com/oracle/node-oracledb/pull/28			**Accepted**
mongodb/mongo	2	2	
https://github.com/mongodb/mongo/pull/927			**Accepted**
https://github.com/mongodb/mongo/pull/928			**Accepted**
rethinkdb/rethinkdb	2	2	
https://github.com/rethinkdb/rethinkdb/pull/3820			**Accepted**
https://github.com/rethinkdb/rethinkdb/pull/3818			**Accepted**
cocos2d/cocos2d-x	2	2	
https://github.com/cocos2d/cocos2d-x/pull/10539			**Accepted**
https://github.com/cocos2d/cocos2d-x/pull/10546			**Accepted**

- We resolved differences in literal expressions (constants) by passing additional parameters (Pattern - Literal differences). We could have chosen to use a global variable, but we believe that using an extra parameters is less intrusive to the existing code and is expected to be preferred by the maintainers of the repositories accepting the pull requests.

- We resolved differences in type expressions using templates (Pattern - Type differences).

- We resolved differences in identifier expressions using additional parameters (promoted to pointers for LValues), and formal parameters specifying the identifier or the address of the variable (Pattern - Variable identifier differences).

We also performed minor manual changes. These include:

- Providing meaningful names for parameters. Our tool generates random fresh names based on the position of the merge points. These names are not suitable for production code.
- Adding function prototypes to header files whenever requested by maintainers.

We added the function prototypes after discussion with the maintainers who had previously looked at our tool generated merges. These manual changes are standard refactorings that are not central to our approach.

2.6.2 Results

Our evaluation involved two phases. The first phase served as a validation to show that our tool can abstract near clones in real code. The first phase also gave us information about what resolution patterns are preferred by developers. We used the insights from the first phase in the second phase to apply our tool on clone group abstractions that were more likely to be preferred by developers. The second phase of our evaluation illustrated the industry acceptability of the abstractions produced our tool.

We performed our initial evaluation (Phase 1) using an early version of our merging tool that could only merge pairs of methods and did not support multiple resolution patterns for the same pair, i.e if the functions had more than one merge point, they all had to be of the same type nodes. During Phase 1, we ran our distance calculator on the top trending C++ repositories in Github for the month

of December 2014, and selected potential clone groups by setting $threshold_r$ to 0.5 and $threshold_n$ to 0, meaning that we considered functions of all sizes. We submitted 8 abstractions as pull requests and only one of the clone groups was **Accepted**. The results of the pull requests highlighted areas of improvement needed in our first prototype.

TABLE 2.3: Phase 1 results summary

Submitted	**Accepted**	**Rejected**	**Pending**
8	1	3	4

We performed our second evaluation (Phase 2) using a complete version of the our merging tool, capable of merging an arbitrary number of methods at the same time. This version also supported resolving multiple merge points with each merge point comprising of different node types. During Phase 2, we ran our distance calculator on the top trending repositories for the month of February 2015. We set $threshold_r$ to 0.15 and $threshold_n$ to 100. We changed the thresholds building on experience from Phase 1 in order to focus on clone groups that would save more lines of code when abstracted than the existing duplicate versions. The clones in Phase 2 were very similar to each other and tied to methods of substantial sizes. We then submitted 10 abstractions as pull requests, summarized in the table below, and found that all but one were **Accepted**:

TABLE 2.4: Phase 2 results summary

Submitted	**Accepted**	**Rejected**	**Pending**
10	9	1	0

We conclude that the repository maintainers found our code to be of sufficient quality (including readability and maintainability) for inclusion. Specifically, we observed no negative comments regarding readability in any of the comments that we received.

2.6.3 Analysis of Rejected and Pending Results

We present the results of the pending and rejected pull requests summarized in **Table 2.2** and provide our analysis of the these results.

2.6.3.1 Pending results

We begin with the feedback to pull requests that were neither **Accepted** nor Rejected. Let us first discuss the pending pull request from RocksDB. The comment from the head maintainer of the project was:
"Great stuff, now its only one commit (after the squash)! Waiting for OK from @anon1 or @anon2 (since they maintain this code) before merging."
We interpret that the pull request was met with positive review. We did check later with the maintainers of the repository to no avail. We suspect that developers have many tasks and only one of them is attending to pull requests; our patch may not be their top priority.

The other pending pull request was from the OpenExr repository. The request merged three clone groups at once, and received a mixture of responses. One maintainer requested an explanation of the advantages. Another maintainer expressed skepticism over the performance overhead of such an abstraction, as it was a low level function. A third maintainer requested a unit test of the introduced abstraction before a merge. We could not satisfy these requests due to a lack of understanding of the semantics of the functions we had merged. All these activities took place over a 3 month period.

2.6.3.2 Rejected results

Of the five rejected clone group abstractions, four were rejected because the maintainers felt that not enough lines were saved. We did not receive an explanation for the rejected clone group abstraction for the Ideawu/ssdb repository.

2.7 Full Repository Evaluation - GIT

While our earlier two sets of experiments illustrated the utility that our tool provides in realistic scenarios, we biased our selection through the use of a clone detector whose similarity metric is closely related to our merging algorithm. To explore whether this bias is a concern in practice, we ran a third experiment with a mainstream off-the-shelf clone detector. We selected Nicad [21] as the clone detector. NiCad does not support C++, but it does support C; since our system is based on the Eclipse CDT, we can also use it on C code, as long as we disable abstraction patterns that are based on C++-only language features.

As the target program we therefore selected one of the top trending C repositories on github, the Git[17] revision control system. At the time of our experiment, Git had 6251 functions. We configured NiCad for our experiment as follows:

Granularity:	functions
Max difference threshold:	30%
Clone size:	20 - 2500 lines

NiCad detected 5 clone groups. In the following, We describe each clone group, as well as the results of merging the functions in each of these clone groups, and the insights that we obtained from each merge.

Clone Group 1

The first clone group contained two functions, **int_obstack_begin_1** and **int_obstack_begin**, that differed by one constant (Line **11** in **int_obstack_begin_1** and Line **9** in **int_obstack_begin**), one extra argument **arg** in **int_obstack_begin_1** and one statement that was present only in **int_obstack_begin**.

```
1 int _obstack_begin_1
2         (struct obstack *h, int size, int alignment,
3                   void *(*chunkfun) (void *, long),
4                   void (*freefun) (void *, void *),
```

[17] https://github.com/git/git

```
5           void *arg)
6 {
7   /*Common Lines */
8
9   h->alignment_mask = alignment - 1;
10  h->extra_arg = arg;
11  h->use_extra_arg = 1;
12
13  chunk = h->chunk =  CALL_CHUNKFUN(h, h->chunk_size);
14
15  /*Common Lines */
16 }
```

```
1  int _obstack_begin
2          (struct obstack *h, int size, int alignment,
3                   void *(*chunkfun) (void *, long),
4                   void (*freefun) (void *, void *))
5  {
6    /*Common Lines */
7
8    h->alignment_mask = alignment - 1;
9    h->use_extra_arg = 0;
10
11   chunk = h->chunk = CALL_CHUNKFUN(h, h->chunk_size);
12
13   /*Common Lines */
14 }
15
16
```

Our tool resolves the difference in the constant values by introducing an additional parameter **parameter**, and using it in place of the constnats. It further resolves the optional statement by introducing a **switch** statement around the optional line (`h->$extra_arg = arg;`), plus a second new parameter **functionId** as the formal selector for the **switch** statement:

```
int
_obstack_begin_merged (struct obstack *h, int size, int alignment,
                  void *(*chunkfun) (void *, long),
                  void (*freefun) (void *, void *),
                  void *arg, int functionId, int parameter)
{
  /*Common Lines */

  h->alignment_mask = alignment - 1;
```

```
  //Generated Switch statement
  switch(functionId)
  {
         case 1:
                h->extra_arg = arg;
                break;
  }
  h->use_extra_arg = parameter; //Generated extra parameter

  chunk = h->chunk = CALL_CHUNKFUN (h, h -> chunk_size);

  /*Common Lines */
}

int_obstack_begin_1(struct obstack *h, int size, int alignment,
                    void *(*chunkfun) (void *, long),
                    void (*freefun) (void *, void *),
                    void *arg)
{
   _obstack_begin_merged(h, size, alignment, chuckfun, freefun, arg, 1, 1);
}

int_obstack_begin(struct obstack *h, int size, int alignment,
                    void *(*chunkfun) (void *, long),
                    void (*freefun) (void *, void *))
{
   _obstack_begin_merged(h, size, alignment, chuckfun, freefun, null, 2, 0);
}
```

Insights: In this example, the values of **parameter** and **functionId** depend on each other, meaning that the two parameters could be merged into one. We envision that a future version of our tool could re-use selectors in multiple selection mechanisms.

Clone Group 2

The second clone group contains 5 statement level differences, which our tool merges using switch statements.

```
static void command_loop(int input_fd, int output_fd) {
    char buffer[MAXCOMMAND];
    while (1) {
        size_t i;
        if (!fgets(buffer, MAXCOMMAND - 1, stdin)) {
```

```
            if (ferror(stdin))
                die("Input error");
            return;
        }
        i = strlen(buffer);
        while (i > 0 && isspace(buffer[i - 1]))
            buffer[--i] = 0;
        if (!strcmp(buffer, "capabilities")) {
            printf("*connect\n\n");
            fflush(stdout);
        } else if (!strncmp(buffer, "connect ", 8)) {
            printf("\n");
            fflush(stdout);
            if (bidirectional_transfer_loop(input_fd, output_fd))
                die("Copying data between descriptors failed");
            return;
        } else {
            die("Bad command: %s", buffer);
        }
    }
}

static int command_loop(const char * child) {
    char buffer[MAXCOMMAND];
    while (1) {
        size_t i;
        if (!fgets(buffer, MAXCOMMAND - 1, stdin)) {
            if (ferror(stdin)) die("Command input error");
            exit(0);
        } /* Strip end of line characters. */
        i = strlen(buffer);
        while (i > 0 && isspace(buffer[i - 1])) buffer[--i] = 0;
        if (!strcmp(buffer, "capabilities")) {
            printf("*connect\n\n");
            fflush(stdout);
        } else if (!strncmp(buffer, "connect ", 8)) {
            printf("\n");
            fflush(stdout);
            return run_child(child, buffer + 8);
        } else {
            fprintf(stderr, "Bad command");
            return 1;
        }
    }
}
```

Our tool also creates a union of the parameters of the two functions, mapping the arguments to the ones in the merged function appropriately and passing dummy values otherwise. The near-clone functions have 2 parameters and 1 parameter respectively and the merged function has 3 parameters.

```
static int command_loop_merge(int input_fd, int output_fd, const char * child,
                              char * str1, int functionId) {
    char buffer[MAXCOMMAND];

    while (1) {
        size_t i;
        if (!fgets(buffer, MAXCOMMAND - 1, stdin)) {
            if (ferror(stdin))
                die(str1);
            switch (functionId) {
            case 1:
                return 1;
            case 2:
                exit(0);
            }
        }
        /* Strip end of line characters. */
        i = strlen(buffer);
        while (i > 0 && isspace(buffer[i - 1]))
            buffer[--i] = 0;

        if (!strcmp(buffer, "capabilities")) {
            printf("*connect\n\n");
            fflush(stdout);
        } else if (!strncmp(buffer, "connect ", 8)) {
            printf("\n");
            fflush(stdout);
            switch (functionId) {
            case 1:
                if (bidirectional_transfer_loop(input_fd, output_fd))
                    die("Copying data between file descriptors failed");
                break;
            case 2:
                return run_child(child, buffer + 8);
                break;
            }

        } else {
            switch (functionId) {
            case 1:
                die("Bad command: %s", buffer);
```

```
                break;
            case 2:
                fprintf(stderr, "Bad command");
                return 1;
            }
        }
    }
}

static void command_loop(int input_fd, int output_fd) {
    command_loop_merge(intput_fd, output_fd, null, "Input error", 1);
}

static int command_loop(const char * child) {
    command_loop_merge(0, 0, child, "Command input error", 2);
}
```

Insights: This clone group may not be a good candidate for merging as the number of lines in the merged code is similar to the number of lines in the existing version of the methods. But, we still performed the merge in order to be able to report on insights gathered from merging all clone groups detected by a mainstream clone detector without any bias. Since the return types of the functions in the clone group are different and C does not support templates, we had to manually modify the code so that both functions have the same return type. Transforming a void function so that it returns an integer only requires adding a dummy return value, so we took this option. The manual effort for performing this transformation took about 4 minutes. We could have used the resolution pattern for type level differences, if the code were written in C++.

We also observed that our tool is unable to detect commonalities and differences inside strings. For example, when generating the calls to the merged function, we would have preferred to only pass the stringss **"Command input"** and **"Input"**, instead of the strings **"Command Input Error"** and **"Input Error"**. Since we were targetting C, such reuse would have required us to introduce additional print statements or formatted prints. Currently, our prototype does not support this form of merging.

Clone Group 3

The third clone group, much like the previous example, contains 5 statement level differences.

```
static int keyring_get(struct credential *c)
{
        char *object = NULL;
        GList *entries;
        GnomeKeyringNetworkPasswordData *password_data;
        GnomeKeyringResult result;

        if (!c->protocol || !(c->host || c->path))
                return EXIT_FAILURE;

        /* Common Lines */

        /* pick the first one from the list */
        password_data = (GnomeKeyringNetworkPasswordData *)entries->data;
        gnome_keyring_memory_free(c->password);
        c->password = gnome_keyring_memory_strdup(password_data->password);
        if (!c->username)
                c->username = g_strdup(password_data->user);
                gnome_keyring_network_password_list_free(entries);

        return EXIT_SUCCESS;
}

static int keyring_erase(struct credential *c)
{
        char *object = NULL;
        GList *entries;
        GnomeKeyringNetworkPasswordData *password_data;
        GnomeKeyringResult result;

        if (!c->protocol && !c->host && !c->path && !c->username)
                return EXIT_FAILURE;

        /* Common Lines */

        /* pick the first one from the list (delete all matches?) */
        password_data = (GnomeKeyringNetworkPasswordData *)entries->data;
        result = gnome_keyring_item_delete_sync(
                password_data->keyring, password_data->item_id);
        gnome_keyring_network_password_list_free(entries);
        if (result != GNOME_KEYRING_RESULT_OK) {
                g_critical("%s", gnome_keyring_result_to_message(result));
```

```
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

Our tool merges the differences using switch statements. The parameter lists match with both near-clone functions containing one parameter and so the abstracted method contains only one extra parameter, which allows to switch between the two clone functions.

```
static int keyring_get_merge(struct credential *c, int functionId) {
    char *object = NULL;
    GList *entries;
    GnomeKeyringNetworkPasswordData *password_data;
    GnomeKeyringResult result;
    switch (functionId) {
    case 1:
        if (!c->protocol || !(c->host || c->path))
            return EXIT_FAILURE;
        break;
    case 2:
        if (!c->protocol && !c->host && !c->path && !c->username)
            return EXIT_FAILURE;
        break;
    }
    /* Common Lines */

    switch (functionId) {
    case 1:
        gnome_keyring_memory_free(c->password);
        c->password = gnome_keyring_memory_strdup(password_data->password);

        if (!c->username)
            c->username = g_strdup(password_data->user);

        gnome_keyring_network_password_list_free(entries);
        break;
    case 2:
        password_data = (GnomeKeyringNetworkPasswordData *) entries->data;

        result = gnome_keyring_item_delete_sync(
            password_data->keyring, password_data->item_id);

        gnome_keyring_network_password_list_free(entries);
```

```
        if (result != GNOME_KEYRING_RESULT_OK) {
            g_critical("%s", gnome_keyring_result_to_message(result));
            return EXIT_FAILURE;
        }
        break;
    }
    return EXIT_SUCCESS;
}

static int keyring_get(struct credential *c) {
    return keyring_get_merge(c, 1);
}
static int keyring_erase(struct credential *c) {
    return keyring_get_merge(c, 2);
}
```

Insights: The conditional at the beginning of each function in the clone group differs only in the conditional expression. However, our tool does not presently support expression-level differences, unless they are on constants or identifiers. Our tool manages this situation by pulling the merge point up to the smallest surrounding syntatic entity whose AST node type we support — the surrounding **if** statement — and applies a suitable abstraction pattern. Recall that we discuss this optimization step of 'going up a parent node' in Section 2.3.4. Our tool is also unable to resolve the duplicate **return EXIT_FAILURE** statements because of it's presence inside a conditional branch that differs.

Clone Group 4

The fourth clone group contains one statement level difference and one statement inserted into the input near-clone function **string_list_split_in_place**. Both differences are resolved using a switch statement.

```
1 int string_list_split(struct string_list *list, const char *string,
2                       int delim, int maxsplit)
3 {
4         int count = 0;
5         const char *p = string, *end;
6
7         if (!list->strdup_strings)
8                 die("internal error in string_list_split(): "
```

```
                    "list->strdup_strings must be set");
        for (;;) {
                count++;
                if (maxsplit >= 0 && count > maxsplit) {
                        string_list_append(list, p);
                        return count;
                }
                end = strchr(p, delim);
                if (end) {
                        string_list_append_nodup(list, xmemdupz(p, end - p));
                        p = end + 1;
                } else {
                        string_list_append(list, p);
                        return count;
                }
        }
 }
```

```
int string_list_split_in_place(struct string_list *list, char * string,
    int delim, int maxsplit) {
    int count = 0;
    char * p = string, * end;
    if (list->strdup_strings)
        die("internal error in string_list_split_in_place(): "
            "list->strdup_strings must not be set");
    for (;;) {
        count++;
        if (maxsplit >= 0 && count > maxsplit) {
            string_list_append(list, p);
            return count;
        }
        end = strchr(p, delim);
        if (end) {
            *end = '\0';
            string_list_append(list, p);
            p = end + 1;
        } else {
            string_list_append(list, p);
            return count;
        }
    }
}
```

```
int string_list_split_merge(struct string_list *list,
    const char *string,
        int delim, int maxsplit, int functionId) {
    int count = 0;
```

```
    const char *p = string, *end;
    switch (functionId) {
    case 1:
        if (!list->strdup_strings)
            die("internal␣error␣in␣string_list_split():␣"
                "list->strdup_strings␣must␣be␣set");

        break;
    case 2:
        if (list->strdup_strings)
            die("internal␣error␣in␣string_list_split_in_place():␣"
                "list->strdup_strings␣must␣not␣be␣set");
        break;
    }
    for (;;) {
        count++;
        if (maxsplit >= 0 && count > maxsplit) {
            string_list_append(list, p);
            return count;
        }
        end = strchr(p, delim);
        if (end) {
            switch (functionId) {
            case 1:
                string_list_append_nodup(list, xmemdupz(p, end - p));
                break;
            case 2:
                *end = '\0';
                string_list_append(list, p);
                break;
            };
            p = end + 1;
        } else {
            string_list_append(list, p);
            return count;
        }
    }
}

int string_list_split(struct string_list *list,
    const char *string,
        int delim, int maxsplit) {
    string_list_split_merge(list, string, delim, maxsplit, 1);
}
int string_list_split_in_place(struct string_list *list,
    const char *string,
        int delim, int maxsplit) {
    string_list_split_merge(list, string, delim, maxsplit, 2);
```

```
}
```

Insights: Although the tool detects the difference as one statement in the left hand side at line 18 of the function **string_list_split** and two statements at lines 15 and 16 of **string_list_split_in_place**, the post processing phase merges the two statement differences into a single one as they occur one after the other.

Clone Group 5

The fifth clone group contains two statement level differences and one constant difference. Our tool introduces two extra parameters, one, **functionId**, to switch between the statements based on the clone functions calling the merged function, and another, **str1**, which our tool detects, is of type **char***.

```
int git_inflate(git_zstream *strm, int flush)
{
        int status;

        for (;;) {
                zlib_pre_call(strm);
                /* Never say Z_FINISH unless we are feeding everything */
                status = inflate(&strm->z,
                                 (strm->z.avail_in != strm->avail_in)
                                 ? 0 : flush);
                if (status == Z_MEM_ERROR)
                        die("inflate: out of memory");
                zlib_post_call(strm);

                /*
                 * Let zlib work another round, while we can still
                 * make progress.
                 */
                if ((strm->avail_out && !strm->z.avail_out) &&
                    (status == Z_OK || status == Z_BUF_ERROR))
                        continue;
                break;
        }

        switch (status) {
        /* Z_BUF_ERROR: normal, needs more space in the output buffer */
        case Z_BUF_ERROR:
        case Z_OK:
```

```
        case Z_STREAM_END:
                return status;
        default:
                break;
        }
        error("inflate: %s (%s)", zerr_to_string(status),
              strm->z.msg ? strm->z.msg : "no message");
        return status;
}

int git_deflate(git_zstream *strm, int flush)
{
        int status;

        for (;;) {
                zlib_pre_call(strm);

                /* Never say Z_FINISH unless we are feeding everything */
                status = deflate(&strm->z,
                                 (strm->z.avail_in != strm->avail_in)
                                 ? 0 : flush);
                if (status == Z_MEM_ERROR)
                        die("deflate: out of memory");
                zlib_post_call(strm);

                /*
                 * Let zlib work another round, while we can still
                 * make progress.
                 */
                if ((strm->avail_out && !strm->z.avail_out) &&
                    (status == Z_OK || status == Z_BUF_ERROR))
                        continue;
                break;
        }

        switch (status) {
        /* Z_BUF_ERROR: normal, needs more space in the output buffer */
        case Z_BUF_ERROR:
        case Z_OK:
        case Z_STREAM_END:
                return status;
        default:
                break;
        }
        error("deflate: %s (%s)", zerr_to_string(status),
              strm->z.msg ? strm->z.msg : "no message");
        return status;
}
```

```
int git_inflate_deflate(git_zstream * strm, int flush, char * str1, int functionId) {
    int status;

    for (;;) {
        zlib_pre_call(strm);
        /* Never say Z_FINISH unless we are feeding everything */
        switch (functionId) {
        case 1:
            status = inflate( & strm->z,
                (strm->z.avail_in != strm->avail_in) ? 0 : flush);

            break;
        case 2:
            status = deflate( & strm->z,
                (strm->z.avail_in != strm->avail_in) ? 0 : flush);

            break;
        };

        if (status == Z_MEM_ERROR)
            die(str1);
        zlib_post_call(strm);

        /*
         * Let zlib work another round, while we can still
         * make progress.
         */
        if ((strm->avail_out && !strm->z.avail_out) &&
            (status == Z_OK || status == Z_BUF_ERROR))
            continue;
        break;
    }

    switch (status) {
        /* Z_BUF_ERROR: normal, needs more space in the output buffer */
    case Z_BUF_ERROR:
    case Z_OK:
    case Z_STREAM_END:
        return status;
    default:
        break;
    }
    switch (functionId) {
    case 1:
        error("inflate: %s (%s)", zerr_to_string(status),
            strm->z.msg ? strm->z.msg : "no message");
```

```
        break;
    case 2:
        error("deflate: %s (%s)", zerr_to_string(status),
            strm->z.msg ? strm->z.msg : "no message");

        break;
    };

    return status;
}

int git_inflate(git_zstream * strm, int flush) {
    return git_inflate_deflate(strm, flush, "inflate: out of memory", 1);
}
int git_deflate(git_zstream * strm, int flush) {
    return git_inflate_deflate(strm, flush, "deflate: out of memory", 2);
}
```

Insights: As we have previously noted, the tool is unable to detect differences within strings. While we would ideally have passed only the strings **"inflate"** and **"deflate"**, our tool considers the whole string as a difference. Our tool also considers strings that contain format directives such as % to be differences for which there is no resolution pattern and moves up one level to resolve the differences. Such strings are likely to be part of a function call, and typically coding-style conventions discourage or forbid using a parameter in a function argument position where a format string is expected. We have described the optimization step where our algorithm moves up one level in Section 2.3.4.

Overall, we found that our tool can be integrated with a mainstream clone detector as a clone removal mechanism with minimal manual effort.

Chapter 3

Source code Reuse - Merging clone instances back into existing abstractions

Chapter Overview

The merging algorithm described in Section 2.3 is only able to merge near clone instances with each other and produce the merged code. We will refer to the algorithm described in Section 2.3 as **CloneMerge**. For **CloneMerge** to work, all clone instances in a clone group must be available at the same time. However, clone instances might not appear all at once. The previous technique begins with a clone group (c_1, c_2, ... c_n) and produces a merged method c_m. What if a method c_{n+1} is written by a new developer who is unaware of the previous methods or the available merge c_m? The developer could benefit from the merging of c_{n+1} with c_m.

One approach would be to attempt to merge the clone instance directly with the existing abstraction using **CloneMerge**. But **CloneMerge** cannot be used to merge clone instances with existing abstractions. In this chapter, we begin by summarizing how **CloneMerge** works and describe why using the same approach

to merge clone instances into existing abstractions would cause an issue. We discuss the challenges of merging clone instances into existing abstractions using real clone merges from popular open source repositories in GitHub. We use the motivating examples to develop an approach to detect if a newly introduced code segment is an instance of the available abstractions in the existing source repository. We also develop an approach to merge the clone instances into the merged code. We evaluate our approach by checking out real code samples from popular open source repositories which have gone through clone merges that were created by our existing clone merging tool. We identify the clone groups that have gone through the clone merges, remove one of the clones from these groups and attempt to merge the removed clone instance back into the merged code. We validate the results by comparing the results produced by our tool with the previously available merge to check for similarity of our merged code with the existing merged code.

The rest of the chapter is organized as follows. Section 3.1 motivates the need for a separate algorithm using a few examples from real open source repositories. Section 3.2 presents an algorithm to merge clone instances into existing abstractions. Section 3.3 walks through the algorithm on real examples to further ease the understanding of the algorithm. Section 3.4 validates the algorithm by applying the implemented tool on real examples.

3.1 Motivation

CloneMerge merges method clone instances into a merged method. In order to motivate the need for a separate merging algorithm for merging a clone instance into an existing merged method, we use the following clone group merge to summarize how the **CloneMerge** approach works:

```
1 void fn1(){
2         int x = 10;
3         int y = x * 100;
4         int z = fnd(x,y);
5         fna(z);
6 }
```

```
7
8  void fn2(){
9          int x = 20;
10         int y = x * 200;
11         float z = fnd(x,y);
12         fnb(z);
13 }
14
15 void fn3(){
16         int x = 30;
17         int y = x * 300;
18         double z = fnd(x,y);
19         fnc(z);
20 }
```

CloneMerge would identify the common code and the merge points. The merge points would be:

1. Between the constants **10, 20, 30** in lines 2, 9 and 16 of the functions **fn1, fn2, fn3** respectively.
2. Between the constants **100, 200, 300** in lines 3, 10 and 17 of the functions **fn1, fn2, fn3** respectively.
3. Between the types **int, float, double** in lines 4, 11 and 18 of the functions **fn1, fn2, fn3** respectively.
4. Between the function call statements **fna(z), fnb(z), fnc(z)** in lines 5, 12, 19 of **fn1, fn2, fn3** respectively.

The existing tool, along with user choices for selection mechanisms would generate the merged method. Let us assume the selection mechanisms chosen were :

1. An extra parameter for merge point 1
2. A global variable for merge point 2
3. A template parameter for merge point 3
4. A switch statement for merge point 4

```
1  /*Definitions of fna, fnb, fnc and fnd omitted for brevity*/
2  int globalConst;
3  template<typename T1>
4  void fnMerged(int constant, int functionId) {
5          int x = constant;
6          int y = x * globalConst;
7          T1 z = fn(x, y);
8          switch (functionId) {
9                  case 1:
10                         fna(z);
11                         break;
12                 case 2:
13                         fnb(z);
14                         break;
15                 case 3:
16                         fnc(z);
17                         break;
18         }
19 }
20
21 void fn1() {
22         globalConst = 100;
23         fnMerged<int>(10, 1);
24 }
25
26 void fn2() {
27         globalConst = 200;
28         fnMerged<float>(20, 2);
29 }
30
31 void fn3() {
32         globalConst = 300;
33         fnMerged<double>(30, 3);
34 }
```

FIGURE 3.1: Example clone group merge

The resulting merged method along with the generated calls to the merged method from the existing clone methods are shown in Figure 3.1:

Now, let us assume that new method **fn4** is introduced as shown below:

```
1 void fn4(){
2         int x = 40;
3         int y = x * 400;
4         char z = fn(x, y);
5         fnd();
6 }
```

The developer of **fn4** cannot use **CloneMerge** to merge **fn4** with **fnMerged**, as **CloneMerge** expects that the type of nodes for all the merge points be the same. Consider the merge point between **40** in **Line 2** of **fn4** and **constant** in **Line 4** of **fnMerged**. Their node types are different (constant and identifier respectively). Although, both constants and identifiers in this context are both expressions, our system would differentiate the nodes since the constant would be

a constant expression and the identifier would be an identifier expression. The node types in our system directly correlate to the node types from Eclipse CDT.[1] Furthermore, **CloneMerge** works by resolving merge points by creating extra parameters. The developer needs to identify that **40** needs to passed as value to the existing parameter **constant**.

The differing types of nodes in merge points are not the only challenge when merging clone instances into merged methods. The current algorithm does not work for merging clone instances into existing abstractions, even if the types of nodes in the merge points are the same. For example, the function call statement **fnd()** in line 4 of the **fn4** and the switch statement in line 7 of **fnMerged** are both statements. Merging these two statements with **CloneMerge** will create a nested switch statement like this:

```
switch(functionId2){
        case 1:
                fnd();
                break;
        case 2:
        switch (functionId) {
                case 1:
                        fna(z);
                        break;
                case 2:
                        fnb(z);
                        break;
                case 3:
                        fnc(z);
                        break;
        }
        break;
}
```

The developer needs to identify that the statement in fn4 is a case of the existing switch statement in the merged function.

[1] http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcdt%2Fcore%2Fdom%2Fast%2FIASTNode.html

3.2 Algorithm

In this section, we begin by summarizing the insights on what we believe are needed to make the merging of clone instances with existing abstractions work. We follow that by a summary of our algorithm, **CloneMergeExt** that merges clone instances into an existing abstraction followed by a walk-through of the algorithm on a synthetic example.

Requirements for merging clone instances with an existing abstraction Our algorithm in **CloneMerge** uses the terminology *selection mechanisms* to describe the the resolution strategy when merging nodes of a particular type. For examples, when merging three constants **10**, **20** and **30** using a global variable of type **int**, the selection mechanism is global variables. Similarly, Switch statements can serve as a selection mechanism for resolving differences between statements. Templates can serve as selection mechanisms for differences between types. **CloneMergeExt** needs to merge nodes in clone instances into existing selection mechanisms in the existing abstraction.

Summary of steps in CloneMergeExt:

1. **CloneMergeExt** takes as input a potential merge method (existing abstraction) and a clone instance method that is attempted to be merged with the existing abstraction. In our example in Figure 3.1, the merged method is **fnMerged** and the clone instance is **fn4**.

2. **CloneMergeExt** needs to keep track of which arguments of the merged method (or templates or global variables) maps to which value (actual selector) in the clone instance provided as input. For this purpose, **CloneMergeExt** creates a map $\mathbf{Map}_{param}$ of actual parameters in the merged method to corresponding values for generating the call from the clone instance to the merged method. The actual parameters comprise the actual nodes which need to be updated along with the names of the parameter. The actual

parameter nodes will come in handy when handling situations like global variables or templates. To give an example of such a mapping, when merging **fn4** with **fnMerged** in Figure 3.1, the value **40** from **fn4** needs to be mapped to the parameter **constant**, the first argument to **fnMerged**.

3. Detect the merge points (using part of our previous approach). Each merge point is a pair comprising the differing nodes in the clone abstraction and the corresponding differing nodes in the clone instance.

4. For each merge point, verify if the differing node in the clone abstraction can serve as a *selection mechanism* to the node in the clone instance. This verification is dependent on the resolution pattern and needs to be extended to handle new types of differences and resolution patterns. If **CloneMerge** is extended into new resolution patterns, then this verification in **CloneMergeExt** should also be extended.

    - If the differing nodes in the clone instance are constants, variable references, then **CloneMergeExt** verifies if the corresponding differing node in the clone abstraction is a parameter to the merged function (or a global variable) and if the differing node in the clone instance is of the same type as the parameter of the global variable. This is because global variables or direct parameters of a particular type can serve as selection mechanisms for constants or variable references of the same type.
    - If the differing nodes in the clone instance are types, then **CloneMergeExt** verifies if the corresponding differing node in the clone abstraction is a reference to a template type. This is because template types can serve as selection mechanisms for type variables.
    - If the differing nodes in the clone instance are statements, then **CloneMergeExt** verifies if the differing node in the clone abstraction is a switch statement. This is because Switch statements can serve as selection mechanisms for statements.

It is to be noted that the verification beginning at the clone instance and identifying the relevant abstraction patterns works only with these three examples. Extensions into other resolution pattern may require other forms of verification. A verification module will in essence be provided as input the differing node in the clone instance and the differing node in the clone abstraction.

5. If the verification fails for any merge point, abort the algorithm.

6. Integrate the differing node in the clone instance in the selection mechanism of the clone abstraction. For example, with statement level differences, the differing statement in the clone instance is integrated as an extra case to the switch statement with a fresh case identifier. This is just one example of how the integration could take place when statement level differences are handled using a switch statement and there are other ways in which statement level differences could be handled which might require clever integration mechanisms.

7. Add an entry in the map $\mathbf{Map}_{param}$ for the merge point. We will look at some actual addition of entries into the map for merges in the following sections where we walk through our algorithm.

In order to explain **CloneMergeExt** and how the above mentioned verification integration and update to Map_{param} works, we use the example clone group merge in Figure 3.1. Let us assume that a developer has introduced a new clone instance **fn5** of the abstracted method **fnMerged**:

```
1 void fn5(){
2         int x = 40;
3         int y = x * 400;
4         char z = fn(x, y);
5         fne(z);
6 }
```

We walk through the steps that **CloneMergeExt** will perform when merging the method **fn5** with **fnMerged**:

1- Create an empty map

CloneMergeExt begins by creating an empty map of parameter of the merged function (or global variables or templates) to values, Map_{param}. This map aids in generating the function body that will replace the body of the existing clone instance.

2- Detect the merge points between the clone instance and the abstracted method

CloneMergeExt identifies the difference between the body of the clone instance method and the body of the abstracted method. This is the same as detecting the merge points between 'n' functions, where 'n' here is **2**. This detection is the same as described in Section 2.3.3.1. The merge points between **fn5** and **fnMerged** are as follows:

1. The constant **40** in line 2 of **fn5** and the reference **constant** in line 4 of **fnMerged**
2. The constant **400** in line 3 of **fn5** and the reference **globalConst** in line 5 of **fnMerged**
3. The type **char** in line 4 of **fn5** and the type reference **T** in line 6 of **fnMerged**
4. The function call statement in line 5 of **fn5** and the switch statement in lines 7-19 of **fnMerged**

3- Verify if the differing node in the clone abstraction can serve as a *selection mechanism* the node in the clone instance

Our tool needs to first check if each of the differing nodes in the clone abstraction can serve as a selection mechanism for the corresponding differing node in the clone abstraction:

1. The constant **40** in line 2 of **fn5** can be merged into the reference **constant** (selection mechanism) in line 4 of **fnMerged**, as **constant** is of type int.
2. The constant **400** in line 3 of **fn5** can be merged into the reference **globalConst** (selection mechanism) in line 5 of **fnMerged**, as **globalConst** is of type int.
3. The function call statement in line 4 of **fn5** can be merged into the switch statement in lines 6-17 (selection mechanism) of **fnMerged**, as the function call statement can be introduced as an extra case inside the switch statement.
4. The type **char** in line 5 of **fn5** can be merged into the type reference **T** (selection mechanism) in line 18 of **fnMerged**, as **T** is a type specifier.

Since the verification does not fail for any of the merge points, we continue with the algorithm.

4- Integrate the differing nodes in the clone instance into the corresponding selection mechanisms in the merged method

As we have observed earlier, type-3 clones require modifying the existing switch statement/conditional in addition to identifying the right values for existing parameters to the merged method. Our tool analyzes the type of the constant being used to switch the individual cases. We generate a fresh value of that type that does not conflict with the existing case values, create a new case inside the switch

statement or an else-if clause in case of a conditional. Our tool inserts the differing statement into the freshly introduced case and adds this entry to the mapping. At this point, the freshly introduced switch statement for our example would look like this:

```
switch(functionId)
{
  case 1:
    fna();
    break;
  case 2:
    fnb();
    break;
  case 3:
    fnc();
    break;
  case 'randInt':
    fnd();
    break;
}
```

Here, **randInt** is a random integer that is not equal to any of the existing case branches. An optimization for integer case identifiers, in those scenarios where the integers are contiguous, is to generate the next integer in the contiguous sequence.

5- Create a mapping of instantiating arguments to the existing parameters

CloneMergeExt, at this point creates a mapping of how each of the parameters or global variables are linked to the differences in the clone instance. This mapping is used to generate the wrapper function that will replace the function body of the existing clone instance. In the example above the mapping will look like this:

1. **constant** (first parameter to the merged method) → **40**
2. **globalConst** (globalConst) → **400**
3. **T** (first template parameter to the merged method) → **char**

4. **functionId** (second parameter to the merged method) → **randInt** excluding the numbers 1,2,3

6- Generate abstracted function call for the clone instance

CloneMergeExt uses the mapping to generate a call for the clone instance to the merged method. This step generates actual selectors for the clone instance corresponding to the appropriate selection mechanisms in the merged method. In our example, the body of the clone instance would look like this:

```
globalConst = 400;
fn5<char>(40, randInt);
```

Although our example is simplified, it illustrates our algorithm's workings. We use the following sections to run our algorithm through real examples and also summarize results of open source validations of our algorithm.

We provide an outline of our algorithm in Figure 3.2.

Input: M = Merged method, CN = Clone instance method

Procedure MergeCloneIntoAbstraction(M, CN)
  D_{list} ← GetDifferences(M, CN)
    The returned D_{list} is a list of pairs, where each pair
    corresponds to a merge point and contains a node in the
    merged method and the corresponding node in the clone instance.
  for each $D_m, D_{cn} \in D_{list}$
    D_m is the node in the merged method and
    D_{cn} is the corresponding node in the clone instance.
    if D_m can serve as a *selection mechanism* to D_{cn}
      Map D_m to D_{cn} in the map of selection mechanisms to actual selectors.
      Perform fix-up if needed
    else
      FAIL
    end if
  end for
  If not FAIL
    Generate abstracted function call

FIGURE 3.2: Outline of the algorithm to merge clone instance into an existing abstraction

3.3 Use Cases - Algorithm walk through

In this section, we apply our algorithm in code inspired by real merges of clone groups from our earlier evaluation in Section 2.6. Our tool is attempting to merge a clone instance into an existing merged method(abstraction) for each of the use cases. For each use case, there is an input to the tool which is one merged method followed by one clone instance and an output of the tool which has the generated merged method and a merged version of the clone instance. There is an arrow flowing from the input to the output in each of the cases. We generate the input for each of the examples by starting with a merged method and removing one of the clone instances from it and then attempting to merge the clone instance back into the merged method using **CloneMergeExt**. We provide a walk through of the steps of the above mentioned algorithm for each of the use cases.

3.3.1 Cocos 3d

In **Figure 3.3**, we are attempting to merge the clone instance **OnTouchesZoomout** with the abstraction **OnTouchesCommon**.

Differences

The difference between the two methods are:

- **_bzoomout** in **onTouchesZoomOut** and **touchProperty** in **onTouchesCommon**

Instance check

touchProperty is a pointer to a boolean which is a parameter to the abstracted function. Both **_bzoomout** and **touchProperty** are booleans, so our code passes this check.

```
//Input

bool Camera3DTestDemo::onTouchesCommon
(Touch* touch, Event* event, bool* touchProperty)
{
  auto target =
  static_cast<Label*>(event->getCurrentTarget());

  Vec2 locationInNode =
    target->convertToNodeSpace(touch->getLocation());
  Size s = target->getContentSize();
  Rect rect = Rect(0, 0, s.width, s.height);

  if (rect.containsPoint(locationInNode))
  {
      *touchProperty = true;
      return true;
  }
  return false;
}


bool Camera3DTestDemo::onTouchesZoomout
(Touch* touch, Event* event)
{
  auto target =
  static_cast<Label*>(event->getCurrentTarget());

  Vec2 locationInNode =
   target->convertToNodeSpace(touch->getLocation());
  Size s = target->getContentSize();
  Rect rect = Rect(0, 0, s.width, s.height);

  if (rect.containsPoint(locationInNode))
  {
    _bzoomout = true;
    return true;
  }
  return false;
}
```

```
//Output

bool Camera3DTestDemo::onTouchesCommon
(Touch* touch, Event* event, bool* touchProperty)
{
  auto target =
  static_cast<Label*>(event->getCurrentTarget());

  Vec2 locationInNode =
    target->convertToNodeSpace(touch->getLocation());
  Size s = target->getContentSize();
  Rect rect = Rect(0, 0, s.width, s.height);

  if (rect.containsPoint(locationInNode))
  {
      *touchProperty = true;
      return true;
  }
  return false;
}

bool Camera3DTestDemo::onTouchesZoomOut
(Touch* touch, Event* event)
{
  return
  Camera3DTestDemo::onTouchesCommon
  (touch, event, &_bzoomout);
}
```

FIGURE 3.3: Cocos 3d - Abstraction

Creation of abstracted call

Our tool will map **_bzoomout** to the third argument of the abstracted function, and pass its address as the third parameter.

3.3.2 Facebook - ROCKSDB

In **Figure 3.4**, we are attempting to merge the instance **function_open_JLjava** with the abstraction **function_open_ROnly__JLjava**. Although **function_open_ROnly__JLjava** contains only one case in the switch statement, implying that it is an unlikely merge of one instance or many instances that do not have a statement at this position, we use this example to illustrate a walk through of our algorithm in real code. We generated this example by beginning with a merged method which had a switch statement with two cases and removing one of the cases belonging to one of the clone instances.

Differences

- The statement in line 8 and the switch statements in lines 20-25

Instance check

The controller of the switch statement, **openType** is a parameter to the abstracted function. The statement can be made a case within the switch statement, so our code passes this check.

Creation of abstracted call

Our tool will create a new case for a new random integer different from **1** for the switch statement, add the appropriate case to the abstracted method, and pass the random integer as the parameter corresponding to **openType**.

```
1  //Input
2
3  //The dots indicate common code that
4  // are abbreviated for readability
5
6
7  jobject
8      function_open_ROnly__JLjava(
9      JNIEnv* env, jobject jdb,..., int openType) {
10   rocksdb::DB* db = nullptr;
11   rocksdb::Status s;
12   /* About 50 lines of common code */
13   switch(openType) {
14           case 1:
15           s = rocksdb::DB::OpenForReadOnly
16           (opt, db_path,column_families, &handles,&db);
17           break;
18           }
19   return null;
20 }
21
22 jobject
23 function_open_JLjava(
24 JNIEnv* env, jobject jdb,...) {
25           rocksdb::DB* db = nullptr;
26           rocksdb::Status s;
27           /* About 50 lines of common code */
28           s = rocksdb::DB::Open(opt,
29           db_path,column_families, &handles,&db);
30           return null;
31 }
```

```
//Output

jobject
    function_open_ROnly__JLjava(
    JNIEnv* env, jobject jdb,..., int openType) {
  rocksdb::DB* db = nullptr;
  rocksdb::Status s;
  /* About 50 lines of common code */
  switch(openType) {
        case 1:
        s = rocksdb::DB::OpenForReadOnly(opt,
        db path,column_families, &handles,&db);
        break;
        case 2:
        s = rocksdb::DB::Open(opt,
        db path,column_families, &handles,&db);
        break;
        }
  return null;
}

jobject
    function_open_JLjava(
    JNIEnv* env, jobject jdb,...) {
  function_open_ROnly__JLjava(
    JNIEnv* env, jobject jdb,...,2);
}
```

FIGURE 3.4: Facebook RockSDB - Abstraction

3.3.3 RethinkDB

In **Figure 3.5**, we are attempting to merge the instance **CreateIntArray** with the abstraction **CreateNumArray**.

Differences

- **int *** in line 19 and **T** in line 3

Instance check

T is a template parameter to the abstracted function. Both **int *** and **T** are type specifiers, so our code passes this check.

Creation of abstracted call

Our tool will map **int *** to the first template parameter of the abstracted function, and pass it as the first template parameter.

3.4 Validation

We envision that **CloneMergeExt** will be used in a code base that has an existing clone abstraction and after a new function has been introduced that could serve as a clone instance to that existing abstraction. Since **CloneMergeExt** is used at a point in software development lifecycle where a new method has been introduced, we need to validate if detecting an appropriate abstraction (if exists) for the newly method is performed in a reasonable time and if the merges that follow the detections are accurate. For this, we use the following setup. We began with code bases of accepted pull requests from our evaluation in Section 2.6 and gathered the clone groups that were part of those accepted pull requests. In order to validate our approach, we answer the following questions:

```
1  //Input
2
3  template<typename T>
4  cJSON *cJSON_CreateNumArray(T numbers,int count) {
5      cJSON *n=0,*p=0,*a=cJSON_CreateArray();
6      for (int i=0;a && i<count;i++) {
7          n=cJSON_CreateNumber(numbers[i]);
8          if(!i) {
9              a->head=n;
10         } else {
11             suffix_object(p,n);
12         }
13         p=n;
14     }
15     a->tail = p;
16
17     return a;
18 }
19
20 cJSON *cJSON_CreateIntArray(int *numbers,int count) {
21     cJSON *n=0,*p=0,*a=cJSON_CreateArray();
22     for (int i=0;a && i<count;i++) {
23         n=cJSON_CreateNumber(numbers[i]);
24         if(!i) {
25             a->head=n;
26         } else {
27             suffix_object(p,n);
28         }
29         p=n;
30     }
31     a->tail = p;
32
33     return a;
34 }
```

```
//Output

template<typename T>
cJSON *cJSON_CreateNumArray(T numbers,int count) {
    cJSON *n=0,*p=0,*a=cJSON_CreateArray();
    for (int i=0;a && i<count;i++) {
        n=cJSON_CreateNumber(numbers[i]);
        if(!i) {
            a->head=n;
        } else {
            suffix_object(p,n);
        }
        p=n;
    }
    a->tail = p;

    return a;
}

cJSON *cJSON_CreateIntArray(int *numbers,int count) {
    return cJSON_CreateNumArray<int*>(numbers,count);
}
```

FIGURE 3.5: RethinkDB - Abstraction

- Are the merges produced by our approach correct?

  Since we are attempting to merge clone instances to clone abstractions, we had to create partial clone merges. For each of the above mentioned clone groups, we began by gathering every possible subset of clones in the clone group and merging the remaining clones using **CloneMerge**. We attempted to merge every method in the subset to the merged method incrementally using **CloneMergeExt**. Except for the case numbers randomly generated by our tool for switch statements and the case ordering, our merges were exactly the same as the existing merges.

- Are we able to identify the method to which a clone instance should be merged in reasonable time?

  We used RTED to detect the differences and defined a method as an appropriate clone group merge for a clone instance if the edit list did not contain any insertions or removals (only relabels) and if the number of merge points did not exceed 30% of the number of nodes in the clone instance. We did not encounter scenarios where there were multiple suggestions for a single chosen clone instance. The individual repositories, with their number of clone groups, the number of methods in the code base and the average time taken for the process of detecting the appropriate clone group for each possible clone instance are listed in Figure 3.6.

  Our approach is currently limited in the sense that the user has to manually change between **CloneMerge** and **CloneMergeExt** depending on his use case since the use cases of the two different tools happens during different situations in the software development life-cycle.

Repository	Clone groups	Number of methods	T (in seconds)
Cocos 2d	2	5459	44.09
Oracle Nodedb	3	155	2.25
RethinkDB	2	1948	23.09
Google Protobuf	1	2682	29.00
MongoDB	2	15609	143.20

FIGURE 3.6: Validation of merging clone instance with merged methods

T - average time taken for the process of detecting the appropriate clone group for each possible clone instance are listed

Part II

Source code Rewriting

Chapter 4

Source code Rewriting - Aiding program transformation using the program dependence graph

Chapter Overview

In this chapter, we discuss the language we have designed to support data representation migration. We begin by summarizing the challenges related to data representation migration that we discussed in the Section 1.2.3. Data representation migration is the task of transforming a program source code that uses an existing data type into a program that uses a new data type. The two main challenges associated with data representation migration are the ability to express customizable transformation rules and the ability to selectively apply transformation rules based on program dependence. Transformation languages provide the ability to perform code transformations based on customizable rules and refactoring systems provide propagation of transformations based on program dependence, but we need a system that can do both. Based on this insight, we built a language and a transformation engine and evaluate it. Our language provides four main features in accordance with our analysis:

- *search* across dependencies of transformed terms to identify terms requiring transformation,
- *scoped rules* that help direct the application of rules that only apply to particular contexts, such as loop bodies or structure definitions,
- *tags* that enable propagating information gathered from simple forms of program analysis on AST fragments, and
- *User guidance* including the choice of when to roll back a series of transformation steps.

Our tool is implemented within the Eclipse framework [22], and relies on the CDT C/C++ program manipulation plugin.

The main contributions of our work are as follows:

- We study a motivating example from a real problem of migrating from scalar to vector representations at the source level.
- Based on the example, we analyze and classify the kinds of code transformations required to perform data representation migrations.
- We propose a transformation language addressing these requirements, and informally describe its semantics.
- We evaluate our transformation system on existing code and quantify the resulting improvements.
- We also evaluate the running time of the tool on a non-trivial program.

The rest of this chapter is organized as follows. Section 4.1 presents a motivating example, in terms of code fragments that illustrate the migration to be performed. Section 4.2 presents the main concepts of our transformation language. Section 4.3 gives an overview of the language semantics. Section 4.4 evaluates our approach on several case studies. Section 4.5 presents the results of our implementation. Section 4.6 discusses the scalability of our approach on a non-trivial transformation.

4.1 Motivating Example

We begin with the example of vectorization to illustrate the challenges in performing data representation migration. The Vc library provides vector versions of primitive types and corresponding operations to allow developers to write portable versions of vectorized code [12]. Figure 4.1 shows a program that converts an array of Cartesian coordinates, represented using floats, into polar coordinates (left) and a program that does the same, but on an array of float vectors (right). In the latter, the vector type float_v represents a machine vector of float variables; its size is hardware-dependent. As we see in line 21 on the right hand side code, Vc can vectorize not only arithmetic operations and updates, but also comparisons and conditional updates, using C++ function call syntax. This example is inspired by a transformation tutorial found on the Vc website [Kretz].

The main challenges in vectorizing this code are as follows:

- The developer must identify and transform syntactic patterns, such as the transformation from the call to **std::sqrt** to the call to **vc::sqrt** .
- The Vc documentation describes migration from an array of structures with fields to an array of structures with the same fields, vectorized. However, the vectorization scheme is only effective when all fields have types that have the same in-memory sizes. For simplicity, we require that all types be the same to ensure that structures can be vectorized. The developer could thus start by analyzing the definition of struct CartesianCoordinate, which is the type of **input**, and observe that it contains only floats. The developer can then create a new vectorized type CartesianCoordinate_v, rename **input** to **input_v**, and update CartesianCoordinate_v's float fields to use the float_v type.
- Vectorizing the elements of **input** changes the array size. The developer must thus collect information about the type of the structure fields, float in our example, and use the size of its Vc counterpart, float_v, to compute the

```
1  struct CartesianCoordinate {
2      float x;
3      float y;
4  };
5  CartesianCoordinate input[1000];
6
7  struct PolarCoordinate {
8      float r;
9      float phi;
10 };
11 PolarCoordinate output[1000];
12
13 void fn() {
14     for (int i = 0; i < 1000; ++i) {
15         float temp1 = input[i].x;
16         float temp2 = input[i].y;
17         output[i].r = std::sqrt((x * x) + (y * y));
18         output[i].phi =
19             std::atan2(y, x) * 57.29578018188476f;
20
21         if (output[i].phi < 0.f) {
22             output[i].phi += 360.f;
23         }   }  }
```

```
1  struct CartesianCoordinate_v {
2      float_v xv;
3      float_v yv;
4  };
5  CartesianCoordinate_v input_v[1000 / float_v::size];
6
7  struct PolarCoordinate_v {
8      float_v rv;
9      float_v phiv;
10 };
11 PolarCoordinate_v output_v[1000 / float_v::size];
12
13 void fn() {
14     for (int i = 0; i < 1000 / float_v::size; ++i) {
15         float_v temp1 = input_v[i].xv;
16         float_v temp2 = input_v[i].yv;
17         output_v[i].rv = vc::sqrt((x * x) + (y * y));
18         output_v[i].phiv =
19             vc::atan2(y, x) * 57.29578018188476f;
20         // vectorized conditional update:
21         output_v[i].phiv(output_v[i].phiv < 0.f)
22             += 360.f;
23     }   }
```

FIGURE 4.1: Vectorization using the Vc library

new size. A similar change is needed on the limit of the **for** loop, based on the type of the array element expressions inside the loop.

- The developer must transform all the uses of **input** to **input_v**. The developer must also transform all the uses of the transformed **input** using program dependency. In our case, we must in particular vectorize **output**, which would require performing steps similar to the vectorization of **input**.

To automate these steps, we need a tool with the following abilities:

1. Propagating the need for transformation across variable def-use dependencies. For example, transforming **CartesianCoordinate input[1000];** must trigger processing of all uses of **input**.
2. Transforming and analyzing code sub-fragments. For example, the transformation must consistently apply the float-to-float_v transformation to all declarations in the bodies of relevant type definitions.
3. Propagating information from one transformation step to another. For example, changing the size of **output** requires knowing that the fields of CartesianCoordinate_v have type float_v, as the new size depends on the size of the float vector in the current hardware implementation.
4. Rolling back the transformation, if the transformation process detects inconsistencies.

4.2 Language

We now present an overview of our transformation language and highlight how it meets the needs identified in Section 4.1. We present its semantics in Section 4.3. Specifications in our language are written in the form of pattern matching and transformation rules.

4.2.1 Rules

A transformation specification in our language consists of a sequence of *top-level* rules which can in turn contain invocations of *scoped* rules.

4.2.1.1 Top-level rules

A *top-level* rule has the form

Category *pattern* ==> *transformation*
[**notransform** *term_list*]
[**where** *condition*]

Category is a syntactic category, such as **expression**, **statement**, **declaration**, or **decldefinition**. The **decldefinition** category, illustrated in Section 4.2.1.3, indicates a pattern that matches a variable declaration and the definition of its type at once. *Pattern* is a pattern that has the form of a term in the specified syntactic category and may contain *metavariables*, which match arbitrary subterms. The name of a metavariable begins and ends with `$`. *Transformation* is another pattern, which describes the generated code. To transform code, we match the code against *pattern*, bind all metavariables, and substitute them in *transformation*. In our language, application of a transformation rule triggers the processing of terms that somehow depend on the transformed term. The optional **notransform** clause contains subterms on which such triggering should not occur. The optional **where** clause puts some constraints on the possible values of the metavariables. Constraints currently relate to the type of an expression-typed metavariable, where the type is obtained using the C++-like operator **decltype** [cppreference.com], and equality checks on the structure of the code bound to the metavariables. We envision that this list of supported **where** clauses can be extended.

The GMP library [GNU] from GNU provides an API for performing arbitrary precision arithmetic. As an example of a transformation specification, the following

top-level rules transform an integer multiplication to its big integer counterpart in the GMP library:

```
expression $a$ = $b$ * $c$ ==> mpz_mul_si ($a$, $b$, $c$)
notransform: $c$
where decltype($c$) == int

expression $a$ = $b$ * $c$ ==> mpz_mul_ui ($a$, $b$, $c$)
notransform: $c$
where decltype($c$) == unsigned int
```

In the first rule, the category is **expression**, the pattern is **a = b * c** and the transformation is **mpz_mul_si(a, b, c)**; the second rule is structured similarly. These rules transform a multiplication and assignment into the bignum multiplication functions *mpz_mul_si*, for signed integers, and *mpz_mul_ui*, for unsigned integers. Each generated function call has three arguments. The first two are of type **mpz_t** and the last one is an integer, i.e., an int or unsigned int, as it is in the original code. Both rules thus use **notransform** to indicate that this third argument should not be scheduled for further transformation. We use **notransform** to exploit optimization opportunities for special purpose rules that exploit special purpose scenarios, as seen in the multiplication examples we just discussed.

The application of top-level rules follows a dependency-based strategy. Specifically, top-level rules trigger on code if either the user selected that code, or if a previously triggered rule depends on that code.

4.2.1.2 Scoped rules

The dependency-based application strategy for top-level rules is effective for following the program's data flow, but in some cases we must transform a region of code exhaustively. For example, in our Vc example, transforming a structure involves transforming int and float declarations into int_v and float_v declarations, exhaustively within the body of a type definition. We support exhaustive transformation of a region of code with *scoped rules*, which are sets of transformation

rules that are limited to a particular region of code and applied to every syntactic match in the region.

A scoped rule declaration has the following structure:

scope *name*[(*parameter_list*)]{{
  (**tag** *typename*;)∗
  transformation_rule+
}}

The optional *parameter_list* allows us to pass parameters to the scoped rule (Section 4.2.1.4). A simple example of a scoped rule is:

```
scope vc_decl_scope {{
  declaration   int $a$; ==> int_v $a$v;
  declaration   float $a$; ==> float_v $a$v;
}}
```

This rule replaces all occurrences of an `int` declaration in the given scope by an int_v declaration and renames the declared variable or field by appending a '**v**'; it transforms float declarations analogously. In our motivating example in Figure 4.1, we would use this scoped rule to change the types and names of all fields in a specific structure.

A scoped rule does not explicitly specify the scope to which it applies. Instead, top-level rules trigger scoped rules explicitly, within their *transformation* specification. The syntax mirrors that of a function call, as illustrated below:

```
declaration struct $s$ { $body$ } ==>
  struct new_$s$ { vc_decl_scope($body$) }
```

In this case, the transformed term is constructed by instantiating the metavariable `$s$` as indicated by the *pattern*, and by replacing the scoped rule invocation, **vc_decl_scope($body$)**, by the result of applying the rule `vc_decl_scope` exhaustively within the term that has been bound according to *pattern* to the metavariable `$body$`.

As an example of the use of the above rule, consider the structure declaration in Figure 4.2. Applying the above top-level rule binds the metavariable `$s$` to

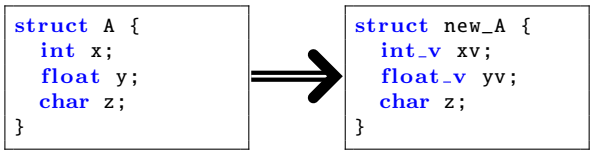


FIGURE 4.2: Struct definition before and after applying scope

the structure name `A` and the metavariable `$body$` to the sequence of field declarations, `int x;` etc. The transformation component of the rule indicates that the resulting structure should be named `new_A`, as shown in Figure 4.2, and the declared fields should be the result of applying the scoped rule `vc_decl_scope` anywhere it matches within the field declaration list, specifically, to the int and float fields.

4.2.1.3 Tags

We further allow reasoning over the code to ensure uniformity constraints through scoped rule *tags*. We observed previously that vectorization works best when all of the fields of the transformed structure have the same type. As illustrated by Figure 4.2, the scoped rule `vc_decl_scope` currently does not ensure this property. To address this issue, we add a *tag* to the scoped rule. A tag serves as local storage for the processing of a scope. A tag is declared as **tag** *tagname* [=initial value];. If no initial value is supplied, then the tag is initialized to $\perp$. It can be updated once in the processing of a scope, and then subsequent attempts to modify it must provide the same value, effectively implementing universal ($\forall$) quantification of the value over the terms in the scope. Tags are updated using an **updatetag** clause that can have either of the following forms:

updatetag *tagname* := *value*
updatetag *tagname* := ((*value* : *condition*,)+)

An **updatetag** clause follows individual rules inside a scoped rule. An **updatetag** clause can update a tag with a specific value, or with one of a set of values, whichever satisfies a corresponding condition. The set of conditions do not have

to be exhaustive. A default value can be provided by setting the condition to **otherwise**.

The following example extends the scoped rule from Section 4.2.1.2 to include a tag:

```
scope vc_struct {{
  tag vctype;
  declaration int $a$; ==> int_v $a$v;
              {{updatetag vctype:=int_v}}
  declaration float $a$; ==> float_v $a$v;
              {{updatetag vctype:=float_v}}
  declaration $T$ $a$; ==> $T$ $a$;
              {{updatetag vctype:=top}}
}}
```

This scoped rule provides the same transformation rules as `vc_decl_scope` above, but on each transformation it also updates the tag `vctype` with the chosen type. Application of the scoped rule only succeeds if all the updates to the tag have the same value; otherwise, the entire transformation process is aborted, reverting the code in its original state. In this example, the first two rules update the tag to the chosen type, which is then used in the transformed code, while the third rule actually performs no transformation, but only gives the tag the value $\top$ (top) to indicate that a type other than int or float has been detected and the scoped rule application should fail. Since we prioritize earlier rules over later rules, this third rule does not match if `$T$` is int or float. The tag thus ensures that the scope does not contain a mixture of ints and floats, or of ints or floats and other types. In particular, our example structure declaration in Figure 4.2 would incur a failure and a rollback of the whole transformation process.

Tag values can also be accessed by the rule that invokes the scoped rule using the form *scopedrule.tagname*, allowing the former rule to obtain information about the code processed in the scope. In the rule in Figure 4.3, our language uses the tag value obtained by processing the type definition in specifying the new size of the array-typed variable. If this functionality is used, the scoped rule can be used only once in the given transformation.

```
decldefinition
  struct $structname$ {$body$} $obj_name$[$s$] ==>
  struct $structname$v {vc_struct($body$)}
  $obj_name$v [$s$/vc_struct.vctype::size]
```

FIGURE 4.3: Example usage of tags

In the context of the CartesianCoordinate structure defined on the left side of Figure 4.1, the scoped rule **vc_struct** will update the tag **vctype** with float_v. Here, **size** is an attribute of the vector types in the Vc API, such as float_v and int_v. Then, applying the rule in Figure 4.3 on the **decldefinition** comprising the type definition of the struct CartesianCoordinate and the declaration **struct CartesianCoordinate input[1000]** will yield a new type, CartesianCoordinate_v, along with its definition, and a new object of that type, **input_v**, as shown on the right side of Figure 4.1.

The rule in Figure 4.3 illustrates the use of the syntactic category **decldefinition**, which is useful when the transformation of a variable declaration, in our case the array declaration, requires information based on analysis of the definition of the type, in our case the structure definition.

4.2.1.4 Scoped rules with parameters

In addition to all the above-mentioned features, scoped rules also support parameter lists. The values to parameters are passed through formal parameters, which follow syntax similar to function definitions in C-like languages. When a top-level rule calls a scoped rule, the top-level rule must supply corresponding actual parameters. The scoped rule can in turn use them to perform checks on the terms processed in the scope or use it to perform transformations based on this value. As an example, consider the requirement that all of the array indexed elements inside a **for** loop must be indexed by the index of the **for** loop. We can express such a rule using the following scoped rule. A detailed specification of this loop for the general loop scenario is described in Section 4.4.

```
scope vc_for($index$) {{
  tag vctype;
```

```
  expression $a$[$i$] ==> $a$[$i$]
      where $i$ == $index$ //failure of the where fails the application of the rule
   //vctype is a tag that contains the type of the array indexed references
   // in the loop. code to populate vctype has been omitted for brevity
}}
```

An example use of such a parametrized scoped rule is as follows:

```
statement
for( int $i$ = 0; $i$ < $limit$; $i$++){$body$}
==>
for( int  $i$ = 0; $i$ < $limit$/vc_for.vctype::size; $i$++)
  { vc_for($body$, $i$) }
```

In a scoped rule invocation, the first argument represents the subterm to which the scoped rule should apply, and the formal parameters, if any, are bound to the remaining arguments.

4.2.2 Assessment

Comparing the features of our language to the requirements that we outlined in Section 4.1, we find that our notion of propagating change across dependencies (which we detail in the next section) supports requirement (1). Scoped rules support requirement (2), while tags support requirement (3), allowing us to pass information from scoped to top-level rules. We have furthermore noted a case where the transformation language's runtime system provides rollback, thus addressing requirement (4), in the context of scoped rules.

4.3 Semantics

The semantics of our transformation language specifies *where* transformations are performed, *i.e., which* terms are selected for transformation, and *how* the transformation of the selected terms is carried out. The latter is straightforward and typical of rule based approaches in which rules are expressed using concrete syntax:

we match a pattern against source code, instantiate the transformation specification according to the match information, and finally schedule the original code to be replaced by the result. Our contribution lies in the choice of where to perform transformation, to meet the needs of data migration. For this we rely on two key notions: propagation over dependencies and user interaction. We start with our program model that supports these features.

4.3.1 Program model

The goal of our language is to change how code represents data. Changing a data representation requires changing types or even type definitions, which naturally affects all of the operations that process that data. Changing these operations may in turn affect their other inputs: for example, vectorizing a variable may trigger the vectorization of a binary operator that uses that variable, which in turn may require vectorizing the other argument as well. The change may also affect the representation of the operation's result, thus requiring changes in other operations that depend on this information. Accordingly, reasoning about dependencies between program elements is a key part of the semantics of our language.

We represent the source code as an abstract syntax tree (AST), in which nodes are annotated with *dependency links*. A dependency link is a symmetric connection between two nodes, such that if the system transforms one of the nodes, the other must also be considered for transformation. The exact positioning of the dependency links depends on the source language. Our implementation for C and C++ programs supports the following dependency links:

1. Between a variable's uses and its declaration and definitions.
2. Between a type reference and its definition.
3. Between a function and its callers.
    - Between a function's return statements and the call sites.
    - Between a function's actual and formal parameters.

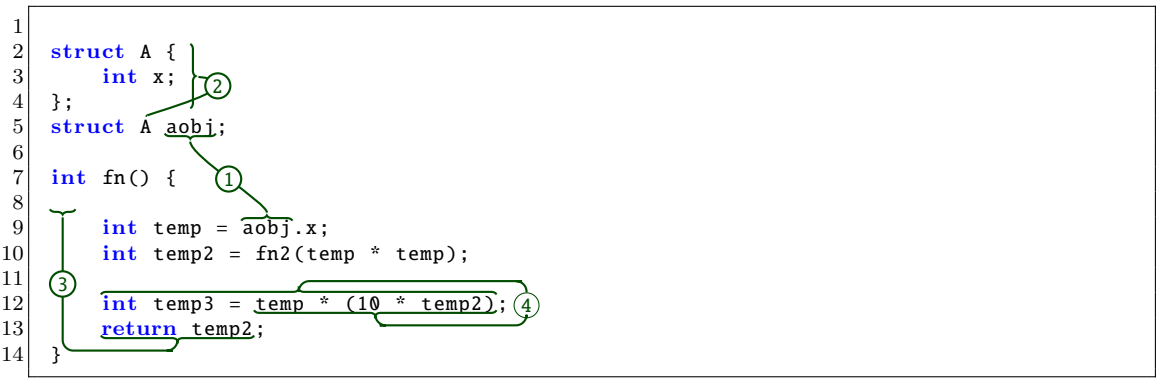


FIGURE 4.4: Dependency links

4. Between a node and its immediate parent.

The first three types of links are also found in Program Dependence Graphs [23]. The fourth type is derived from the fact that a change in a term can affect the side effect or result of that term, and thus may require changing any containing term that observes that effect or result, and inversely a change in an operation may require changing the computations whose results are manipulated by that operation. Our approach currently does not take aliases into account, but we envision that it could be extended with an existing alias analysis for C or C++ code [24].

Figure 4.4 illustrates all four types of links: Link type (1) occurs between the declaration of `aobj` in line 5 and its use in line 8, while type (2) occurs between the definition of struct `A` and its use in line 5. Type (3) connects the function's return statement with its call sites, and type (4) connects the expression `temp * (10 * temp2)` with the declaration and initialization of `temp3`.

4.3.2 Algorithm

The algorithm, outlined in Figure 4.5, begins with a node, *SN*, selected by the user as the starting point of the transformation process. The algorithm is iterative and is guided by two data structures, WORKLIST, containing all the nodes of the original AST that need to be processed, and DONE, containing the nodes of the

original AST that have already been processed. WorkList initially contains only *SN*. Each iteration takes the first AST node off of the work list and searches for a rule whose pattern matches the term rooted at that AST node. If a rule is found, the algorithm performs the transformation described by the rule, updates the AST node with a record of the chosen transformation to be performed at the end of the rule application process, and updates the end of the work list with the dependencies of the original term rooted at the AST node. Finally, whether or not a rule is applied, the algorithm adds the AST node to DONE. Note that the algorithm does all rule matching before any transformation. Concretely, it collects the transformations that need to be performed on each node and after all of the nodes have been processed, *i.e.*, the work list is empty, it performs all the transformations at once. In case of conflicting transformations where multiple transformations are staged on the same tree, the transformations that were staged later overwrites the previously staged transformations. Our implementation logs the situations where such overwrites have taken place for the user to have a look at. The rest of this section describes the key procedures of the algorithm.

4.3.2.1 Finding a node at which to apply a rule

The function FindRule(*SN*) searches top to bottom through the rule list to find a rule whose pattern matches an AST that contains the node *SN*. Specifically, *SN* must be matched either to part of the concrete syntax of the rule pattern or by a metavariable. It cannot be a proper subtree of the AST matched to a metavariable. The motivation behind such a matching strategy is that the algorithm may need to search up AST ancestors from *SN* to find a node where a rule can be applied, but the search should be limited to subtrees that are dependent on *SN*. The result is a pair of the root of the matched AST, N, and the chosen rule, R, or '(*SN*, null)' if no rule is found.

As an example of the matching process, consider the AST `e(f(g(h(x))))`. In this AST, if the node under consideration *SN* is rooted at `f`, at the call to `f`, at `g`, at the call to `g`, or at `h(x)`, then the matching of the pattern `f(g($i$))` will

```
Input: R = Rule List, P = Program to transform, SN = Selected Node
Internal Variables: WorkList, DONE

Procedure Main
    WorkBlock(SN)
    while WorkList ≠ ∅
        N_d ← WorkList.popfirstelement
        WorkBlock(N_d)
    end while

Procedure WorkBlock(SN)
    (N,R) ← FindRule(SN)
    if R == null
        return
    N′ ← SUBST(N, R)
    DONE ← DONE ∪ {N, SN}
    if N′ ≠ FAIL
        RecordInAST(N, N′)
        D ← Get the dependency links of N
        for d ∈ D where d ∉ WorkList ∧ d ∉ DONE do
            WorkList ← WorkList.appendtoend(d)
        end for
    else if User choice is to roll back ∨ tag value is bottom
        Roll code back to original state
        Abort transformation
    else
        Apply one of the chosen tag values
    end if
```

FIGURE 4.5: Outline of the transformation algorithm

succeed. In all of these cases, FINDRULE returns the node at the call to `f`. The pattern does not match if *SN* is rooted at `x`, because in that case *SN* is merely a proper subterm of the term `h(x)` matched by `$i$`.

4.3.2.2 Substituting the Right-Hand-Side of a Rule

The function SUBST(N, R) performs the following operations on node N and rule R:

1. If a rule R that was applicable to node N was found, unify R's *pattern* with N and substitute all metavariables in R's *transformation* accordingly. The transformation may still contain *scope applications*, *tag updates*, and *tag references*.

2. Process all scope applications *scope*(`$s$`) left to right, in depth-first pre-order by applying a scoped rule *scope* to the node n_s as described in Section 4.3.2.3, where n_s is the AST node matched by `$s$`.

3. Replace each tag reference with the corresponding tag value. Tag references have the form *scope*.*t* for scoped rule *scope* and tag *t*. Each scoped rule with such a tag reference may be applied at most once per *transformation*, and thus the notation is unambiguous.

   If the value of any tag *scope*.*t* that occurs in the *transformation* is $\top$, *i.e.*, inconsistent or undefined, then ask the user whether to roll back the entire transformation. If the user chooses roll back, then abort and return FAIL. If the user chooses to continue with the transformation, and the tag value is $\top$, ask the user to choose one of the proposed tag values. If the tag value is $\bot$ or undefined, then the transformation has to be rolled back without user choice. The WorkBlock can fail only when there is a failure to find a tag value whenever there is a reference to a tag value.

4. The **RecordInAST** function appends the returned value to the end of list of the transformations to be performed.

4.3.2.3 Applying a Scoped Rule

A scoped rule *scope* is applied to the AST rooted at node n_s as follows. We refer to the nodes of this AST as the *scoped nodes*. The process of searching for nodes to which to apply the rules in a scoped rule is done in depth-first pre-order, and is thus quite different from the process for searching for nodes to which to apply a top-level rule, based on dependencies. Indeed, the process of applying a scoped rule is simply an extension of the dependency-triggered process of applying the top-level rule that invokes it, and thus it effects the complete processing of the scoped nodes.

Our algorithm first initializes any tags in the scoped rule to $\bot$. It then traverses all of the scoped nodes in depth-first pre-order and, for each node N, applies the

first rule within the scoped rule, if any, whose pattern matches the subtree rooted at N, according to the process described in Section 4.3.2.1. This process ignores the set DONE of previously transformed nodes, neither checking nor recording scoped nodes. However, for any scoped node that we transform, we add all nodes connected via dependency links that are not in DONE to the WORKLIST only if a top level rule applies.

The exhaustive strategy used in applying a scoped rule to the scoped nodes also increases the probability that the tag values reflect the properties of all matching nodes, thus implementing $\forall$ quantification. While updating a tag, if the current value is not $\bot$, a scoped rule ensures that the new value for the tag is equivalent to the current value. Otherwise, the tag is set to $\top$ to indicate disagreement.

By not consulting or extending DONE, the exhaustive strategy allows a given node to be transformed by any number of scoped rules in addition to at most one top-level rule. Whenever we observe multiple proposed transformations for an AST node, our system applies the transformation that was recorded last. However, the system records the discarded earlier transformations in a log file and displays them to the user.

We summarize the differences between rule application for the top-level rules and the scoped rules as follows:

- The rules in a scoped rule are applied to all the scoped nodes, using a depth-first pre-order traversal. Top-level rules are applied to nodes chosen in the order in which the WORKLIST was updated via dependencies.
- Scoped rules do not update DONE, whereas top-level rules always update DONE.
- Top-level rules do not trigger user interaction. Scoped rules trigger user interaction when there is a failure to arrive at a unique tag value. The user can choose to roll back or select one of the observed alternative options for the tag values.

4.3.2.4 Dependencies

We gather the dependencies of a node N by following its dependency links (Section 4.3.1), whenever those are part of subterms that are not excluded via **no-transform**. The order in which we process these links is determined first by their link category (as in Section 4.3.1), and second by the order in which they occur in the source code when following a pre-order traversal.

Automatic Renaming A transformation rule that renames a declared variable or a field declared as part of a type definition also implicitly renames that variable or field is consistently throughout the variable's scope. We perform such renaming after applying all top-level rules.

4.3.2.5 User Interactivity

Our approach is interactive and therefore contains 'choice points' where the user needs to interact with our system to aid the migration process. The user interacts with our system to make the following choices:

- The user chooses the starting point as a code segment.
- If a tag update fails when applying a scope rule to a set of nodes, our system presents the user with the choice of either rolling back the transformation or selecting from the list of values that were proposed for that tag.

4.3.2.6 Formal properties

In this section, we discuss the formal properties of our algorithm: termination, confluence, and determinism.

Termination. Our algorithm only processes nodes from the original AST, and processes each node by a top-level rule at most once. As the original AST is finite, the number of iterations of the while loop in the algorithm is finite. Each node contains a finite number of dependency links that can be added to the worklist and a node is never added to the worklist more than once. Each iteration processes at most a set of nodes from a scope and there is a finite number of scope calls per iteration; the number of such nodes is also bounded by the number of nodes in the original AST. Thus, termination is guaranteed.

Confluence. The result of our approach depends on the starting point chosen. For example, if the user transforms the term `a + b` with one rule replacing `a` by `1` and with another rule replacing `b` by `2`, the result will be `1 + b` if the user selects `a` as the starting point, and `a + 2` if the user selects `b`. Note that in both cases the dependency relations do not trigger transformation of the other argument because no transformation rule applies to the addition expression itself.

Determinism. Overall, the approach is deterministic, in how nodes are added to the worklist (depth-first pre-order traversal), how they are removed from the worklist (first-in-first-out), and how rules are selected. Nevertheless, the order in which nodes are added to the worklist, specifically the processing of the subterms, depends on the structure of the AST, which may not be known by the user. All of the matching for rule selection done by our approach depends only on the structure of the original program's dataflow graph, and thus it makes no difference to the rule matching whether a particular node is treated earlier or later in the transformation process. The order in which nodes are treated does, however affect which transformation is chosen when multiple transformations accumulate at a single node, and thus in this case the user may not automatically obtain the expected result. We record such conflicts and emit suitable warnings to allow the user to manually intervene later.

Complexity. Our algorithm in the worst case visits each node once, and for each node scope applications can process all the nodes rooted at the scoped node once. The worst case complexity of our algorithm is therefore, $O(n^2)$ where n is the number of nodes in the AST representing the program.

4.3.3 Limitations

Our algorithm does not perform transformations in-place; instead, it collects transformations at AST nodes, and applies them after all rule matching has taken place. This limitation implies that the application of one rule cannot be sensitive to the effect of applying another rule and that multiple transformations may accumulate for a single node, all but one of which will be thrown away. This limitation stems from the implementation framework that we use, Eclipse CDT, which does not allow in-place transformations. We log the scenarios where multiple transformations are staged on a single node.

Another limitation of our approach is expressiveness. Our language allows matching of a code fragment with metavariables and transforming it into another code fragment that makes use of the code fragments bound to the metavariables. Our approach depends on the existence of a one-to-one mapping from a code fragment in the existing data representation to a corresponding code fragment in the target data representation. For example, in order to express vectorizing only alternate elements of an array, our language would require special load and store functions from the Vc library that provide this functionality. Thus, expressing such migrations using our approach would require support from the library supporting the target data representation

4.4 Case Studies

We have evaluated our algorithm through three case studies. In each case study, we start with an input specification, an input program and a starting node and

```
// Block A:
// top-level rule to transform an array of structs
decldefinition struct $structname$ {$body$}[$s$] ==>
struct $structname$v { vc_struct ($body$) }[$s$/ vc_struct.vctype :: size ]

// Block A': scoped rule to transform the body of a struct
// definition and compute the tag vctype within the body
scope vc_struct {{
  tag vctype;
  declaration int $a$; ==> int_v $a$v;
    {{ updatetag vctype := int_v }}
  declaration float $a$; ==> float_v $a$v;
    {{ updatetag vctype := float_v }}
  declaration $T$ $a$; ==> $T$ $a$;
    {{ updatetag vctype := top }}
  // Rules for other primitive types is ommitted for brevity
}}

// Block B: top-level rule to perform
// vectorization on for statements
statement for(int $i$ = 0; $i$ < $limit$; $i$++){$body$}
==> for(int $i$ = 0;
        $i$ < $limit$/vc_for.vctype :: size; $i$++)
    { vc_for($body$, $i$) }


// Block B': scoped rule to transform the body of a for
// statement and compute the tag vctype on the loop body
scope vc_for ( $ind$ ) {{
  tag vctype ;
  tag loopindex =  $ind$;
  expression $a$[$i$ ]. $b$ ==> $a$[$i$ ]. $b$
    {{ updatetag vctype :=
       ( int v : decltype ($a$[$i$ ]. $b$) == int ,
         float v : decltype ($a$[$i$ ]. $b$) == float ,
         top : otherwise );
       updatetag loopindex := $i$}}

  expression std :: sqrt($expr$) ==> vc :: sqrt($expr$)
  expression std :: atan2($expr$) ==> vc :: atan2($expr$)
  declaration int $a$;  ==> int_v $a$v;
    {{ updatetag vctype := int_v }}
  declaration float $a$; ==> float_v $a$v;
    {{ updatetag vctype := float_v }}
  // Rule CE
  statement  if ($c$) { $x$ += $e$; } ==> $x$($c$) += $e$;
}}

// Guidance rule
expression $a$[$i$] ==> $a$[$i$]
```

FIGURE 4.6: Case Study Vc Specification

apply our tool **DMF** (Data migration framework), based on our algorithm, over it. We run through each step of the algorithm and give a glimpse of the evolution of the work list and the rules being applied.

4.4.1 Vectorization

Our first case study is based on a tutorial presented on the Vc website [Kretz] that details the conversion from a scalar program to a vectorized program. As previously discussed, the Vc API allows explicit vectorization of C++ source code. We started by writing a specification that describes how to transform scalar code into Vc-style vector code (**Figure 4.6**). The specification has three main top level rules:

- The rule in **Block A** transforms an array of structures that contain either int or float fields into an array of structures of corresponding vectorized int_v or float_v fields. The constraints on the fields are checked by the scoped rule **vc_struct** in **Block A'**.
- The rule in **Block B** performs vectorization on **for** statements, reducing their number of iterations according to the size of the machine vector registers. The scoped rule **vc_for** in **Block B'** performs most of the work of this transformation.
- The guidance rule, at the end of the specification, serves to trigger the propagation of the array index variable to the work list, the need for which will be illustrated in the algorithm walk through below.

The above rules assume that the size of the array is cleanly divisible by **float_v::size** or **int_v::size**. To make our rules more general, we can force them to compute the array size rounding up; we omitted this step for readability.

We use the specification shown in **Figure 4.6** to transform the code shown in **Figure 4.1**. As a selected node, we choose the declaration of the **input** array, **CartesianCoordinate input[1000]**, and (implicitly) the corresponding type definition struct CartesianCoordinate. The rule in **Block A** in **Figure 4.6** then applies to this **decldefinition** node, via the following steps:

- **DMF** applies the rules of the scoped rule **vc_struct** (**Block A'**) to the body of the structure definition. This scoped rule contains three rules, for

int, float, and other-typed fields, respectively. The first two rules transform the type to the corresponding vectorized type and update the tag with the new type name. The third rule updates the tag with $\top$, which will trigger a failure if **DMF** attempts to apply the rule.

- Reflecting the transformation of the structure definition back to the calling context, the rule in **Block A** uses the vectorized type name collected in vc_struct to transform the size of the array **input**.

Once the declaration of the array **input** and the definition of the structure type CartesianCoordinate are transformed, all the other uses of the identifier **input** and the type CartesianCoordinate are put into the work list; this includes the reference to **input**, specifically its parent, **input**[**i**] for which the guidance rule will apply. The guidance rule does not make any transformations, but rather puts the declaration of **i** from the **for** statement's initializer clause into the worklist, among other things. The immediate parent of the initializer clause is the **for** statement itself, which would be transformed.

The **for** statement can be transformed using the rule in **Block B**. The steps in this transformation are similar to those used to transform the structure.

- The **DMF** tool applies the scoped rule **vc_for**, in **Block B'**, to the body of the **for** statement in order to make it vectorizable. A vectorizable **for** statement is one whose iterations are independent of each other. In particular, we must ensure that inside such a **for** statement, all the array-indexed references that must be of the same vectorizable type. Independent iterations must also have array indexed references with the same index as the iterator of the for loop itself. This is a sufficient condition and not a necessary condition for independent iterations.
  - **DMF** then uses the array subscript expression rule and the declaration rules to update the tag **vctype**. The use of this tag ensures that all references inside the array are of the same type, which must be either float or int.

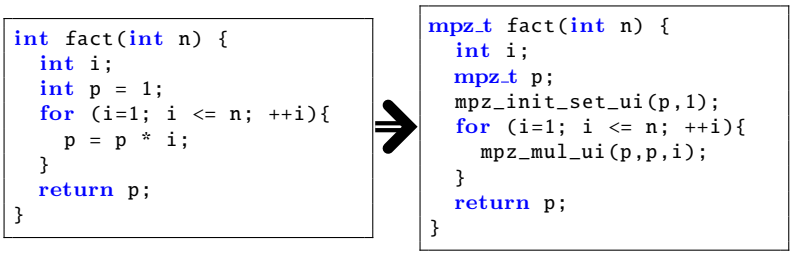
  - **DMF** uses the tag **loopindex** to ensure that all the array indexed elements share the same index. The **loopindex** is initialized to the value of the parameter *ind* (which is set to the iterator of the for statement). This ensures that **loopindex** is updated only if the index of every array indexed element i is the same as *ind*. Otherwise the value of **loopindex** would conflict and result in a tag update failure. Our specification does not provide an exhaustive characterization of when loop iterations are independent. What we have currently is an approximate solution to ensure independent iterations of loop that works on this example. For other code, the user would need to manually check that the loops are independent.
  - The **Rule CE** in **Block B‘** implements the vectorization of conditional increments. Other conditional updates can be expressed analogously.

- **Block B‘** also transforms known scalar operations into corresponding Vc operations.

Finally, **DMF** does automatic renaming (as described in Section 4.3.2.4) on all of the uses of **input** and **output**. The result is the code shown on the right side of **Figure 4.1**.

4.4.1.1 Guidance Rule

A guidance rule is a regular transformation rule that expresses a transformation that is only semantic, not syntactic. In this example, the guidance rule expresses that the array changes from scalar to vectorized. While this change requires no syntactic modifications, it does affect dependencies. Specifically, in the transformation in Figure 4.1, the algorithm gets **output**[**i**] from the worklist and examines it. The guidance rule identifies **i** as a dependency, and thereby leads our algorithm to the **for** statement, where it can perform the rest of the migration. Without the guidance rule, our algorithm would not understand that the migration needs more work at this point and needs to continue even without syntactic change at



```
int fact(int n) {
  int i;
  int p = 1;
  for (i=1; i <= n; ++i){
    p = p * i;
  }
  return p;
}
```

```
mpz_t fact(int n) {
  int i;
  mpz_t p;
  mpz_init_set_ui(p,1);
  for (i=1; i <= n; ++i){
    mpz_mul_ui(p,p,i);
  }
  return p;
}
```

FIGURE 4.7: Integer to big integer conversion using GMP

```
//This is just a part of a specification that transforms integer
//to multi precision numbers using GMP
statement int $p$ = $c$; ==>
  mpz_t $p$; mpz_init_set_ui($p$,$c$);
  notransform : $c$

expression $a$ = $b$ * $c$ ==> mpz_mul_si ($a$, $b$, $c$)
notransform: $c$
where decltype($c$) == int

expression $a$ = $b$ * $c$ ==> mpz_mul_ui ($a$, $b$, $c$)
notransform: $c$
where decltype($c$) == unsigned int

expression printf("%d", $p$) ==>
  mpz__out_str(stdout, 10, $p$)
```

FIGURE 4.8: Case Study GMP Specification

this point. The algorithm would miss the need to propagate the dependencies of **i** without the guidance rule.

4.4.2 GMP Specification

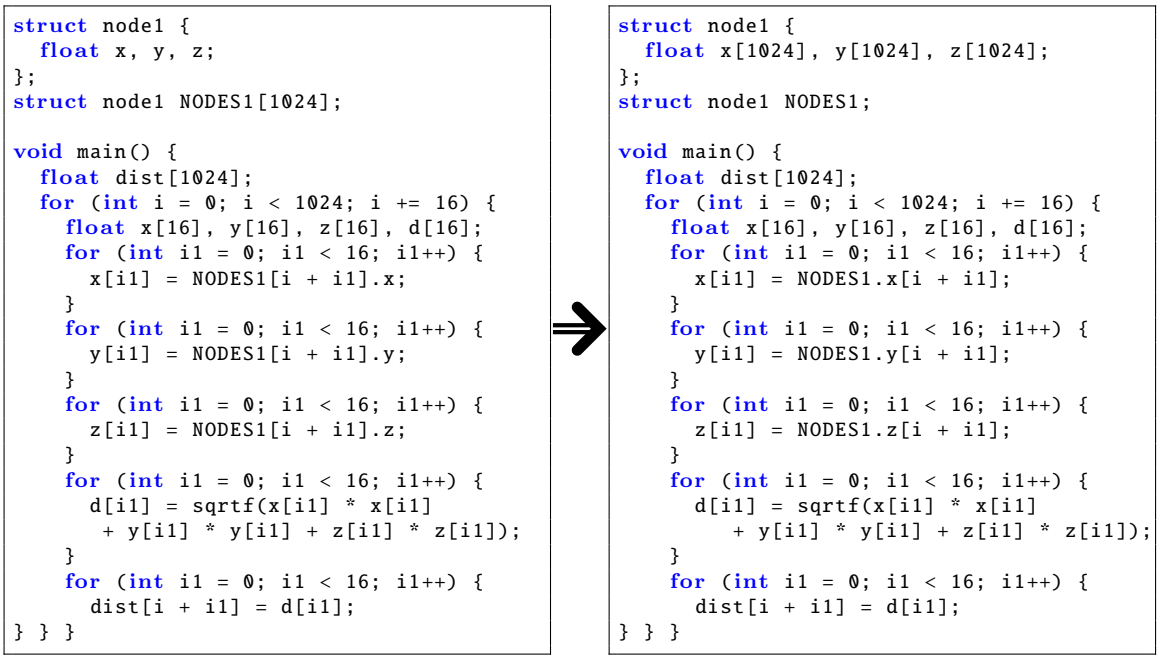
The second use case is based on an example taken out of a GMP tutorial [Sankaranarayan]. The GNU Multiple Precision (GMP) Arithmetic Library provides arbitrary precision arithmetic for C and C++ programs [GNU]. To illustrate its use, we converted an integer factorial function to work on multi-precision values and to return a multi-precision result. **Figure 4.7** shows the original code (left) and the conversion result (right), from the tutorial. **Figure 4.8** shows part of a specification that implements this transformation. This specification uses **notransform** to indicate immediate subterms that should not be added to the work list.

To initiate the transformation on the code shown on the left side of **Figure 4.7**, we select the declaration and initialization of the local variable **p**. The first rule can be applied to the initialization statement. If the declaration and initialization were two separate statements, then two different rules would be required which can be expressed using our language. The set of dependencies of the initialization statement includes the references to **p** in the assignment statement in the body of the **for** loop. There is a rule to transform assignment of a multiplication (involving unsigned int) that applies to the immediate parent of the left hand side reference of **p**. This rule converts the multiplication and assignment to a GMP library call. At this point, **DMF** would not put the uses of the variable **i** in the worklist, due to the **notransform** annotation. Finally, **DMF** traces the dependency flow into the final print (not shown) of the computed factorial and updates it with the corresponding GMP operation for printing numbers.

4.4.3 Array of Struct vs Struct of Arrays

We used **DMF** to perform the transformations shown in a tutorial published by Intel [Amanda S] that illustrates a useful data representation migration: transforming data organized as an Array of Structures (AoS) into a Structure of Arrays (SoA). This transformation allows the compiler to access data more efficiently in many applications, by improving locality. Such a transformation also helps the compiler vectorize loops that iterate over the array. The example consists of some variable declarations and an 8-line loop. The program uses array notations that are not native to C++, but are from the Cilk extension, as discussed on Intel's Cilk website.[1] In order to process the Cilk code, we wrote C code performing the same computations. It is to be noted that Cilk code is designed for parallelism and our C code will not run in parallel.

[1] https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm



```
struct node1 {
  float x, y, z;
};
struct node1 NODES1[1024];

void main() {
  float dist[1024];
  for (int i = 0; i < 1024; i += 16) {
    float x[16], y[16], z[16], d[16];
    for (int i1 = 0; i1 < 16; i1++) {
      x[i1] = NODES1[i + i1].x;
    }
    for (int i1 = 0; i1 < 16; i1++) {
      y[i1] = NODES1[i + i1].y;
    }
    for (int i1 = 0; i1 < 16; i1++) {
      z[i1] = NODES1[i + i1].z;
    }
    for (int i1 = 0; i1 < 16; i1++) {
      d[i1] = sqrtf(x[i1] * x[i1]
       + y[i1] * y[i1] + z[i1] * z[i1]);
    }
    for (int i1 = 0; i1 < 16; i1++) {
      dist[i + i1] = d[i1];
} } }
```

```
struct node1 {
  float x[1024], y[1024], z[1024];
};
struct node1 NODES1;

void main() {
  float dist[1024];
  for (int i = 0; i < 1024; i += 16) {
    float x[16], y[16], z[16], d[16];
    for (int i1 = 0; i1 < 16; i1++) {
      x[i1] = NODES1.x[i + i1];
    }
    for (int i1 = 0; i1 < 16; i1++) {
      y[i1] = NODES1.y[i + i1];
    }
    for (int i1 = 0; i1 < 16; i1++) {
      z[i1] = NODES1.z[i + i1];
    }
    for (int i1 = 0; i1 < 16; i1++) {
      d[i1] = sqrtf(x[i1] * x[i1]
         + y[i1] * y[i1] + z[i1] * z[i1]);
    }
    for (int i1 = 0; i1 < 16; i1++) {
      dist[i + i1] = d[i1];
} } }
```

FIGURE 4.9: Going from Array of Struct to Struct of Array

```
decldefinition
struct $structname$ { $structbody$ } $structobj$[$limit$]
==> struct $structname$
      { struct_scope($structbody$, $limit$) }
      $structobj$;

scope struct_scope($l$){{
  declaration $type$ $fieldname$ ==> $type$ $fieldname$[$l$]
}}

expression $name$[$i$].$field$ ==> $name$.$field$[$i$]
```

FIGURE 4.10: AoS to SoA Specification

The specification for converting a structure of arrays into an array of structures, as shown in Figure 4.10, again illustrates the usefulness of scoped rules with parameters. The first rule applies a scoped rule, **struct_scope** to the body of the struct definition. The scoped rule transforms every member field inside the struct definition into an array declaration with the limit set to **l**, which is a parameter set to the number of elements in the declared array. In our code this parameter would be **1024**. There is then one more top-level rule that modifies the array references.

4.5 Implementation and Results

We have prototyped **DMF** in Java (∼1200 LOC) using a parser that acts as a frontend for our language and Eclipse CDT [Eclipse CDT] as the basis of our transformation system. We ran our examples on an Intel Core i7-4770 CPU at 3.40GHz with 2 × 4 cores, running a 64-bit Ubuntu. For the Vc examples, we used the C++ compiler settings specified by the Vc makefiles. In this section, we report on our experiments to validate that each of our transformed programs has met the intended purpose of the transformation.

4.5.1 Vc

The primary purpose of the Vc API is to improve performance. We measured the running time of the scalar version of the loop in our non-Vc code from Figure 4.1. We compared it to the running time of the loop in the Vc code, on various architectures (AVX, AVX2, SSE), all of which are x86 extensions that support SIMD (single instruction, multiple data) instructions. For each of our settings we ran the **for** loop 100 times by wrapping the **for** loop in code which has a limit of 1000 within another for loop that runs for 100 iterations. We removed the first run to account for warm-up effects, without restarting the program. The compiler optimization levels were set to **O3** for all of the runs. **O3** instructs the C++ compiler to auto-vectorize as much as possible, thereby eliminating the possibility that the runtime measurements we get are something that the compiler could have achieved automatically. **Figure 4.11** summarizes the results. The x-axis shows the duration in microseconds, and the y-axis shows the name of the tested architecture and whether it benchmarked the post-transformation Vc version(vector) or the original unvectorized version(scalar). On each of the vector architectures, AVX, AVX2, and SSE, the non-Vc version of the loops takes more than twice as long as the corresponding vector version.

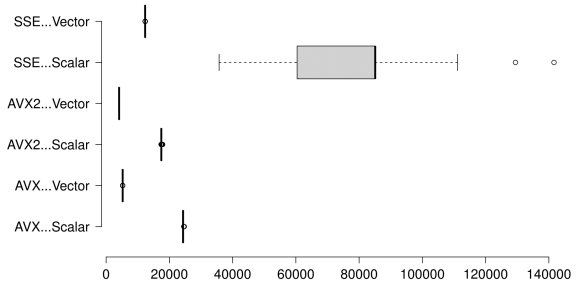


FIGURE 4.11: Vc runtimes, as box plots summarizing 99 runs.

4.5.2 GMP

The primary purpose of GMP is to represent numbers of unbounded size and precision. In our tests, for example, the int-based implemenation of factorial prints 2004310016 for the factorial of 15, indicating an integer overflow, while the generated GMP implementation prints the correct result, 1307674368000.

4.5.3 Array of Struct vs Struct of Array

Choosing between the Array of Structures and Structure of Arrays representations is a known challenge in practice [Amanda S][25], with either representation potentially advantageous, depending on usage.

The benefits of the array of structures to structure of arrays transformation and the results are discussed in detail at the website [Amanda S]. We calculated the running time of both the Array of Structures and the Structure of Arrays versions of the program in Figure 4.9 by running each version 300 times and calculating the average time per run. The array of structures runs took on average about twice as long as the structure of arrays runs.

4.6 Furter Evaluation

On top of our case studies, we also performed a few experiments with the GMP specification to illustrate the scalability and generality of our transformation approach. Scalability validates if our approach scales to large applications and generality tests whether specifications written in our language are applicable to more than one software project. We first describe the running time of our search algorithm on a larger code base and set of specifications. We then describe our experiences in applying specifications created for one program to different, randomly selected programs, and testing the transformed results.

4.6.1 Scalability

To evaluate the scalability of our approach, we selected a nontrivial piece of software to which one of our data representation migrations is pervasively applicable. As a transformation, we selected our integer to big integer conversion, since it is the most broadly applicable of our examples (requiring only the use of int values for arithmetic). As a target program, we selected a nontrivial piece of software that already uses the GMP library, and manually converted it to use integers, thus providing both the source and the desired target for our transformation process.

We selected the software by searching GitHub[2] for a project that contains `include<gmp.h>` with a source code size of at least 10,000 bytes. The first result[3] was a file of 3616 lines of code that makes 1167 references to the GMP API, involving variable declarations and function calls. The code is fairly complex, defining functions to add, multiply and do other arithmetic operations on big polynomials. We replaced all calls to GMP operations with integer arithmetic operations and verified that the resulting code compiled. We used the GMP specification (28 rules), and ran this specification on the resulting integer code and a starting point involving a declaration of a variable of type r_big_polynomial. This starting

[2] https://github.com

[3] https://github.com/PuRoTeam/PuRoAtkinMorain/blob/7620e46a59fb0e8659bfbed7ee75ad3af5de35f5/tools/polynomial.c

point was sufficient to reach all the code that used the GMP library in the original implementation. The transformations completed in 133 seconds.

The resulting code was not exactly the same as the code from which we started. Indeed, there are multiple ways to transform some kinds of integer operations. In particular, an assignment of the form a = b can be translated both to mpz_init_set(a,b) and mpz_set(a,b). The difference between the two functions is that one of the functions is used when assigning a value immediately after initializing it and the other function is used for general-purpose initialization. **DMF** currently does not support detecting such differences automatically. There were 35 such assignment related differences between the generated code and the original GMP implementation from Github. **DMF** logged the available options in cases where multiple transformations are possible and in all of the cases, we observed that the expected transformation was among the logged options. This result indicates that the approach can complete the search of a larger code with a realistic specification in a couple of minutes.

4.6.2 Generality

In order to illustrate the generality of our specifications, we applied our GMP specification to arithmetic algorithms in three algorithm libraries. We randomly selected these libraries by providing three search terms and picking the top repository obtained for each term. The search terms were:

- **arithmetic algorithms** with language set to C, which returned the repository https://github.com/celeritas17/arithmetic_algorithms.
- **algorithms** with language set to C++, which returned the repository https://github.com/PetarV-/Algorithms
- **algorithms** with language set to C, which returned the repository https://github.com/davidreynolds/algorithms

We randomly picked from these libraries six algorithms that run on integers and applied our specifications to transform their code to run on big integers. Unlike the previous experiment, we had no ground truth to compare to. The selected code and the results are summarized in Figure 4.12.

On manual inspection, we found that the **DMF** performed most of the required transformations correctly, but we did encounter a few issues. One of the examples had a condition **if ((first+second) == N)** where all of **first**, **second** and **N** had to be transformed from integer to big integer. Unfortunately, the GMP library does not provide a function that adds two big integers and returns a big integer. It merely has a function **mpz_add** which takes three parameters, where the sum of the last two is stored in the first parameter. We thus had to modify the input code to store the result in a temporary variable before the check. Another challenge, which was recurring, was the commutativity of arithmetic operations. Our specification contains rules for transforming expressions of the form **a op const** into **mpz_op_ui(a, const)** where **const** is a constant value. However, some of the expressions had the form **const op a**. We thus had to create rules for both cases, which mapped to the same GMP call. The rules differentiate the position of the constant using a **where** clause.

Partial transformation of code. One of the main goals of our algorithm was to ensure that the chosen starting point does not end up making the algorithm apply the transformation rules to all of the code that uses the existing data type. Consider the specifications written to transform integers to big integers using GMP. One of our example transformations in the Section 4.6.2 was to transform a function that finds the largest integer among an array of integers shown in Figure 4.13 into a function that uses large integers. In that example, consider the scenario where the developer just wants to convert the array elements into big integers and not any other integers in the program.

If we assume the starting point chosen was the first parameter to the function **A[]**, the transformation would be as shown in Figure 4.14.

Algo	**Description**	**Lines/Sites**	**Code (Before/after)**
Insertion Sort	Standard sort algorithm.	17/19/4	github.com/PetarV-/Algorithms/blob/b5ed8b5/Sorting%20Algorithms/Insertion%20Sort.cpp github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/insertionsort.c
Subset Sum	Find integer pairs in an integer array that sum up to a given number N.	27/33/11	github.com/davidreynolds/algorithms/blob/7d59299/c/subset_sum/int_pair_sum.c github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/intpairsum.c
Binary Search	Standard search algorithm.	16/16/4	github.com/davidreynolds/algorithms/blob/7d59299/c/searching/binary_search.c github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/binarysearch.c
Divide and conquer	Divide-and-conquer search for largest array element	17/19/3	github.com/davidreynolds/algorithms/blob/7d59299/c/searching/divide_and_conquer.c github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/divide_and_conquer.c
Fast Multi-ply	An interesting method to multiply integers.	18/28/13	github.com/celeritas17/arithmetic_algorithms/blob/5d79830/src/div_and_mult.h github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/multiplynew.c
Euclid GCD	Euclid's algorithm for finding the greatest common divisor.	14/19/6	github.com/celeritas17/arithmetic_algorithms/blob/5d79830/src/div_and_mult.h github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/euclid_gcdnew.c

FIGURE 4.12: Results of usability study of GMP specifications. Lines/Sites list lines before/lines after/transformed sites.

```
1  int find_largest(int A[], int i, int j) {
2          int m;
3          int left, right;
4
5          if (i < j) {
6                  m = (i+j) / 2;
7                  left = find_largest(A, i, m);
8                  right = find_largest(A, m+1, j);
9                  if (A[left] > A[right])
10                 return left;
11                 return right;
12         }
13         return i;
14 }
```

FIGURE 4.13: Function to find the largest number in an array of integers

```
int find_largest(mpz_t A[], int i, int j) {
        int m;
        int left, right;

        if (i < j) {
                m = (i+j) / 2;
                left = find_largest(A, i, m);
                right = find_largest(A, m+1, j);
                if (mpz_cmp(A[left], A[right]) > 0)
                return left;
                return right;
        }
        return i;
}
```

FIGURE 4.14: Function to find the largest number in an array of big integers

The algorithm would transform the **int A**[] in the parameter list of the function into **mpz_t A**[]. The dependency of this would comprise uses of **A** whose parent dependency link would comprise the comparison **A**[**left**] > **A**[**right**] in line 9. The dependencies of this comparison comprise the uses of the indices **left** and **right**. The dependencies of **left** and **right** are the declaration statement **int left, right;** (variable use and definition dependency link) in line 3 and statements assign the values to the recursive function call in lines 7 and 8 (parent dependency link).

No rules apply to both cases thereby terminating the algorithm. Of course, it is imaginable that an exhaustive set of specifications may contain rules that transform the whole function . A limitation of our approach is the dependence on the rule set. Clever uses of **notransform** and modifications to a set of specifications obtained from someone else are sometimes required to selectively transform the intended program. We have not evaluated how easy it is to use an existing specification and modifying it to examples of interest as this would involve an exhaustive

user study on how specifications written using our language evolve.

The results from the experiments performed in this section indicate that specifications written using our approach are generalizable to not just one project but to multiple code bases requiring the same kind of transformation.

Part III

Related work and conclusions

In this chapter, we will explore the scientific and industrial work that is closely related to our work on source code reuse and rewriting. For code reuse, we will explore the related work on clone detection and clone management. For rewriting, we will explore some major program transformation systems and compare it with our rewriting system targeted at data migration, **DMF**. Both **CloneMerge** and **DMF** aid software evolution by refactoring the source code. We will end the chapter with discussion on work related to general software evolution and refactoring.

Chapter 5

Related work

5.1 Software reuse

In this section, we will explore some of the most important scientific and industrial work that aid software reuse. We will begin our discussion with the techniques that permit clone detection followed by a discussion on techniques that aid managing the detected clones. Our work on source code reuse falls under the clone management techniques.

5.1.1 Clone detection

Clone detection is a well known problem in the area of software engineering and many techniques have been proposed for detecting code clones [26]. Depending on the representation of source code the analysis is performed, clone detection techniques are categorized into:

1. **Text/String based clone detection techniques**, that compare the textual representation of unprocessed source code to detect clones.

2. **Token based code clone detection techniques**, that perform clone detection on code that has been transformed into lexical tokens.

3. **Tree based clone detection techniques**, that transform source code into abstract syntax trees and detect clones by matching tree fragments

4. **Metric based clone detection techniques**, that compare various metrics related to code fragments to detect clones.

5. **Program analysis based clone detection techniques** that use static analysis to determine similarities in code.

Although this categorization is standard in clone detection literature, the categorization presented in our related work chapter is inspired by the work of Roy et al. [27]. We will now look into some of the important clone detection techniques in each category.

5.1.1.1 Text/String based clone detection techniques

The technique of using textual information to detect clones was pioneered by Johnson. His approach [28][29] works by hashing sequences of code of a particular length and identifies code with similar hashes. The technique can also be extended to work on different code lengths by using a sliding window technique on code sequences of different lengths. He termed these hashes as "fingerprints". Manber [30] also used the concept of fingerprints but to identify similar source code files and not code fragments.

Ducasse et al. [31][32] proposed a technique for detecting clones [32] based on dot plots, where both axes list source entities. Here, source entities are lines of program code and a dot in the co-ordinate (x, y) indicates that the lines x and y are equal. Every diagonal in the plot represents a clone. Type-3 clones are represented by diagonals with gaps.

Wettel & Marinescu [33] created an extension of the approach by Ducasse et al. to find near-miss clones. They use a technique that joins neighboring lines of code to detect such clones. Another approach that applies neighbor based approach to

detect near-miss clones is SDD [34] that can detect similar code which are not just exact clones.

Marcus and Maletic [35] use information retrieval to identify potential clones. They find clones at the level of abstract data types by applying latent semantic indexing (LSI) [36] to source text. This approach detects code fragments as clones when there is a high level of similarity between the source code particularly making their sets of identifiers and comments important for the clone detection process.

5.1.1.2 Token based clone detection techniques

Baker described a tool Dup [37][38] which identifies clones in a stream of tokens. The tool consists of a lexical analyzer that splits lines of source code into tokens, which are further divided into parameter (identifiers and literals) and non-parameter tokens. Parameter tokens are encoded using their lines of occurences and non-parameter tokens are encoded using a hash. A suffix tree, which is a tree in which suffixes share the same set of edges if they have a common prefix, is used to represent all prefixes of the resulting sequence of symbols. [39] Two code fragments are considered to be clones if two suffixes of the input code have more than one common prefix. Baker also summarized the technique of finding the longest common subsequence using a dynamic-programming technique [40].

Another token based technique is CP-Miner [41] that uses data mining to detect frequently occurring subsequences of code. Cordy et al. [42][43] use a token based approach to detect near-miss clones in HTML web pages. Cordy extracts code fragments of cloning interest and uses a line by line diff tool to find similarities.

CCFinder [44] is an extension to the work of Baker by Kamiya et al. An important motivation for developing CCFinder was to detect superficial bracketing like the ones surrounding **val = constant;** when comparing the code fragments **if(cond) val = const;** and **if(cond) {val=const;}**. Gemini [45] builds on top of CCFinder and provides a visualization of near-miss clones using scatter plots. Another approach that uses suffix trees to detect clones is RTF [46]. A more

memory-efficient suffix-array is used in place of traditional suffix trees that makes the tokenization customizable to the user.

5.1.1.3 Tree based clone detection techniques

Tree based clone detection techniques compare the abstract syntax trees of source code, match the nodes and detect code clones. Baxter et al. pioneered this technique by introducing their tool CloneDr [9]. This tool categorizes subtrees into buckets based on the hash values. Trees within the same bucket are then compared.

Bauhaus [47] adapted the tree based approach of Baxter et al. in their tool ccdiml and added explicit modeling of sequences, and detects the clones by searching for group of trees that form a clone. A method that converts syntax trees into XML and uses data mining to detect clones was proposed by Wahler et al. [48]. To handle the scenarios where clone detection happening only at the leaves, Evans et al. proposed Structural abstraction [49], which allows for variation in arbitrary subtrees rather than just leaves (tokens). This technique allows detection of near-miss clones with gaps. Yang proposed ccdiff [50] which also uses tree matching combined with dynamic programming. Although ccdiff is not for clone detection, but rather for finding commonalities between two programs, it can be used for clone detection also.

5.1.1.4 Metric based clone detection techniques

Metrics-based techniques gather metrics calculated from syntactic units such as a class, function, method and statement and use these metrics to compare with other code fragments and detect clones. Mostly, the metrics are calculated after converting the source code into abstract syntax trees. So these techniques can be considered as a sub-category of the tree based approaches. Kontogiannis et al. [51] proposed an approach that defines five metrics to compare code fragments. They also proposed another approach that applies dynamic programming using

minimum edit distance to compare code blocks on a statement by statement basis. They hypothesize that small edit distance is likely to be clones caused by copy-paste-modify. [51] incorporated in their tool SMC (similar methods classifier), that combines metric characterization with dynamic programming. One method that uses several metrics on functions to identify clones was proposed by Mayrand et al. [52]. A set of functions is considered a clone if their function bodies have similar metrics calculated from the program constructs.

5.1.1.5 Program analysis based clone detection techniques

In order to be more precise than mere syntactic approaches to detect code similarity, approaches which are aware of program semantics are available that perform static analysis on the code before proposing code clones. One of the main differences between these approaches from the token based approaches is that in most of these approaches, the program is represented as a program dependence graph (PDG) . PDG when compared to syntatic approaches has the added advantage of semantic connection between code fragments that share a dependency link between them. Since PDG is in essence a graph, the problem of finding clones in PDGs boils down to finding isomorphic subgraphs. There are efficient approximate algorithms that compute isomorphic subgraphs in a given graph. [53][54][55]. Komondoor and Horwitz [53] proposed one such technique that uses the backward slice of the program to detect isomorphic subgraphs in a given PDG. An iterative approach called k-length patch matching that detects maximally similar subgraphs was proposed by Krinke [54]. A recent study by Gabel et al. [56] maps PDG subgraphs to related structured syntax and have developed a plagiarism detector based on PDGs. Another approach that uses PDG was proposed by Liu et al. [55] that detects plagiarism.

Hybrid approaches Leitao [57] proposed a technique called **R2D2** that combines AST metrics (syntactic) and call graphs (semantic) to come up with special comparison functions that can be used to detect clones.

5.1.1.6 Studies on clone detection

Laguë et al. [8] find that between 6.4% and 7.5% of the source code in different versions of a large, mature code base are clones. They only count clones that are exact copies (Type-1 clones, in the terminology of Koschke et al. [6]), or copies modulo alpha-renaming (Type-2 clones). Baxter et al. [9] report even higher numbers, sometimes exceeding 25%, on different code bases and with a different technique for clone detection that also counts near clones (Type-3 clones). The prevalence of such near clones is a strong indicator that copy-paste-modify is a widespread development methodology.

Other related work on clone detection detects clones and near clones to identify faults [11] where Jurgens et al. use a novel clone detection algorithm that analyzes the impact of inconsistent clones on the correctness of the program. They validate their approach by performing a large case study on a number of inconsistent clones detected using their novel algorithm.

5.1.2 Clone management

Once clones have been detected, it is preferred that they are managed in some form or another(Appendix A). One closely related work on clone management is refactoring [58]. Code refactoring is the process of restructuring existing code, without changing its external behaviour.[1] Our work on clone merging described in Chapter 2 can be considered as a complex form of refactoring, as we transform one version of the code with clones, into another version without the clones, without changing the program's behavior. Refactoring was pioneered by Griswold and Opdyke. [59] We break our transformations into individual, atomic components [16][17], namely merges (which may be nested and require individual interaction) and fix-ups for existing code to use the re-factored ode. In his book, Refactoring to patterns [60], Kerievsky discusses techniques to refactor code to

[1] https://en.wikipedia.org/wiki/Code_refactoring

be transformed into refactored code that reflect good programming practices. Although this work is similar to ours in that our work refactors code to remove clones, the amount of manual effort required to extend the techniques presented in the book into removing clones is substantial.

Other work on clone management include tracking tools like CloneBoard [61] and Clone tracker [62]. While CloneBoard provides the ability to organize clones and to some extent to suggest the types of clones and possible resolution mechanisms, it lacks the ability to actually perform the resolution. Another approach to handling clones is linked editing [63], which maintains the clones as they are, but allows editing of multiple clones simultaneously. This has the advantage of preserving code 'as is', but the disadvantage of requiring continued tool usage for future evolution. Linked editing shares our view that copy-paste-modify is an effective way to evolve software, but disagrees on how clones should be managed; it is an open question which approach is more effective for long-term software maintenance.

Perhaps the most closely related clone management approach to our algorithm is Cedar [64], which targets Java and relies on Eclipse refactorings for abstraction. Cedar is limited to Type-2 clones. To the best of our knowledge, ours is the only work to support merging the common Type-3 clones (near clones) in a wide variety of cases. As Roy et al. [65] note, Type-3 clones are particularly common and frequently evolve out of Type-1 and 2 clones.

Our work ignores the C preprocessor [66] by operating only on preprocessed code. There is prior work on supporting the C preprocessor [67]. This work could be adapted to C++ to enable our system to support preprocessor-based abstraction patterns.

5.2 Software rewriting

Software rewriting is accomplished through transformation systems. In our work, we primarily focused on the (semi-)automated program transformations required

to perform data representation migration in Chapter 4. We begin this section by discussing features in other transformation systems and compare them to to our data migration system **DMF**. We then discuss other related work in the area of program transformation.

5.2.1 Related features

Among existing program transformation systems for C/C++, the most closely related systems to our data migration system **DMF** are Coccinelle, Stratego, TXL and Rascal MPL.

5.2.1.1 Coccinelle

Coccinelle [15] is a program matching and transformation engine that provides SmPL, the Semantic Patch Language, for specifying desired matches and transformations in C code. Syntactically, Coccinelle supports a patch-like notation for describing pattern matching and transformation rules. A simple example is:

```
@@
expression e1;
@@
- fn1(e1)
+ fn2(e1, 20)
```

The above specification declares a metavariable **e1**, matches the pattern **fn1(e1)** and transforms it to **fn2(e1, 20)**. This specification would e.g. transform the code fragment **fn1(x)** into **fn2(x, 20)**.

Coccinelle supports finding the existence of a match of a pattern, but is less well suited to checking that all instances of a pattern satisfy certain properties, as provided by our tag construct. As an example, the following Coccinelle specification reimplements our transformation of a structure containing fields that all have the same type to its vectorized counterpart:

```
1 /* get the type of the first field of each structure */
2 @r@
```

```
3  identifier i,varname;
4  type T;
5  @@
6  struct i { T varname; ... };
7
8  /* Mark other fields having the same type */
9  @others@
10 identifier r.i,varname;
11 type r.T;
12 position p;
13 @@
14 struct i { ...  T varname@p;  ... };
15
16 /* Find fields having a different type */
17 @bad@
18 identifier r.i,x;
19 type T;
20 position p != others.p;
21 @@
22 struct i { ... T x@p; ... };
23
24 /* Make a new type name if the rule 'bad' was not satisfied */
25 @script:ocaml change depends on !bad@
26 _i << r.i;
27 t << r.T;
28 t1;
29 @@
30 t1 := Coccilib.make_type (t^"_v")
31
32 /* change the types to use the new name */
33 @@
34 identifier r.i,x;
35 type r.T, change.t1;
36 @@
37 struct i {
38    ...
39 - T
40 + t1
41    x;
42    ...
43 };
```

To check that all of the fields have the same type, the first rule (lines 1–6) obtains the type of the first field. The second rule (lines 8–14) marks all of the fields that have the same type as that one. The third rule (lines 16–22) identifies whether there are fields that have another type, the fourth rule (lines 24–30) makes a new

type name if it has been found that all fields have the same type, and the fifth rule (lines 32–43) makes the transformation.

Thus, while it is possible to express ∀ quantification using Coccinelle, the implementation is much more complex, indeed amounting to ¬¬∃, than the dedicated abstraction provided by our language. Furthermore, rules in Coccinelle are triggered by syntactic pattern matching, which can be less precise than the dependency based rule triggering provided by our approach. For example, the various rules assume that all of the declarations of a structure type `i`, where `i` is a metavariable established in the first rule, are one and the same. This hypothesis could be incorrect if the code contains multiple definitions due to `#ifdefs` (Coccinelle does not first apply the C preprocessor). Our approach triggers rule applications to specific AST nodes, eliminating this ambiguity.

5.2.1.2 Stratego

Stratego is a language independent code transformation system [14]. Internally, terms are represented as abstract syntax trees, such as **Call("double", [Minue(Var("y"), Int(10))])** represents the expression double(y - 10), but front ends are provided for a variety of languages allowing the use of concrete syntax, as illustrated by the following transformation rule that removes useless zeros in addition binary expressions:

```
|[ 0 + x ]| -> |[ x ]|
```

Stratego, like our approach, permits application of rules in a guided fashion. For this purpose, Stratego supports the definition of *strategies*, which allow rewrite rules to apply conditionally and in specific user-defined orders. For example, the following strategy (inspired by the tutorial slides [2]) involving the rules `r1`, `r2`, and `r3`

```
r1 < r2 + r3
```

indicates that rules `r1`, `r2`, and `r3` should be applied as follows:

[2] http://martin.bravenboer.name/docs/gpce06-tutorial-slides.pdf

- Begin by applying `r1`. Upon successful application of `r1`, apply `r2` to the result; if the rule application fails, then apply `r3` to the original term.
- if rule application of `r2` fails, then do not backtrack.

Stratego, however, has no built-in mechanism for triggering rule application based on dependency links. Stratego also allows user to annotate terms when pattern matching which can then be used when transforming the terms, which is similar to the way tag values can be passed from matched patterns into transformations. But, tags allow for basic program analysis and propagation of information based on a for all quantification which we have found to be useful with data representation migration, which is not supported by Stratego's annotations.

5.2.1.3 TXL

TXL [68] is a language used by software tool designers for creating and rapidly prototyping their language tool ideas. It is designed to allow programmers to perform source code analysis and transformation tasks. The inputs to TXL are a context free grammar in BNF like notation and a set of transformation rules that apply to input programs in that specified language. Figure 5.1 shows an example of a part of a specification of a top level rule in **DMF**.

```
define program
        [category] [dmfexpression] === [dmfexpression]
end define

define category
        declaration |
        decldefinition |
        statement |
        expression
end define

// The specification is not complete and is only used to explain the basic features of TXL
```

FIGURE 5.1: Grammar for numerical expression evaluator

A *define* statement provides the description of one *non-terminal* type. A non-terminal type is used to describe a component of the language and has to be defined

later. Non-terminals occur inside brackets, and symbols that appear outside of any such brackets are terminals, in our case the terminals would be **declaration, decldefinition, statement, expression**. Terminal symbols indicate the code that appears exactly as it is in the program written in the language. In addition, TXL grammars also supports specifying keywords.

Now, let us consider an example transformation rule defined in TXL to transform simple struct definitions with struct definitions with a new name, in order to understand the basic rule specification in TXL.

```
rule structdefinition
    replace [dmfexpression]
        struct structname[id] {structbody[id]}
    by
        struct structname _v {structbody}
end rule
```

FIGURE 5.2: Transformation rules for numerical expression evaluator

The rule in Figure 5.2 shows a simple rule specification in TXL. TXL, similar to **DMF** support pattern matching and transformation of code belonging to a particular syntactic category. In the example, the category is **dmfexpression** and it transforms a struct with name **structname** which is flagged as an **id** (id is a TXL built in non-terminal) into a struct with a new name **structname** followed by **_v**. TXL, like **DMF** also supports **where** clauses which filter rule application and simple ordering of rules. The ordering cannot be based on program dependence as supported by **DMF**.

DMF currently supports the data migration transformation only for C and C++ code. TXL is more general than our language as TXL allows definition of custom languages where rules can be applied. TXL also supports transformation in-place, which our language does not currently support. But, TXL does not support propagating transformations based on data dependencies or propagation of information between rules as a result of simple program analysis (tags), as needed for data representation migration.

5.2.1.4 Rascal

Rascal [69] is a meta programming language like TXL and provides support for expressing context-free grammars and transformation rules. Rascal provides libraries that act as front-ends for multiple languages, including Java and C.

Rascal, like TXL is more general and supports application of transformation rules to multiple languages compared to our data migration system **DMF**. Therefore, Rascal is more suited to integrate software tools written in multiple languages. Rascal allows developers to use the visitor pattern to specify transformation rules that are applied whenever a particular type of code fragment is reached. Since **DMF** also internally uses the visitor pattern to implement the data driven transformation approach, it can be envisioned that DMF can be implemented as an extension to Rascal.

Rascal is a language workbench that supports defining textual languages using a meta-language. Other such language workbenches include Spoofax [3] and MPS [4]. Language workbences are targetted at generating new languages and not particularly tuned to handle the particular challenges of data representation migration like selectively applying transformation rules based on program dependence as supported by **DMF**.

5.2.1.5 IDE Refactorings

Refactoring environments such as Eclipse [22] and Netbeans,[5] provide simple code transformations like 'rename variable', some of which are supported by data-driven transformations. For example, when a variable is renamed in one location, then all of its uses and declarations are transformed automatically. However, refactoring systems offer only a fixed set of transformations that are difficult or impossible for the user to extend or customize.

[3] http://metaborg.org/en/latest/
[4] https://www.jetbrains.com/mps/
[5] http://wiki.netbeans.org/Refactoring

5.2.2 Other program transformation related work

One of the first works on program transformation languages was that of Burstall and Darlington [70], who described a theoretical framework for describing program transformation systems. They primarily targeted the problem of program derivation, rather than data representation migration issues.

Our work is closely related to term rewriting tools. Tom [71] is a non-intrusive tool for extending existing programming languages with pattern matching. It is very lightweight and can be used to write simple term rewriting extensions for a variety of imperative languages. Tom, however, does not provide abstractions for managing data propagation information and dependencies, and thus falls short when addressing the issues of expressing rewriting rules for data representation migrations. Batory et al [72] proposed an approach for data structures in the form of minimal libraries that contain primitive building blocks and generators. Although such an approach can be used to build abstract data structures that can have many implementations and these implementations ease the migration between the implemented data types, our approach supports the problem of data migration and the challenges associated with it directly.

Our work also involves program analysis, through our tags. Although program analysis has been used to address issues related to performance and security, few works incorporate program analysis along with program transformation. Spoon [73] is a library for implementing analyses and transformations of Java source code. Although Spoon provides features that can be used to combine program analysis with program transformation, the developer has to write their own transformation and analysis code using a CDT-like low-level library by hand. Our approach on the other hand simplifies the task of writing transformation rules for migrating data representations by providing simple forms of program analysis and information propagation, via scopes and tags.

Our work is a form of incremental refactoring of the source code, by incrementally selecting transformations to be applied. Reichenbach et al. [74] also present

an incremental transformation approach, with behavioral guarantees ensured by a general comparison mechanism, albeit for traditional refactorings and with much more limited support for automation. Schäfer et al. [75] analogously present a general approach to implementing software refactorings by viewing them as invariant-preserving transformations on programs in an enriched language. Although both approaches use the idea of incremental refactoring to aid development of complex refactoring systems, they fail to address the specific challenges of data representation migration, *i.e.*, the ability to express propagation of information across code fragments and flagging dependencies for further changes. Our work can also be looked as a form of fixing type errors, if we consider our approach to help the problem of changing the representation of program to use the correct types. The most notable work on fixing type errors automatically was proposed by McAdam [76] where they use a bottom up approach based on a construct called type isomorphisms to suggest methods to repair programs with type errors.

Our work also involves transformations of program to create vectorizable loops. One of the earliest works on program transformations that perform loop trasnformations was from Loveman [77]. Loveman presented a set of source to source transformations that not only covered the classical transformations like dead-code elimination and constant propagation, but also transformations that are applied on loop constructs like loop unravelling and loop fusion. **DMF** performs syntactic source to source transformations. One of the earliest suggestions on how source to source transformation systems can be built in a way that they preserve the flow of computation was suggested by Arsac [78].

5.3 Software refactoring

We already saw a few refactoring approaches in the presentation of the related work specific to software reuse and software rewriting. This thesis in essence is a collection of refactoring approaches targeted at software evolution. In this section, we will look at a few general state-of-the-art refactoring approaches. The most

notable connection between software evolution and refactoring tools can be seen by the fact that Simmonds and Mens [79] perform a comparison of refactoring tools based on the taxonomy of software evolution presented by Mens et al. [80]. Mealy and Strooper [81] further expand upon this taxonomy to arrive at a framework that helps to evaluate software refactoring tools. This evaluation strategy is based on the DESMET method [82] which aids evaluation of software engineering methods. In general refactoring tools are interactive and require user involvement to guide the refactoring process, but there are a few fully automated refactoring approaches. Fully automated refactoring approaches completely eliminate the user and detects refactoring opportunities by themselves. Moore developed a tool Guru [83] that automatically extracts common portions of methods into a new method. Moore's approach is superior to normal extract method refactoring in that it can also create super classes whenever required during the pulling up of method. Refactoring changes program code and therefore may end up changing program behavior which may not be intended. Soares and Gustavo proposed an approach called SafeRefactoring [84] that makes refactoring safer by identifying bugs that lead to behavioral changes as part of the refactoring process. Refactoring tools, like any other software tool used by software developers, suffer from usability issues. Campell and Miller [85] came up with the prominant issues that plague refactoring and ways in which they can be handled.

Both **DMF** and **CloneMerge** transform C and C++ code. There are not a lot of refactoring approaches that are targeted at these languages. C supports conditional compilation using preprocessor directives which is a challenge for refactoring tools. Garrido and Johnson [86] propose an approach that enables refactoring of C programs in the presence of such conditional compilation directives. Most of the tool support for refactoring is centered around object-oriented and imperative languages. Li et al. [87] proposed an approach that provides tool support for functional languages and evaluated it in Haskell.

Most of modern software refactoring takes place in the presence of Integrated development environments like Eclipse. Xing and Stroulia [88] exploited this to by studying how refactorings take place in evolving software that is managed using

Eclipse. They used their study to gain insights on the proportion of code being refactored and the most common types of refactoring in practice. One of the key challenges in the process of refactoring is knowing where to refactor. Simon et al. [89] came up with various metrics that can be useful in suggesting code locations which present refactoring opportunities.

It is well known that code refactoring improves code readability and maintainability. But one concern that refactorings pose is the amount of performance overhead a refactored code presents. Sahin et al. [90] performed a comprehensive study by applying 6 common refactorings to 197 applications and reporting on their impact on energy usage.

5.4 Software Evolution

Although the primary focus of our contributions are related to program analysis and transformation, the broader goal of our work is to aid software evolution. In this section, we discuss some of the related work in the area of Software evolution.

Software evolution was a term first coined in 1974 by Lehmann in his book *Programs, Cities, Students: Limits to Growth?* [91]. Throughout the 70s the evolution of software was studied and one of the major breakthroughs came when he observed that software undergoes maintenance and evolution not only as a result of bad project planning or code deficiencies. They observed that software maintenance and evolution is also triggered by changing customer preferences and management policies [92]. The early studies on software evolution focused primarily on large systems and how organizations developed them for practicality [93]. The conclusion from these studies were that the organizations that maintains the software and the software that evolves as a result of this maintenance constitute a multi-agent, self-stabilizing, multi-loop feedback system [94]. These studies led to the better understanding of how software evolves and eventually to the set of laws governing software evolution [95][96].

Eventually, the investigation by Lehman and Belady led to other investigators gaining interest and conducting their own studies on software evolution, e.g., [97][98][99]. These investigations led to modelling of software growth and improving the predictive power using these models [100]. Confidence in these models was further increased by simulating qualitative models for software evolution [101]. Lehman also proposed the first theory of software engineering which provides a foundational background to understanding evolving software [102].

Chapter 6

Conclusion

In this thesis, we looked at significant challenges that a developer faces when evolving source code, specifically when reusing old code and rewriting existing code. In the following sections, we discuss our conclusions from each of these challenges

6.1 Reusing old code

Managing code clones is a significant problem, given the amount of copied and pasted production-level code. This suggests that developers find reuse through code clones useful in practice, even when they know that reuse through manual abstraction would yield superior and more maintainable code; we find this confirmed both by prior work and by an informal poll that we conducted among C++ developers. We propose to close the gap between reuse through copy-paste based clones and abstraction through semi-automatic refactoring.

We have implemented a prototype of a suitable refactoring tool that identifies the parts of clones that can be merged, and proposes to the user suitable resolution patterns. The user then makes the design decision of how to merge. We have evaluated this approach by implementing a prototype merging tool and applying a select set of resolution patterns to near-clones in popular Github repositories. We

submitted the merged code back to the developers via pull requests and observed that the original developers found more than 50% (90% with the most recent version of our tool) of our changes to be desirable, merging them into their code bases.

6.2 Rewriting existing code

We have analyzed real world data representation migrations and identified core features that an automated framework must support to perform such transformations. We have designed a language around these features and implemented a prototype tool supporting this language. We have evaluated our language on data representation migration examples chosen from external websites. We have also illustrated the scalability of our algorithm by applying a large specification to a non-trivial code base and have established the difference between our approach and other program transformation frameworks and tools. We therefore find that our approach is effective at expressing and performing a variety of transformation tasks, and offers a novel view on datatype migration.

6.3 Overall contributions and future work

In this work, we have explored specific challenges that a developer faces when evolving source code and have come up with program analysis and transformation approaches that automate the same. We have hopefully provided a glimpse of the program analysis and tool support that are most relevant for practitioners when developing software tools that support source code evolution. We propose the following future work that are of importance in our opinion, based on our experience with the thesis:

1. Develop a specification mining tool which creates specifications using the language of **DMF** using a set of manually performed data representation migrations

2. Integrate the clone merging approaches with a customizable clone detection tool

3. Extend both the language and the clone merging work to be usable by other popular languages, especially Java.

Appendix A

Coding Tasks and Programmer Poll

This appendix summarizes our informal poll. We asked five students (**Table A.1**) to perform reuse tasks with copy-paste-modify and with manual abstraction; **Figure A.1** summarizes the amount of time taken to complete the tasks. Each graph represents one task. For each task, x-axis is the time taken (in minutes) and y-axis indicates the user. The red triangles represent the time taken for copy-paste tasks and the blue rectangle represent the time taken for abstraction tasks. Whenever one student performed both copy-paste-modify *and* manual abstraction, the student first completed the copy-paste-modify tasks. We later polled students as to whether they would prefer for the outcome to have been copy-paste-modified code or abstracted code. Four students responded; we summarize their responses for each task in **Figure A.2**.

TABLE A.1: Student experience levels (self reported).

Student	#1	#2	#3	#4	#5
Experience	10 yr	3 mo	4 yr	1 yr	2 mo

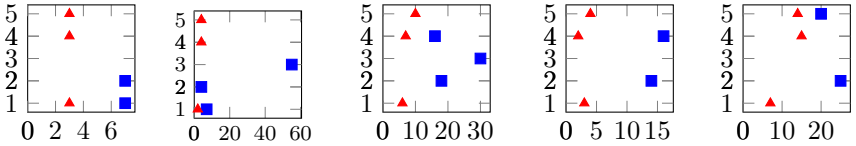


FIGURE A.1: Amount of time used for extending functionality. x-axis = time taken, y-axis = user, red triangle = copy-paste, blue triangle = abstraction

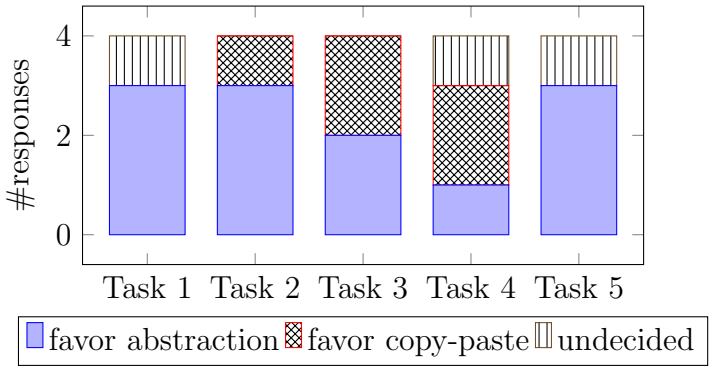


FIGURE A.2: Preferred results after extending functionality. Out of the 20 answers we received, 3 were undecided, 5 preferred copy-pasted code, and 12 preferred abstracted code.

Bibliography

[1] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Comput. Surv.*, 46(2):28:1–28:28, December 2013. ISSN 0360-0300. doi: 10.1145/2543581.2543595. URL http://doi.acm.org/10.1145/2543581.2543595.

[2] ISO/IEC. International standard - iso/iec 14764 ieee std 14764-2006 software engineering 2013; software life cycle processes 2013; maintenance. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pages 10–46, 2006. doi: 10.1109/IEEESTD.2006.235774.

[3] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004. ISBN 0321210263.

[4] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-83595-9.

[5] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, September 1984. ISSN 0164-1212. doi: 10.1016/0164-1212(79)90022-0. URL http://dx.doi.org/10.1016/0164-1212(79)90022-0.

[6] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1.

[7] Cory J. Kapser and Michael W. Godfrey. cloning considered harmful considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008. ISSN 1382-3256.

[8] Bruno Laguë, Daniel Proulx, Ettore M. Merlo, Jean Mayrand, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 314–321. IEEE Computer Society Press, 1997.

[9] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8779-7. URL http://dl.acm.org/citation.cfm?id=850947.853341.

[10] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495, May 2009.

[12] Matthias Kretz and Volker Lindenstruth. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, 2012.

[13] Matthias Kretz. Efficient use of multi- and many-core systems with vectorization and multithreading. Master's thesis, University of Heidelberg, 2009. URL http://code.compeng.uni-frankfurt.de/attachments/13/Diplomarbeit.pdf.

[14] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. URL http://www.sciencedirect.com/science/article/pii/S0167642308000452. Special Issue on Second issue of experimental software and toolkits (EST).

[15] Yoann Padioleau, Julia L. Lawall, Ren Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

[16] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program Metamorphosis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 394–418, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3.

[17] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In *ECOOP*, pages 369–393, 2009.

[18] M. Pawlik and N. Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. In *Proceedings of the VLDB Endowment, Vol. 5, No. 4*, 2011.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[20] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 79–103, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31056-0. doi: 10.1007/978-3-642-31057-7_5. URL http://dx.doi.org/10.1007/978-3-642-31057-7_5.

[21] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 219–220, June 2011. doi: 10.1109/ICPC.2011.26.

[22] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developers Guide to Eclipse*. Addison-Wesley, May 2003.

[Kretz] Mathias Kretz. Vc: Cartesian to polar co-ordinates. URL http://code.compeng.uni-frankfurt.de/docs/Vc-0.7/ex-polarcoord.html.

[cppreference.com] cppreference.com. decltype specifier. http://en.cppreference.com/w/cpp/language/decltype.

[GNU] GNU. GMP library. https://gmplib.org/.

[23] 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, July 1987.

[24] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. Technical report, Tucson, AZ, USA, 1997.

[Sankaranarayan] Sriram Sankaranarayan. Tutorial on GMP. https://www.cs.colorado.edu/~srirams/courses/csci2824-spr14/gmpTutorial.html.

[Amanda S] Amanda S. Memory layout transformations. https://software.intel.com/en-us/articles/memory-layout-transformations.

[Eclipse CDT] Eclipse CDT. Eclipse CDT. http://www.eclipse.org/cdt/.

[25] Kamal Sharma, Ian Karlin, Jeff Keasler, James R McGraw, and Vivek Sarkar. Data layout optimization for portable performance. In *European Conference on Parallel Processing*, pages 250–262. Springer Berlin Heidelberg, 2015.

[26] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY*, 115, 2007.

[27] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2009.02.007. URL http://www.sciencedirect.com/science/article/pii/S0167642309000367.

[28] J. Johnson. Identifying redundancy in source code using fingerprints, in: Proceedings of the 1993 conference of the centre for advanced studies on collaborative research, cascon 1993. *pp*, 171, 1993.

[29] J. Johnson. Visualizing textual redundancy in legacy source, in: Proceedings of the 1994 conference of the centre for advanced studies on collaborative research, cascon 2004. *pp*, pages 171–183, 1994.

[30] U. Manber. Finding similar files in a large file system, in: Proceedings of the winter 1994 usenix technical conference. *pp*, pages 1–10, 1994.

[31] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, 2006.

[32] S. Ducasse, M. Rieger, and A S. Demeyer. Language independent approach for detecting duplicated code. In *Proceedings of the 15th International Conference on Software Maintenance*, pages 109–118. ICSM 1999, 1999.

[33] R. Wettel and R. Marinescu. Archeology of code duplication: recovering duplication chains from small duplication fragments. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pages 8 pp.–, Sept 2005. doi: 10.1109/SYNASC.2005.20.

[34] S. Lee and I. Jeong. Sdd: High performance code clone detection system for large scale source code. In *141*, systems, languages, and applications, OOPSLA Companion 2005, pp. 140â, 2005. Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming.

[35] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 107–114. ASE 2001, 2001.

[36] Susan T. Dumais. Latent semantic analysis. *Annual Review of Information Science and Technology*, 38(1):188–230, 2004. ISSN 1550-8382. doi: 10.1002/aris.1440380105. URL http://dx.doi.org/10.1002/aris.1440380105.

[37] B. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering*, 33(9):608–621, 2007.

[38] B. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 86–95. WCRE 1995, 1995.

[39] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3): 331–353, 1997. ISSN 1432-0541. doi: 10.1007/PL00009177. URL http://dx.doi.org/10.1007/PL00009177.

[40] B. Baker and R. Giancarlo. Sparse dynamic programming for longest common subsequence from fragments, journal algorithms. *Vol.*, 42(2):231–254, 2002.

[41] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[42] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. *Special issue on WCREâ0*, 8, 2009.

[43] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software systems. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 81–90, (2008). [100] C.K.Roy and J.R. Cordy, WCREâ08 Clones, Last accessed, November 2008. WCRE 2008.

[44] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[45] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, pages 185–197. PROFES 2002, 2002.

[46] H. Basit, S. Pugliesi, W. Smyth, A. Turpin, and Efficient S. Jarzabek. Token based clone detection with flexible tokenization, in: Proceedings of the 6th european software engineering conference and foundations of software engineering, esec/fse 2007. *pp*, pages 513–515, 2007.

[47] Project Bauhaus. URL Last accessed, 2008. URL http://www.bauhaus-stuttgart.de.

[48] V. Wahler, D. Seipel, J. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques, in: Proceedings of the 4th ieee international workshop source code analysis and manipulation, scam 2004. *pp*, pages 128–135, 2004.

[49] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction, acm transactions on programming languages and systems. *Vol.*, 22(2):378–415, 2000.

[50] W. Yang. Identifying syntactic differences between two programs. *SoftwarePractice and Experience*, 21(7):739–755, 1991.

[51] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the IEEE Symposium on Software Metrics*, pages 292–303. METRICS 1999, 1999.

[52] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th International Conference on Software Maintenance*, pages 244–253. ICSM 1996, 1996.

[53] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. SAS 2001, 2001.

[54] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309. WCRE 2001, 2001.

[55] C. Liu, C. Chen, J. Han, and P. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881. KDD 2006, 2006.

[56] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, pages 321–330. ICSE 2008, 2008.

[57] A. Leitao. Detection of redundant code using r2d2. *Software Quality Journal*, 12(4):361–382, 2004.

[58] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[59] W. G. Griswold and W. F. Opdyke. The birth of refactoring: A retrospective on the nature of high-impact software engineering research. *IEEE Software*, 32(6):30–38, Nov 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.107.

[60] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004. ISBN 0321213351.

[61] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, 2009.

[62] E. Duala-Ekoko and M. P. Robillard. Clonetracker: Tool support for code clone management. In *ICSM*, 2008.

[63] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC*, 2004.

[64] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12): 1297–1307, December 2012. ISSN 0950-5849.

[65] C.K. Roy, K.A. Schneider, and D.E. Perry. Understanding the evolution of Type-3 clones: An exploratory study. In *MSR*, 2013.

[66] Flavio Medeiros, Christian Kästner, Mrcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *ECOOP*, 2015.

[67] Paul Gazzillo and Robert Grimm. SuperC: Parsing all of C by taming the Preprocessor. *SIGPLAN Not.*, 47(6):323–334, June 2012. ISSN 0362-1340.

[68] James R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190 – 210, 2006. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2006.04.002. URL http://www.sciencedirect.com/science/article/pii/S0167642306000669.

[69] P. Klint, T. v. d. Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, Sept 2009. doi: 10.1109/SCAM.2009.28.

[70] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, January 1977.

[71] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 61–76. Springer-Verlag, 2003.

[72] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. *SIGSOFT Softw. Eng. Notes*, 18(5):191–199, December 1993. ISSN 0163-5948. doi: 10.1145/167049.167078. URL http://doi.acm.org/10.1145/167049.167078.

[73] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, page na, 2015.

[74] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. *Program Metamorphosis*, pages 394–418. Springer, 2009.

[75] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege Moor. Stepping stones over the refactoring Rubicon. In *ECOOP*, pages 369–393. Springer-Verlag, 2009.

[76] Bruce J. McAdam. How to repair type errors automatically. In *Selected Papers from the 3rd Scottish Functional Programming Workshop (SFP01)*, SFP '01, pages 87–98, Exeter, UK, UK, 2001. Intellect Books. ISBN 1-84150-070-4. URL http://dl.acm.org/citation.cfm?id=647188.720010.

[77] David B. Loveman. Program improvement by source to source transformation. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 140–152, New York, NY, USA, 1976. ACM. doi: 10.1145/800168.811548. URL http://doi.acm.org/10.1145/800168.811548.

[78] Jacques Arsac. *Langages Sans Etiquettes et Transformations de Programmes*, pages 112–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974. ISBN 978-3-662-21545-6. doi: 10.1007/978-3-662-21545-6_8. URL http://dx.doi.org/10.1007/978-3-662-21545-6{_}8.

[79] Jocelyn Simmonds and Tom Mens. A comparison of software refactoring tools, 2002.

[80] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a taxonomy of software evolution, 2003.

[81] E. Mealy and P. Strooper. Evaluating software refactoring tool support. In *Australian Software Engineering Conference (ASWEC'06)*, pages 10 pp.–, April 2006. doi: 10.1109/ASWEC.2006.26.

[82] Barbara Ann Kitchenham. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *SIGSOFT Softw. Eng. Notes*, 21(1):11–14, January 1996. ISSN 0163-5948. doi: 10.1145/381790.381795. URL http://doi.acm.org/10.1145/381790.381795.

[83] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 235–250, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: 10.1145/236337.236361. URL http://doi.acm.org/10.1145/236337.236361.

[84] Gustavo Soares. Making program refactoring safer. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 521–522, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1810295.1810461. URL http://doi.acm.org/10.1145/1810295.1810461.

[85] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 9:1–9:2, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-339-6. doi: 10.1145/1636642.1636651. URL http://doi.acm.org/10.1145/1636642.1636651.

[86] A. Garrido and R. Johnson. Refactoring c with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 323–326, Oct 2003. doi: 10.1109/ASE.2003.1240330.

[87] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 27–38, New York, NY, USA, 2003. ACM. ISBN 1-58113-758-3. doi: 10.1145/871895.871899. URL http://doi.acm.org/10.1145/871895.871899.

[88] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Sept 2006. doi: 10.1109/ICSM.2006.52.

[89] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001. doi: 10.1109/.2001.914965.

[90] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652538. URL http://doi.acm.org/10.1145/2652524.2652538.

[91] M.M. Lehman. *Programs, Cities, Students: Limits to Growth?* Inaugural lecture - Imperial College of Science and Technology. Imperial College of Science and Technology, University of London, 1974. URL https://books.google.de/books?id=j4pXGwAACAAJ.

[92] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-442440-6.

[93] L.A. Belady and M.M. Lehman. *Programming System Dynamics or the Meta-Dynamics of Systems in Maintenance and Growth*. IBM Thomas J. Watson Research Center, 1971. URL https://books.google.de/books?id=o8R2QwAACAAJ.

[94] M.M. Lehman. Feedback in the software evolution process. *Information and Software Technology*, 38(11):681–686, 1996. ISSN 09505849. doi: 10.1016/0950-5849(96)01121-4. URL http://dx.doi.org/10.1016/0950-5849(96)01121-4.

[95] Meir M Lehman and Juan F Ramil. Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering*, 11(1):15–44, 2001. ISSN 1573-7489. doi: 10.1023/A:1012535017876. URL http://dx.doi.org/10.1023/A:1012535017876.

[96] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, September 1984. ISSN 0164-1212. doi: 10.1016/0164-1212(79)90022-0. URL http://dx.doi.org/10.1016/0164-1212(79)90022-0.

[97] A. Bauer and M. Pizka. The contribution of free software to software evolution. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 170–179, Sept 2003. doi: 10.1109/IWPSE.2003.1231224.

[98] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, pages 131–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0753-0. URL http://dl.acm.org/citation.cfm?id=850948.853411.

[99] M. J. Lawrence. An examination of evolution dynamics. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, pages 188–196, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press. URL http://dl.acm.org/citation.cfm?id=800254.807761.

[100] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA,

2000. ACM. ISBN 1-58113-253-0. doi: 10.1145/336512.336534. URL http://doi.acm.org/10.1145/336512.336534.

[101] Juan F. Ramil and Neil Smith. Qualitative simulation of models of software evolution. *Software Process: Improvement and Practice*, 7(3-4):95–112, 2002. ISSN 1099-1670. doi: 10.1002/spip.158. URL http://dx.doi.org/10.1002/spip.158.

[102] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 70–74, New York, NY, USA, 2001. ACM. ISBN 1-58113-508-4. doi: 10.1145/602461.602473. URL http://doi.acm.org/10.1145/602461.602473.

Zussamenfassung

Bei dieser Arbeit werden wir uns einige unerforschte Herausforderungen ansehen, die bei der Entwicklung von Analysen und Transformationen involviert sind, die die Evolution von Quellcode unterstuetzen. Wir behandeln deren Wichtigkeit bei Nutzerstudien und aehnlicher Arbeit. Wir erstellen Programmanalysen und transformationsbasierte Herangehensweisen, um diese Herausforderungen zu meistern. Wir erstellen Prototyp-Tools aus diesen Herangehensweisen. Wir schaetzen die Effektivitaet, Verwendbarkeit und Machbarkeit unserer Herangehensweise ein. Wir berichten ueber Ergebnisse und Erkenntnisse. Unser Hauptfokus liegt auf dem Hinzufuegen/ Modifizieren von Funktionen und auf Softwareanpassung als Wartung, da diese zentral fuer die Entwicklung von Software sind. Fehlerreparatur faellt unter die Kategorie Korrekturwartung, die, wie wir uns erinnern, nicht als Entwicklungswartung kategorisiert ist. Die Kernhypothese unserer These ist es, dass Softwareentwicklungsaufgaben kompliziert genug sind, dass sie automatisiert werden muessen. Aber die Tools, die fuer diese Automatisierung erstellt wurden, sollten im wesentlichen den Entwicklern helfen, und daher muessen die Entwickler das letzte Wort darin haben, wie die Entwicklung vor sich geht. Wir schlagen vor, zu einem Programmanalyse- und

Transformations-Framework zu gelangen, das der Softwareentwicklung hilft, indem eine Automatisierung entwickelt wird, die durch Nutzereingabe gesteuert wird. Insbesondere machen uns zwei Aufgaben Sorgen, die waehrend der Softwareentwicklung entstehen, und die erst noch gruendlich erforscht werden muessen. Aufraeumen von Copy-Paste-Klonen. Das Hinzufuegen von Features geschieht durch das Einfuehren neuer Funktionalitaet. Neue Funktionalitaet ist manchmal der existierenden Funktionalitaet sehr aehnlich und Entwickler verwenden bestehende Funktionalitaet auch erneut wieder. Wir sehen uns zwei markante Methoden an, wie man Code auf diese Art wiederverwendet, d. h. Abstraktion von Funktionen und Copypaste-modify (nderung des Copy-Paste?). Wir fuehren Nutzerstudien durch, um diese Methoden der Wiederverwendung zu vergleichen und unsere Hypothese zu errichten, dass copy-paste-modify einfacher ist, aber dass Abstraktion bevorzugt wird. Wir schlagen vor, dass Tool-Support Code-Klone aufraeumen muss, die von der copy-paste-modify Methode herruehren, indem gut gewaehlte Abstraktionen verwendet werden. Migration der Datendarstellung. Softwareanpassung involviert typischerweise Code-Transformationen. Es existiert eine Unzahl von Tools und Frameworks, die Entwicklern mit automatisierten Code-Transformationen helfen. Die beiden Hauptkategorien von Tools, die bei der Code-Transformation helfen, sind Refactoring Tools (Restrukturierungstools) und Transformation Languages (Transformationssprachen). Refactoring Tools bieten einfache Transformationen wie Umbenenn-Variablen oder Extraktionsmethoden. Refactoring Tools haben den Vorteil, dass Sie die Dependencies bei der Transformation an einem Ort des Codes nachverfolgen und die nderung in alle Dependencies uebernommen werden, wodurch es dem Nutzer

erlaubt wird, die Transformationen selektiv anzuwenden. Aber Refactoring Tools haben ein spezifisches vorbestimmtes Set an Transformationen. Eine Alternative zu solchen Transformation sind Transformation Languages, die anpassbare Regeln unterstuetzen, in denen der Entwickler so viele Transformationsregeln erstellen kann, wie er moechte und diese auf seinen Code anwenden kann, um Featureanpassungen zu unterstuetzen. Aktuelle Transformation Languages unterstuetzen keine selektive Anwendung von Regeln. Wir stellen die Hypothese auf, dass eine spezifische Art der Softwareanpassung, d. h. der Migration von Software von einer Datendarstellung zu einer anderen, sowohl die Einstellbarkeit (Customizability) von Transformation Languages als auch eine interaktive, selektive Transformation des Refactoring erfordert. Wir schlagen vor, dass Tool-Support erforderlich ist, um beide diese Funktionen zu unterstuetzen. In den folgenden Unterabschnitten besprechen wird die Wichtigkeit und unseren Beitrag zu den oben erwaehnten Herausforderungen im Detail. Wenn Softwareentwickler neue Funktionen zu ihrer Software hinzufuegen, entdecken sie oft, dass eine neue Funktion einer alten sehr aehnlich ist. Der Entwickler steht dann vor einer Entscheidung: Entweder er fuehrt eine (moeglicherweise komplexe) Abstraktion in den bestehenden, funktionierenden Code, oder er kopiert (Copy Paste) den bestehenden Code und modifiziert das Ergebnis. Die Einfuehrung einer Abstraktion erzeugt kleineren, besser wartbaren Code, aber sie veraendert die bestehende Funktionalitaet auf der Betriebsebene (operational level), was das Risiko beinhaelt ungewollt semantische nderungen einzufuehren. Copy-Paste-Modify fuehrt eine Duplizierung in Form von Fast-Klonen (near clones) ein (Type-3 Clones in der Terminologie von Koschke et al., d. h. Klone mit nicht-trivialen Unter-

schieden (non-trivial differences)), was haeufig zu schlechterer Wartbarkeit fuehrt, aber die Risiken der Beschaedigung bestehender Funktionalitaet vermeidet . Duplizierung ist weit verbreitet , besonders, wenn wir Fast-Klone mitzaehlen. Duplizierung ist jedoch in der Praxisliteratur unbeliebt und kann ein substantielles Problem bei der Entwicklung und Wartung sein , da inkonsistente Klone eine Fehlerquelle darstellen. Gleichermaen bevorzugten die C++ Entwickler unserer Studie (Abschnitt 2.2.1) es, Code mithilfe von Abstraktion zu lesen, und nicht mit Duplizierung. Daraus schlieen wir, dass es eine Diskrepanz gibt, die wir die Reuse-Discrepancy [Wiederverwendungsdiskrepanz] nennen. Dies ist die Diskrepanz zwischen dem, was Entwickler wollen und dem, was sie tun. Kapser und Godfrey bieten eine moegliche Erklaerung: Sie behaupten, dass Code-Cloning in verschiedenen Situationen als effektive und vorteilhafte Entwicklungspraxis verwendet werden kann, aber sie bemerken, dass bestehende Codebasen viele Klone beinhalten, die deren Kriterien fuer einen guten Klon nicht erfuellen. Wir schlagen eine alternative Erklaerung vor, und zwar, dass Entwickler Cloning als eine Implementierungstechnik sehen, und nicht als Entwicklungspraxis: Sie wuerden Abstraktion bevorzugen, aber finden copy-paste-modify schneller oder einfacher zu verwenden. In einer informellen Umfrage haben wir Hinweise gefunden, die diese These unterstuetzen. Wir schlagen fuer die Reuse-Discrepancy eine neue Loesung vor, die die gleiche Geschwindigkeit und Einfachheit von copy-paste-modify zusammen mit den Entwicklungsvorteilen von Abstraktion bieten, indem ein Refactoring-Algorithmus verwendet wird, um aehnlichen Code halbautomatisch zu verschmelzen (merge). Es kann aber fuer Entwickler wuenschenswert sein Klone einzufuehren, ohne sich bestehender

Abstraktionen bewusst zu sein und Tool-Support zu haben, der die neu eingefuehrten Klone entdeckt und in bestehende Abstraktionen einfuehrt, wenn diese verfuegbar sind. Die Herangehensweise, die verwendet wurde, um Klone in Abstraktionen einzufuehren, reicht nicht aus, um Klone in bestehende Abstraktionen einzufuehren (merge). Wir haben einen Prototypen implementiert. Wir fanden heraus, dass die von unserem Ansatz durchgefhrten Abstraktionen von der Industrie gut angenommen wurden. Ein haeufig sehr nuetzlicher Typ der Softwareanpassung ist eine Migration zwischen Datendarstellungen. Integer zu Bigints umzuwandeln erlaubt es zum Beispiel mit groeeren Werten zu rechnen, die Umwandlung eines Arrays von Strukturen zu einem Strukturen-Array kann Lokalitaet verbessern und Vektorisierung auf Quellcodeebene kann die Kapazitaeten einer Maschine besser nutzen, die vektorisierte Anweisungen bietet. Eine solche Entwicklung kann verschiedene aber zusammenhaengende Transformation ueber ein gesamtes Programm erfordern. Das Problem der Vektorisierung sollte bedacht werden. Vektorisierung ermoeglicht es der Software gleichzeitig verschiedene arithmetische Operationen auf benachbarten Array-Elementen auszufuehren und nicht einzeln auf Skalarwerten. Die Vc API bietet eine Sammlung von Funktionen und Datentypen, die explizite Vektorisierung von C++ Code auf Quellebene erlaubt. Anders als Compilerebene-Vektorisierung, die erfolgreich sein kann, aber nicht immer ist, garantiert die Quellebenen-Vektorisierung eine vektorisierte Ausfuehrung. Die Vc API bietet benutzerdefinierte Typen, wie Float v, das einen Floatvektor repraesentiert und die die mehr als einen Float beinhalten koennen, wobei die exakte Anzahl auf der darunterliegenden Maschine beruht. Vektorisierung mit der Vc API erfordert daher von

Nutzern die Typendeklarationen zu aendern, erfordert Operationen, die die Werte dieser Typen verwenden, benutzerdefinierte Datenstrukturen, die die Ergebnisse dieser Operationen beinhalten, Funktionen, die Argumente erhalten oder als Ergebnis den neuen Typ der Werte zurueckgeben, usw. Es ist muehsam und fehleranfaellig viele solche weit verteilten Veraenderungen schnell durchzufuehren. Wir plaedieren daher fuer die Notwendigkeit von Tool-Support fuer solche nderungen. Um die Anforderungen verschiedener Arten von Datentypen zu erfuellen, sollte ein solches Tool leicht erweiterbar sein. Bestehende Tools fuer automatisierte Programmtransformation koennen eine solche Migration bis zu einem gewissen Grad unterstuetzen: Nutzer koennen eine Spezifikation fuer z. B. Stratego oder Coccinelle schreiben und diese Tools die Spezifikation auf das gesamte Programm anwenden lassen. Bei diesem Prozess wird jedoch das Problem nicht bedacht, dass Nutzer eine Transformation moeglicherweise selektiv angewendet haben moechten, nur auf spezifische Instanzen eines Datentyps, oder aus semantischen oder Performance-Gruenden. Man ueberlege sich z. Bsp. die Migration von int zu einem bigint Typ, um ein arithmetisches Modul anzupassen, damit es groeere Zahlen unterstuetzt. In einem C, C++ oder Java-Programm ist dies keine nderung, die wir automatisch auf alle int Variablen des gesamten Programms angewendet haben moechten. Stattdessen moechten wir typischerweise selbst ein Unterset der integer Variablen im Programm, und deren Nutzung, zur Transformation waehlen koennen. Wir schlagen ein Tool vor, das halbautomatisierte Migrationen von Datendarstellungen unterstuetzt. Unser Tool bietet eine regelspezifische Sprache, vereint mit einer Transformationengine, die die Regeln auf den C oder C++ Code anwendet. Unser Transforma-

tionssystem war gut gerstet, um umfangreiche Transformationen der realen Weltdatendarstellung Migration zu untersttzen.

Krishna Narasimhan

Resume

EDUCATION

Degree	University	Year	Grade
PhD	Goethe University, Frankfurt	2016 (Expected)	
MSc, Computer Science	Saarland University	2013	1.9 ECTS
Master of Computer Application	Anna University	2010	8.8 (on a 1-10 scale)
BSc, Mathematics	Madras University	2007	78%

PhD Thesis
Supervisor : Prof. Dr. Christoph Reichenbach
Title: Combining user interaction and automation to evolve source code

MSc Thesis
Supervisor : Prof. Dr. Andreas Zeller
Title: Android Software decompression chamber

PUBLICATIONS

Conference Proceedings

- Copy and Paste Redeemed, Krishna Narasimhan and Christoph Reichenbach, ASE 2015 ***ACM SIgSoft Distinguished Paper***
- Clone Merge : An Eclipse Plugin to abstract C++ methods, Krishna Narasimhan, ASE 2015 Tools track
- Interactive data representation migration, Krishna Narasimhan, PEPM 2017(to appear)

Symposium/ Report

- Transformation Language for API Migration, Krishna Narasimhan, ECOOP Doctoral Symposium 2015
- Report : Copy and Paste Redeemed, SE 2016, Vienna

EMPLOYMENT

Teaching Assistant, Software Engineering and Programming Languages, September 2014 – March 2015

I, along with the professor designed a novel way of handling exercises which involved a peer reviewed submission model. My responsibilities involved building the system that handled the submission and the reviews through gitolite. The course also involved an experiment with online video lectures and live discussion sessions. My responsibilities also involved helping the professor out with the recordings and updating them online.

Programmer, Ion Sources, GSI April 2014 – March 2015

My responsibilities involved understanding a framework called IBSimu, an ion beam simulator in C++ and helping the physicists in the Ion Sources department of GSI to simulate particle emissions.

Software Engineering Chair, Saarland University Teaching Assistant - Mobile Testing and Analysis, April 2013 – September 2013

I was responsible for the Mobile Testing and Analysis seminar offered in Summer 2013. I had to help students understand advanced research papers on current mobile program analysis , grade the summaries and final presentations on advanced topics in the area

DFKI, Saarbruecken (German Centre of Artificial Intelligence), April 2014 – May 2015

Designed and developed a CDN for huge data (Texture images / 4d cinema videos) in J2EE

Software Engineering Chair, Saarland University Teaching Assistant – Software Engineering, April 2012 – September 2012

I was a tutor for the Software Engineering course offered in Summer 2012, My responsibilities involved managing 3 teams of 6-7 in implementing projects for various clients inside university. Challenges involved both technical and people management

Scantron , Software Development Engineer, June 2010 – July 2011

Developer for the Pinnacle Leraning Management Series product of Scantron, Primary role involved writing code in C#, Javascript and Sql Was also involved in design and architecture because it was a small, closed setup

RESEARCH PROJECTS

Copy and Paste Redeemed

- Novel Approach to abstract near-clone methods. Evaluated with industry standard open source code from repositories involving Google Protobuf, Oracle's Nodedb with positive reviews

Search based Transformation language

- Transformation language with innovative search based application of program transformation rules.

ADMINISTRATION

Program Committee, ECOOP Doctoral Symposium 2016
Student Volunteer, ECOOP 2014
Student Volunteer, ECOOP 2015

SKILLS

Operating System: Linux, Unix, Windows
Administration: Apache Tomcat, JBoss, GlassFish

Programming: Java, Python, C/C++, C#

Scripting: Script#, Javascript

Web Design: HTML, CSS

Versioning System: Git, SVN, Mercurial
Database: MySql, PostGreSql

ACHIEVEMENTS

Microsoft certified Professional in developing Window based Applications using C# (90-316) with a score of 940 out of a 1000 possible.

Zertifikat Deustch(Beginner -Intermediate German) with a score of 272 on a 300 possible

Zentrale mittelstufe Prufung(Intermediate-Advanced German) with a score of 61 on a 100 possible

Among the top 10% Scorers in the Inter level screening test of national Mathematics competition organized by Association of Mathematics teachers of India (2001)

Secured 185/200 in the Grand Achievement test (2003) organized by Association of Mathematics teachers of India