AUTOMATED DUPLICATED-CODE DETECTION AND PROCEDURE EXTRACTION

by

Raghavan V. Komondoor

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2003

© Copyright by Raghavan V. Komondoor 2003

All Rights Reserved

ACKNOWLEDGMENTS

I thank my advisor Susan Horwitz, first and foremost, for giving me the opportunity to work with her, and for making it possible for me to carry out my thesis work. She provided important direction in technical matters, was ever willing to meet for discussions, and always had time to read drafts and comment on them. I appreciate her willingness to give me the independence I needed in my research. Finally, her role in improving my writing and presentation skills is immense.

I thank my committee members, Profs. Charles Fischer, Thomas Reps, Somesh Jha, and Mikko Lipasti for offering useful suggestions to improve the dissertation, and for asking thought-provoking questions. I also thank Prof. Ras Bodik, who was on my Preliminary Exam committee. I thank the staff at the IBM Toronto Lab, in particular Marin Litoiu and Joe Wigglesworth, for providing funding support for several years, and for giving the opportunity to spend summers at the Lab interacting with engineers there. I thank John Field and G. Ramalingam at the IBM T. J. Watson Research Center, and Rajeev Rastogi at Lucent Bell Labs, for providing me the opportunity to work with them on interesting summer projects.

I thank Kai Ting and Qinwei Gong for their role in the implementation of one of our algorithms. I thank my fellow graduate students Alexey Loginov and Suan Yong for providing friendship and for being a part of many fun occasions over my 6 years in graduate school. I thank my friends Amitabh Chaudhary, Sridhar Gopal, K. Jayaprakash, Prateek Kapadia, Srini Krishnamurthy, and Devs Seetharam for being major influences on me, for guiding and supporting me in various ways, for providing light-hearted moments, and for taking part in technical discussions.

I thank my family members for their support, and for always being sure that I was doing the right thing by being in graduate school. I especially thank my father for being an inspiration for me in several ways, and my sister, uncle (of Naperville), and his family, for their love and affection. Finally, I thank my wife Priya, who has changed my life immeasurably. Her confidence in me and her constant support have been among the most important reasons why I could go through graduate school until the very end.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

ABSTRACT x

1 Introduction 1

1.1 Duplication in source code 2
1.2 Motivating example 5
1.3 Steps in clone detection and elimination 7
1.4 Contribution 1: Automated approach for clone detection 9
1.5 Contribution 2: Automated approach for clone-group extraction 10

2 Assumptions, terminology, and background 19

2.1 Program dependence graphs and slicing 22

3 Duplication detection in source code 25

3.1 Finding all pairs of clones 25
3.1.1 Illustration using an example 27
3.1.2 Formal specification of Step 1 of the algorithm 30
3.2 Motivation behind the approach, and its benefits 33
3.2.1 Finding non-contiguous, out-of-order, and intertwined clones 34
3.2.2 Finding good candidates for extraction 36
3.3 Some details concerning the algorithm 39
3.3.1 Step 2 of the algorithm: Eliminating subsumed clones 39
3.3.2 Improving the chances of identifying extractable clones 41
3.3.3 Step 3 of the algorithm: Grouping pairs of clones 44
3.4 Discussion 45
3.4.1 Examples of interesting, extractable clones identified 45
3.4.2 Examples of interesting, not-easily-extractable clones identified 49
3.4.3 Non-one-to-one variable-name mappings 57

Page

3.4.4 A drawback of the approach: variants 60
3.4.5 Other problems with the approach 65
3.4.6 Time complexity of the approach 68

4 Terminology for extraction algorithms 71

5 Individual-clone extraction algorithm 74

5.1 Step 1: find the smallest e-hammock containing the clone 80
5.2 Step 2: generate ordering constraints 83
5.2.1 Data-dependence-based constraints 83
5.2.2 Control-dependence-based constraints 85
5.2.3 Exiting-jumps-based constraints 86
5.3 Step 3: promote unmovable unmarked nodes 89
5.4 Step 4: partition nodes into buckets 91
5.5 Step 5: create output e-hammock $\mathcal{O}$ 94
5.6 Step 6: convert *marked* e-hammock into a hammock 95
5.7 Summary 97
5.8 Complexity of the algorithm 98

6 Clone-group extraction algorithm 102

6.1 Algorithm overview 102
6.1.1 Illustrative example 104
6.1.2 Handling partial matches 105
6.2 Input to the algorithm 110
6.3 Making a set of corresponding maximal block sequences in-order 112
6.3.1 Constraints generation 112
6.3.2 Permuting the block sequences 116
6.4 Complexity of the algorithm 119

7 Experimental results for the clone-detection algorithm 122

7.1 Discussion 126

8 Experimental results for extraction algorithms 129

8.1 Dataset selection 129
8.2 Implementation 131
8.3 Performance of the individual-clone extraction algorithm 132
8.3.1 Techniques used to make clones extractable 134

Appendix

Page

8.3.2 Non-ideal behavior of individual-clone algorithm on some clones . . . 137

8.4 Performance of the clone-group extraction algorithm 138

8.4.1 Failure due to out-of-order issues 140

8.4.2 Failure due to mappings not preserving nesting structure 140

8.4.3 Non-ideal performance of the algorithm 141

8.5 Program size reduction achieved via procedure extraction 141

8.6 Summary 143

9 Related Work 144

9.1 Related work on clone detection 144

9.1.1 A text-based approach 145

9.1.2 AST-based approaches 145

9.1.3 Metrics-based approaches 146

9.1.4 CFG-based approaches 146

9.1.5 Dependence-based approaches 147

9.1.6 Searching for specified patterns 149

9.1.7 Subgraph isomorphism 149

9.2 Previous work related to procedure extraction 150

9.2.1 Exiting jumps 152

9.2.2 Using a range of transformations 153

9.2.3 Comparison of our algorithm with Lakhotia's algorithm 155

9.2.4 Comparison of our algorithm with Debray's algorithm 157

10 Conclusions 161

APPENDICES

Appendix A: The partitioning in the individual-clone algorithm satisfies all constraints 164

Appendix B: The individual-clone algorithm is semantics-preserving 170

Appendix C: The clone-group algorithm is semantics-preserving 187

LIST OF REFERENCES 203

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

1.1 Example illustrating two clones . . . 6

1.2 Output of individual-clone extraction algorithm on fragments in Figure 1.1 . . . 11

1.3 Output of clone-group extraction algorithm on fragments in Figure 1.1 . . . 12

1.4 Example in Figure 1.1 after extraction . . . 14

2.1 CFGs of fragments in Figure 1.1 . . . 21

2.2 An example program, its CFG, and its PDG. The PDG nodes in the backward slice from "`print(k)`" are shown in bold. . . . 23

3.1 Duplicated code from *bison*, with non-contiguous clones . . . 28

3.2 Matching partial slices starting from nodes 3a and 3b. The nodes and edges in the partial slices are shown in bold. . . . 29

3.3 Algorithm to find pairs of clones . . . 31

3.4 Subroutine for growing current clone pair, invoked from Figure 3.3 . . . 32

3.5 Duplicated, out-of-order code from *bison* . . . 34

3.6 An intertwined clone pair from *sort*. . . . 35

3.7 Error-handling code from *tail* that motivates the use of forward slicing. . . . 38

3.8 A slice-pair subsumed by another slice-pair . . . 40

3.9 Example illustrating treatment of loops by algorithm . . . 42

3.10 Illustration of clone-pairs grouping . . . 44

Figure Page

3.11 Two non-contiguous clones identified by the tool in *bison* 47

3.12 Seven copies of this clone were found in *bison* 48

3.13 Two clones in *bison* that are semantically identical but structurally different . . 52

3.14 Two fragments in *bison* that each read a comment from the input grammar . . 53

3.15 Two non-contiguous clones identified by the tool in *bison* 55

3.16 Example clone pair from *make* illustrating non-one-to-one variable renaming . . 58

3.17 Skeleton of a clone pair in *make* that illustrates non-one-to-one variable renaming 59

3.18 Extraction of a difficult-to-extract pair of clones 60

3.19 A clone pair identified by the tool – this pair and the pair in Figure 3.11 are variants of the same ideal pair 63

3.20 A pair of uninteresting clones identified by the tool in *bison* 66

3.21 Example illustrating a group of clones missed by the tool 67

5.1 Transformation done by the individual-clone algorithm 75

5.2 Result ($\mathcal{O}$) of applying individual-clone algorithm on each clone in Figure 2.1. Each dashed oval is the "marked" hammock; the fragments above and below the ovals are the "before" and "after" e-hammocks, respectively. 79

5.3 Algorithm to find the smallest e-hammock that contains the marked nodes . . . 81

5.4 Example illustrating backward exiting jumps 82

5.5 Rules for generating base ordering constraints 84

5.6 Generation of extended ordering constraints 84

5.7 Example illustrating control-dependence-based extended constraints 86

5.8 Example illustrating handling of exiting jumps 87

5.9 Procedure for promoting nodes 90

Appendix
Figure — Page

5.10 Rules for forced assignment of nodes to buckets 92

5.11 Procedure for partitioning nodes into buckets 93

6.1 Example illustrating clone-group extraction 103

6.2 Output of clone-group extraction algorithm on clone group in Figure 6.1 104

6.3 A clone-group with partial matches 105

6.4 Clone-group extraction algorithm. 108

6.5 Output of algorithm on example in Figure 6.3 109

6.6 Procedures for generating control-dependence- and data-dependence-based constraints 113

6.7 Procedure for making a set of corresponding block sequences in-order 114

6.8 Example illustrating control dependence constraints 115

6.9 Constraints graph for the two outermost-level block sequences in the example in Figure 5.2 118

7.1 Example program sizes and running times 123

7.2 Results of running the clone-detection tool 124

7.3 CodeSurfer's decomposition of the predicate "`if(!type_name && typed)`" . . . 127

8.1 Dataset statistics 130

8.2 Comparison of the individual-clone algorithm and ideal extraction 133

8.3 A clone pair in *bison* extracted using predicate-value reuse 135

8.4 A difficult clone pair in the dataset from *make* 137

8.5 Performance of clone-group algorithm in comparison to ideal extraction 139

8.6 Space savings achieved in NARC via procedure extraction 142

9.1 PDG subgraphs illustrating Krinke's approach 148

Figure Page

9.2 Comparison of our algorithm and Lakhotia's algorithm 156

9.3 Comparison of our algorithm to that of Debray et al. 159

Appendix
Figure

C.1 Chains of hammocks 189

ABSTRACT

Making changes to software is a difficult task. Up to 70% of the effort in the software process goes towards maintenance. This is mainly because programs have poor structure (due to poor initial design, or due to repeated ad hoc modifications) which makes them difficult to understand and modify. The focus of this thesis is duplication in source code, which is a major cause of poor structure in real programs. We make two contributions: (a) a novel program-slicing-based approach for detecting duplicated fragments in source code, and (b) a pair of algorithms, one that works on a single selected fragment of code, and the other that works on a group of matching fragments, for making the fragment(s) easily extractable into a separate procedure.

The key, novel aspect of our duplication-detection approach is its ability to detect "difficult" groups of matching fragments, i.e., groups in which matching statements are not in the same order in all fragments, and groups in which non-matching statements intervene between matching statements. Our procedure-extraction algorithms are an advance over previous work in this area in two ways: they employ a range of transformations, including code motion and duplication of predicates, to handle a wide variety of difficult clone groups that arise in practice; and they are the first, to our knowledge, to address the extraction of fragments that contain *exiting jumps* (jumps from within the region containing the fragment to locations outside that are not the "fall through" exit of the region). We present

experimental results, using implementations of our duplication-detection algorithm and one of our extractability algorithms, that indicate that our approaches are effective and useful in practice.

Chapter 1

Introduction

Software maintenance involves changing programs to remove bugs, add new features, adapt to changes in the environment, or improve the structure of the program. Maintenance-related changes are continually made to programs after they are deployed. However, maintenance is a difficult activity, with the effort needed to make a change often being out of proportion to the magnitude of the change. In fact, a study of 487 companies [LS80] found that over 70% of the total effort in the software life cycle went to maintenance activities (as opposed to initial development).

Maintenance is difficult because programs often have poor structure. One aspect of poor structure is lack of *modularity* – the part of the program that is pertinent to the upcoming change is not localized and is not easily locatable. Poor structure is sometimes due to poor initial design or lack of necessary features in the programming language, but is often due to repeated, poorly planned changes made to programs. Belady and Lehman [BL76] hypothesize that under real conditions programmers cause program structure to degrade exponentially with age. They provide supporting evidence for their hypothesis: in a study of the development of the OS/360 system, they found that even though the amount of new code added to the system was roughly the same in each successive release, the time taken per release grew exponentially with the age of the system.

The maintainability of a much-modified program can be improved by periodically *restructuring* the program (improving its structure without affecting its externally observed behavior). Restructuring rolls back the degradation caused by ad-hoc modifications, and

therefore improves future maintainability. However, restructuring itself is a maintenance activity; as such it is time consuming, and it carries the risk of inadvertently changing the program's semantics (behavior). For these reasons, and also because it does not have any immediate benefits, restructuring is rarely done in practice. The study by [LS80], mentioned earlier, found that only about 5% of the time devoted to maintenance is spent in restructuring activities. This motivates the need for developing restructuring tools, which reduce the effort required for restructuring programs, and give a guarantee that the program's behavior is not changed in any way. The focus of our work is on providing tool support for dealing with one of the common sources of poor program structure, duplication in source code.

1.1 Duplication in source code

This thesis presents automatic techniques for:

- detecting duplicated fragments (clones) in source code, and
- eliminating duplication by extracting clones into separate procedures, and replacing them by calls to these procedures.

Programs undergoing ongoing development and maintenance often have a lot of duplicated code, a fact that is verified by several reported studies. For instance, a study by [LPM+97] looked at six releases of a large telecommunications-software system, with each release averaging 14.8 million lines of code. Their finding was that on the average 7% of the functions in a release were exact clones of other functions. Another study [Bak95] found that in the source code of the X Window System (714,479 lines of code), there were 2,487 matching pairs of (contiguous) fragments of length at least 30 lines (the matches were exact, except possibly for one-to-one variable-name substitutions). These matches involved 19% of the entire source code. The same study found that in a production system of over 1.1 million lines 20% of the source code was duplication.

Duplication is usually caused by copy-and-paste activities: a new feature that resembles an existing feature is implemented by copying and pasting code fragments, perhaps followed

by some modifications (such modifications result in the creation of "inexact" copies). Programmers often take this approach because it is quicker and safer, at least in the immediate term, than the other approach of factoring out the code that is common to the old and new features into a separate procedure, and calling this procedure from both places. This is especially true when, due to differences between the existing and new features, it is necessary to restructure the existing code to tease out the part that is needed by the new feature.

Programmers also introduce duplication by reimplementing functionality that already exists in the system, but of which they are not aware. We have located several clones in real programs that have nearly identical functionality, that have many matching lines, but that nevertheless differ in the way the matching lines are ordered. These clones are evidence that duplication can be introduced without copy-and-paste.

Duplication in source code, irrespective of the reason why it exists, is the cause for several difficulties. It increases the size of the code, which can be a problem in domains (such as embedded systems) where space is a limited commodity. It also makes software maintenance more difficult (which is the aspect we focus on). For example, if an enhancement or bug fix is done on a fragment, it may be necessary to search for clones of the fragment to perform the corresponding modification. Therefore, not only is the effort involved in doing a modification increased, but so also is the risk of doing it incorrectly. By modifying some, but not all copies of a fragment, bugs may be removed incompletely, or worse still, new bugs may be introduced (due to the inconsistency introduced between the clones). In fact, the study by Lague et al. [LPM+97] found that whenever a function was modified, the change was usually propagated to some but not all other copies of the function; i.e., it is likely that some propagations that were necessary for correctness were nevertheless missed. Basically, since the copies of a fragment can be related to (be "coupled" with) each other, their presence can be an indication of poor modularity. This, in general, makes maintenance more difficult.

Detecting clones and eliminating them therefore offers several benefits. Elimination works by extracting the cloned code into a separate new procedure, and replacing each clone by a call to this procedure. This reduces the size of the program, and since there will be only one

copy of a clone to maintain, improves maintainability. Furthermore, the fact that the new procedure can be reused may cut down on future duplication.

There are certain situations in which clone detection by itself, without extraction, gives benefits. Extracting a group of clones into a separate procedure can in some cases reduce the understandability of the program; this can happen when the differences between the clones are so significant that a lot of restructuring is needed for extraction, or that the extracted procedure contains a lot of guarded code, or many parameters. Extraction can also be infeasible in situations where the overhead of a procedure call cannot be tolerated, although automated inlining by the compiler, or the use of macros instead of procedures (where available) can offset this disadvantage. Finally, the programming language may have limitations that work against clean extraction. Clone detection alone can be used as a reporting mechanism in these situations, with some benefits: it can be used to find other copies (if any) once an enhancement or bug-fix is made to a fragment, it can be used as an aid for program understanding, and it can be used to guide design decisions if the system is re-engineered or redeveloped from scratch (for some reason). It is noteworthy though that most moderate-to-large-size clones are likely to be extractable into separate procedures. This estimate is made by Baker [Bak95], and is also supported by our own experience (see Chapter 7).

Tool support for both activities – clone detection, and clone elimination via procedure extraction – is desirable. Manual clone detection can be very time consuming; so can manual clone elimination, although to a lesser extent. Due to the size and complexity of software systems, manual clone detection is likely to miss clones. Manual clone extraction, on the other hand, carries the risk of introducing unintended bugs into the program. The study by [LPM+97], which looked at six releases of a large system, provides evidence regarding the usefulness of tool support for clone detection and elimination. The study found that that system's developers detected and eliminated clones (manually) on a regular basis. However, the developers missed many opportunities; in fact, each new release contained more clones

than the previous release, because the number of *new* clones introduced exceeded the number of old clones eliminated.

1.2 Motivating example

Figure 1.1 contains an example (with two code fragments) that we use to illustrate clone detection and elimination. The two clones are indicated using "`++`" signs. The first fragment has a loop that iterates over all employees, while the second fragment has a loop that iterates over employees whose records are present in the file `fe`. Each clone computes the pay for the current employee (indexed by `emp`): if the employee has worked over 40 hours it computes overtime pay using the overtime pay rate available in the array `OvRate`, then picks up the base pay from the array `BasePay`, then checks that the company's policy that overtime pay does not exceed base pay is respected, then computes and stores the total pay into the array `Pay` (the three arrays are global variables, as is the file pointer `fe`). Besides the set of employees that each fragments deals with, the two code fragments have other differences. The first fragment obtains the hours worked from the `Hours` array, whereas the second fragment obtains this information from the `fe` file. Also, the first fragment (alone) counts the number of employees earning overtime pay (via the statement "`nOver++`"), while the second fragment (alone) caps the overtime hours worked at 10.

The two clones in this example have characteristics that make them difficult to detect. Each clone is *non-contiguous* (i.e., has intervening non-matching statements). Also, the matching statements in the two clones are *out-of-order*: the relative positions of the two statements "`if (hours > 40) ..`" and "`base = BasePay[emp]`" are different in the two clones, as are the relative positions of the two statements "`oRate = OvRate[emp]`" and "`excess = hours - 40`".

The clone pair in this example is also difficult to extract. Let us first consider the extractability of each individual clone in the example (into a procedure of its own). Each clone is difficult to extract for two reasons: the clone is non-contiguous, and it involves an *exiting jump* (the `break` statement). A non-contiguous set of statements is difficult to

Original fragment 1	Original fragment 2

<pre>   emp = 0;
    while(emp < nEmps) {
      hours = Hours[emp];
++   overPay = 0;
++   if (hours > 40) {
++   oRate = OvRate[emp];
++   excess = hours - 40;
        nOver++;
++   overPay = excess*oRate;
++   }
++   base = BasePay[emp];
++   if (overPay > base) {
++   error("policy violation");
++   break;
++   }
++   Pay[emp] = base+overPay;
      emp++;
    }</pre>
	<pre> fscanf(fe, "%d", &emp);
	    while(emp != -1) {
	++ base = BasePay[emp];
	++ overPay = 0;
	      fscanf(fe, "%d", &hours);
	++ if (hours > 40) {
	++ excess = hours - 40;
	        if (excess > 10)
	          excess = 10;
	++ oRate = OvRate[emp];
	++ overPay = excess*oRate;
	++ }
	++ if (overPay > base) {
	++ error("policy violation");
	++ break;
	++ }
	++ Pay[emp] = base+overPay;
	      fscanf(fe, "%d", &emp);
	    }</pre>

Figure 1.1 Example illustrating two clones

extract because it is not clear which of the several “holes” that remain after the statements are removed should contain the call to the new procedure. Exiting jumps are jumps within the region that contains the statements in the clone, whose targets are outside the region and are not the “fall-through” exit of the region. A code fragment that involves an exiting jump cannot be extracted as such because, after extraction, control returns from the new procedure to a single statement in the remaining code (the statement that immediately follows the call).

The extractability of both clones into a single separate procedure is complicated by the out-of-order match; to guarantee that the extraction is semantics-preserving the statements in one or both clones must be reordered to provide a single matching sequence of statements for the extracted procedure.

1.3 Steps in clone detection and elimination

Clone detection and elimination involves the following steps:

1. Identify all groups of clones (groups of matching sets of statements) in the program.
2. For each group of clones, check if the group is a good candidate for extraction; if yes, then:
   (a) For each individual clone in the group, if the statements in the clone are non-contiguous and/or involve exiting jumps, apply semantics-preserving transformations to the clone to make its statements form a contiguous, well-structured block of code that contains no exiting jumps. Such a block is *extractable*, i.e., easy to extract into a separate procedure, because it is contiguous and because control flows from it to a unique outside statement.
   (b) If the group of clones involves out-of-order matches, transform one or more of the clones so that matching statements are in the same order in all clones; this makes the group of clones extractable into a single separate procedure.

(c) Extract the new procedure, and replace each (transformed) clone by a call to this procedure.

In this thesis we focus on automatic techniques for Steps 1, 2(a), and 2(b). Providing automated support for Step 1 is important because, as we mentioned earlier, manual clone detection is time consuming, and involves the risk of missing clones. Steps 2(a) and 2(b) are non-trivial to perform, and hence benefit from automation, because they involve semantics-preserving code transformations. These transformations, if done manually, involve the risk of inadvertently introducing errors into the program, and can be time consuming.

We do not address Step 2(c) of clone-group elimination. The main issue in that step is to determine what local variables, "input" parameters, and "output" parameters the procedure needs. This has been addressed in previous clone-group extraction approaches [Zas95, DEMD00]; although those approaches would need to be modified to work on source-level languages.

We envision that the programmer will inspect the clone groups detected automatically in Step 1, decide which ones are good candidates for extraction, and then supply them to the extraction component (Steps 2(a) and 2(b)). Although our proposed approach for Step 1 (introduced below) generally identifies clone groups that are likely to be good candidates for extraction (this is discussed in Section 3.2.2), it is not perfect; the programmer will need to select which clone groups to extract, and furthermore, might need to adjust (modify) the selected clones to make them "ideal" for extraction. We discuss in greater detail the nature of programmer involvement that is required while using the approach, in Sections 3.4.4 and 3.4.5.

It is noteworthy that since there is some subjectivity in this matter, no automatic approach for clone detection is likely to identify exactly the clone groups that the programmer would like to extract. Therefore, although support can be provided to automate the process of duplication elimination significantly, a certain amount of programmer involvement is unavoidable.

1.4 Contribution 1: Automated approach for clone detection

An approach for clone detection in source code, and a tool based on this approach that finds clones in C programs and displays them to the programmer, are key contributions of this thesis. The novel aspect of the approach is the use of *program dependence graphs* (PDGs) [FOW87], and a variation on *program slicing* [Wei84, OO84] to find isomorphic subgraphs of the PDG that represent clones. The key benefits of a slicing-based approach, compared with previous approaches to clone detection that were based on comparing text, control-flow graphs, or abstract-syntax trees, is that the tool can find non-contiguous clones, clones in which matching statements are out of order, and clones that are intertwined with each other (the example in Figure 1.1 illustrates the first two characteristics, while the example in Figure 3.6 illustrates intertwined clones). Such inexactly matching clones are usually introduced when programmers copy code and then modify it so that it suits the new requirements. They can also be introduced when programmers reimplement functionality that already exists in the system, with slight changes.

The other key benefit of the approach is that clones found are likely to be meaningful computations, and thus good candidates for extraction (this is addressed in Section 3.2.2).

We describe the approach in detail in Chapter 3, with a discussion of the strengths and limitations of the approach, and examples of interesting clone groups found in real programs. We then describe the implementation in Chapter 7, and provide the results of experiments with the implementation on real programs. The goals of these experiments were to quantify the efficacy and limitations of the approach.

Our implementation is for C programs. However, our ideas apply to imperative languages in general, or to any language for which PDGs can be built.

1.5 Contribution 2: Automated approach for clone-group extraction

Our second key contribution is a pair of algorithms that support extraction of a group of clones into a separate procedure:

1. The first algorithm, which we call the "individual-clone extraction algorithm", carries out Step 2(a) of the clone-elimination process (see Section 1.3). The input to the algorithm is a clone (a set of statements) that is non-contiguous and/or involves exiting jumps; the algorithm transforms the clone to make it (individually) extractable into a separate procedure. The algorithm handles exiting jumps by converting them into non-exiting jumps. It handles non-contiguity by moving together the set of statements in the clone to the extent possible; any intervening non-clone statements that cannot be moved out of the way are "promoted", which means they are retained in the extracted procedure in guarded form.

2. The second algorithm is for making a group of clones extractable. We call this the "clone-group extraction" algorithm. This algorithm first invokes the individual-clone algorithm on each clone in the group. It then permutes the statements in each clone so that matching statements are in the same order in all clones.

We illustrate the two algorithms by showing the result of applying the clone-group extraction algorithm to the two clones in Figure 1.1. Recall that the first step in the clone-group algorithm is the application of the individual-clone algorithm on each clone in the group; Figure 1.2 shows the result of this step. The resultant code illustrates all four transformation techniques that the individual-clone algorithm incorporates:

1. **Statement reordering:** As many intervening non-clone statements as possible are moved out of the way to make each clone contiguous. In this example, the two non-clone statements "`nOver++`" and "`fscanf(fe,"%d",&hours)`" that intervene between cloned statements in the two clones, respectively, are moved.

Individual-clone algorithm output 1	Individual-clone algorithm output 2

<pre>   if (hours > 40)
        nOver++;
++   overPay = 0;
++   if (hours > 40) {
++   oRate = OvRate[emp];
++   excess = hours - 40;
++   overPay = excess*oRate;
++   }
++   base = BasePay[emp];
++   if (overPay > base) {
++   error("policy violation");
++   exitKind = BREAK;
++   goto L1;
++   }
++   Pay[emp] = base+overPay;
++   exitKind = FALLTHRU;
  L1: if(exitKind == BREAK)
        break;</pre>
	<pre> fscanf(fe,"%d",&hours);
	++ base = BasePay[emp];
	++ overPay = 0;
	++ if (hours > 40) {
	++ excess = hours - 40;
	**** if (excess > 10)
	**** excess = 10;
	++ oRate = OvRate[emp];
	++ overPay = excess*oRate;
	++ }
	++ if (overPay > base) {
	++ error("policy violation");
	++ exitKind = BREAK;
	++ goto L2;
	++ }
	++ Pay[emp] = base+overPay;
	++ exitKind = FALLTHRU;
	  L2: if(exitKind == BREAK)
	        break;</pre>

Figure 1.2 Output of individual-clone extraction algorithm on fragments in Figure 1.1

<table>
<tr><th>Clone-group algorithm output 1</th><th>Clone-group algorithm output 2</th></tr>
<tr><td><pre>
      if (hours > 40)
        nOver++;
++   overPay = 0;
++   if (hours > 40) {
++   excess = hours - 40;
++   oRate = OvRate[emp];
++   overPay = excess*oRate;
++   }
++   base = BasePay[emp];
++   if (overPay > base) {
++   error("policy violation");
++   exitKind = BREAK;
++   goto L1;
++   }
++   Pay[emp] = base+overPay;
++   exitKind = FALLTHRU;
  L1: if(exitKind == BREAK)
        break;
</pre></td><td><pre>
      fscanf(fe,"%d",&hours);
++   overPay = 0;
++   if (hours > 40) {
++   excess = hours - 40;
****   if (excess > 10)
****   excess = 10;
++   oRate = OvRate[emp];
++   overPay = excess*oRate;
++   }
++   base = BasePay[emp];
++   if (overPay > base) {
++   error("policy violation");
++   exitKind = BREAK;
++   goto L2;
++   }
++   Pay[emp] = base+overPay;
++   exitKind = FALLTHRU;
  L2: if(exitKind == BREAK)
        break;
</pre></td></tr>
</table>

Figure 1.3 Output of clone-group extraction algorithm on fragments in Figure 1.1

2. **Predicate duplication:** Moving the statement "`nOver++`" requires creating a duplicate copy of the predicate "`if (hours > 40)`".

3. **Promotion:** The intervening non-clone statement "`if (excess > 10) excess=10`" in the second fragment cannot be moved out of the way without affecting the program's semantics. Therefore it is *promoted* (as indicated by the "`****`" signs), meaning it will occur in the extracted procedure in guarded form (the extracted procedure is shown in Figure 1.4(b)).

4. **Handling exiting jumps:** The `break` statement cannot simply be included in the extracted procedure. Firstly, it is not possible to have a `break` statement without the corresponding loop in a procedure. Furthermore, no exiting jump of any kind can be included in the extracted procedure without any compensatory changes, because, as mentioned earlier, control flows out from a procedure call to a single statement – the statement that follows the call. Therefore, the extracted procedure (Figure 1.4(b)) has a `return` in place of the `break`; it also sets a flag (the new global variable `exitKind`) to indicate whether the `break` must be executed after the procedure returns.

   In the output of the individual-clone algorithm (Figure 1.2) the appropriate assignments to `exitKind` are included, a new copy of the `break`, conditional on `exitKind` is added immediately after the cloned code, and its original copy is converted into a `goto` to the new conditional statement.

   Other exiting jumps (caused by `returns`, `continues` and `gotos`) are handled similarly, with `exitKind` set to a value that encodes the kind of jump.

Notice that the region that originally contained each clone (i.e., everything from the first cloned statement through the last, in Figure 1.1) has been transformed such that in the output of the individual-clone algorithm (Figure 1.2) there is a contiguous block of code that contains the clone and that contains no exiting jumps (this block is indicated by the "`++`" signs in the first fragment, and by the "`++`"/"`****`" signs in the second fragment).

(a) Rewritten Fragment 1

```
emp = 0;
while(emp < nEmps) {
  hours = Hours[emp];
  if (hours > 40)
    nOver++;
  CalcPay(emp,hours,0);
  if(exitKind == BREAK)
    break;
  emp++;
}
```

Rewritten Fragment 2

```
fscanf(fe, "%d", &emp);
while(emp != -1) {
  fscanf(fe,"%d",&hours);
  CalcPay(emp,hours,1);
  if(exitKind == BREAK)
    break;
  fscanf(fe, "%d", &emp);
}
```

(b) Extracted Procedure

```
   void CalcPay(int emp,
    int hours,int doLimit) {
    int overPay,excess,
           oRate,base;

++   overPay = 0;
++   if (hours > 40) {
++     excess = hours - 40;
       if (doLimit)
****     if (excess > 10)
****       excess = 10;
++     oRate = OvRate[emp];
++     overPay = excess*oRate;
++   }
++   base = BasePay[emp];
++   if (overPay > base) {
++     error("policy violation");
++     exitKind = BREAK;
++     return;
++   }
++   Pay[emp] = base+overPay;
++   exitKind = FALLTHRU;
   }
```

Figure 1.4 Example in Figure 1.1 after extraction

This contiguous block of code is easily extractable into a separate procedure. The algorithm does not modify any code outside the region that contains the clone, and therefore that code (which includes the loop header) is not shown in Figure 1.2.

After applying the individual-clone algorithm on each clone, the clone-group algorithm permutes the statements in one or more clones so that matching statements are in the same order in all clones. Figure 1.3 shows the resulting code (which is the final output of the algorithm). Note the differences between the code in Figures 1.2 and 1.3: the statements in the first clone have been permuted so that "`excess = hours - 40`" is before "`oRate = OvRate[emp]`", and the statements in the second clone have been permuted so that "`base = BasePay[emp]`" is after the "`if (hours > 40) ..`" statement. (The algorithm permutes statements only when certain data- and control-dependence-based conditions that are sufficient to guarantee semantics preservation hold; otherwise it fails, with no permutations.)

The new procedure is created, after the algorithm terminates, by basically "merging" copies of the two contiguous blocks produced by the algorithm into a single block: one of the two copies of the cloned code is eliminated, and promoted code from both blocks is retained. The parameters and local variables are determined at this time; promoted statements are surrounded by guards that check boolean flag parameters that are set to *true/false*, as appropriate, at each call to the new procedure; also, the `gotos` produced by the algorithm (from exiting jumps) are converted into `returns`. Then, each of the contiguous blocks produced by the algorithm is simply replaced by a call to the new procedure (with the correct actual parameter values, including the call-site-specific boolean guards). Most of these steps can be automated, although the programmer might want to choose names for the extracted procedure and its parameters/locals.

Figure 1.4 shows the new procedure, as well as the rewritten fragments obtained by replacing the contiguous blocks produced by the algorithm with calls to this procedure. Notice that the intervening non-clone code moved to before the contiguous blocks by the algorithm, as well as the conditional jump code introduced after the blocks, are present adjacent to the calls in the rewritten fragments.

The individual-clone extraction algorithm has applications of its own outside the context of clone-group extraction. One such application is the decomposition of long procedures into multiple smaller procedures. Legacy programs often have long procedures that contain multiple strands of computation (sets of statements), each one achieving a distinct goal. Such strands may occur either one after the other within the procedure, or may be *interleaved* with each other. Interleaved strands occur often in real programs, and complicate program understanding [LS86, RSW96]. Interleaved strands can be separated by applying the individual-clone algorithm to each strand so that it becomes a contiguous extractable block of code; the strands can then be extracted into separate procedures. This improves the program's understandability, eases maintenance by localizing the effects of changes, and facilitates future code reuse. This activity can also be an important part of the process of converting poorly designed, "monolithic" code to modular or object-oriented code. In this thesis, however, we focus on the use of the extraction algorithms in the context of clone-group extraction only.

We define both of the extraction algorithms in the context of the C language. (For the purposes of this dissertation we do not address `switch` statements; the clone-detection tool actually handles `switch` statements, using a variant of the CFG representation proposed in [KH02]; we believe that the extraction algorithms, which currently do not handle `switch` statements, work with little or no modification if the representation proposed in [KH02] is used.) The algorithms are provably semantics preserving (proofs in Appendices B and C). Since semantic equivalence is, in general, undecidable, it is not possible to define an algorithm that succeeds in semantics-preserving procedure extraction whenever that is possible; therefore, the algorithms are based on safe conditions that are sufficient to guarantee semantics preservation.

Our approach is an advance over previous work on procedure extraction in two respects:

- It employs a range of transformations – statement reordering, predicate duplication, guarded extraction, and exiting jump conversion – to handle various kinds of difficult clone groups that arise in practice.

- It is the first to address extraction of fragments that contain exiting jumps into separate procedures.

We discuss both these aspects in greater detail in Chapter 9.

In addition to the extraction algorithms, a (related) contribution of this thesis is a study of 50 groups of clones that we considered worthy of extraction, identified in 3 real programs using the clone-detection tool. The goals of the study were:

- To determine what proportion of the clone groups involved problematic characteristics such as non-contiguity, out-of-order matches, and exiting jumps.

- To determine how well the extraction algorithms performed (on the 50 clone groups) compared with two previous algorithms ([LD98] and [DEMD00]), and compared with the results produced manually (by the author).

We found that nearly 54% of the clone groups exhibited at least one problematic characteristic. We also found that our algorithms produced exactly the same output as the programmer on 70% of the difficult groups, while the previous algorithms matched that "ideal" output on only a small percentage of the difficult groups. Because many of the individual clones in the study were non-contiguous and/or involved exiting jumps, and because some of the clone groups had out-of-order matches, the study measures the performance of both of the extraction algorithms. The study was performed using a partial implementation. The heart of the individual-clone extraction algorithm was implemented; the rest of this algorithm was applied manually, as was the clone-group extraction algorithm.

The rest of this dissertation is organized as follows. Chapter 2 introduces assumptions and terminology for the clone-detection approach, and provides some background on program dependence graphs and slicing. Chapter 3 defines the clone-detection approach, discusses its strengths and weaknesses, and provides examples of clone groups found by the implementation of the approach in real programs. Additional terminology required for the extraction

algorithms is introduced in Chapter 4. We then define the individual-clone extraction algorithm and clone-group extraction algorithms, respectively, in Chapters 5 and 6. We provide experimental results regarding the efficacy and limitations of the clone-detection approach in Chapter 7. Then, in Chapter 8, we discuss our study of the extraction algorithms when applied to the dataset of 51 clone groups obtained using the clone-detection tool. Chapter 9 discusses related work, as well the advances made in this thesis over previous approaches. Finally, Chapter 10 provides directions for future work, as well as the conclusions of this thesis. The appendices contain proofs of correctness (semantics-preservation) for both of the extraction algorithms.

Chapter 2

Assumptions, terminology, and background

We assume that programs are represented by a set of control-flow graphs (CFGs), one per procedure. Each CFG has a distinguished *enter* node as well as a distinguished *exit* node. The other kinds of CFG nodes are: assignment nodes, procedure-call nodes, predicate nodes (`if`, `while`, `do-while`), and jumps (`goto`, `return`, `continue`, `break`). Each assignment statement in the source code is represented by one assignment node in the CFG; the same is true for predicates and jumps. Procedure calls that represent values (i.e., function calls) are not given separate nodes; rather they are regarded as part of the node that represents the surrounding expression. Other procedure calls (ones that return no value or ones whose values or not used) are represented using separate nodes. Labels are not included in the CFG, and are implicitly represented by an edge from a `goto` node to its target.

A CFG's exit node has no outgoing edge; predicate nodes have two outgoing edges, labeled *true* and *false*; assignments and procedure-call nodes have a single outgoing edge. Jump nodes are considered to be pseudo-predicates (predicates that always evaluate to *true*), as in [BH93, CF94]. Therefore, each jump is represented by a node with two outgoing edges: the *true* edge goes to the target of the jump, and the (non-executable) *false* edge goes to the node that would follow the jump if it were replaced by a no-op. Jumps are treated as pseudo-predicates so that the statements that are semantically dependent on a jump – as defined in [KH02] – are also *control dependent* on it (control dependence is defined later in this section). *True* edges out of jump nodes are called *jump edges*; every other CFG edge is called a *non-jump* edge. Every node in a CFG lies on some path from the *enter* node to

the *exit* node. For technical reasons the *enter* node is also treated as a pseudo-predicate; its *true* edge goes to the first actual node in the CFG, while its (non-executable) *false* edge goes straight to the *exit* node.

Example: Figure 2.1 contains the CFGs (CFG subgraphs, actually) for the fragments in Figure 1.1. Labels on edges out of the predicate nodes have been omitted for the sake of clarity. Note that the `breaks` are pseudo-predicates, with two outgoing edges; the non-executable *false* edges are shown dashed. □

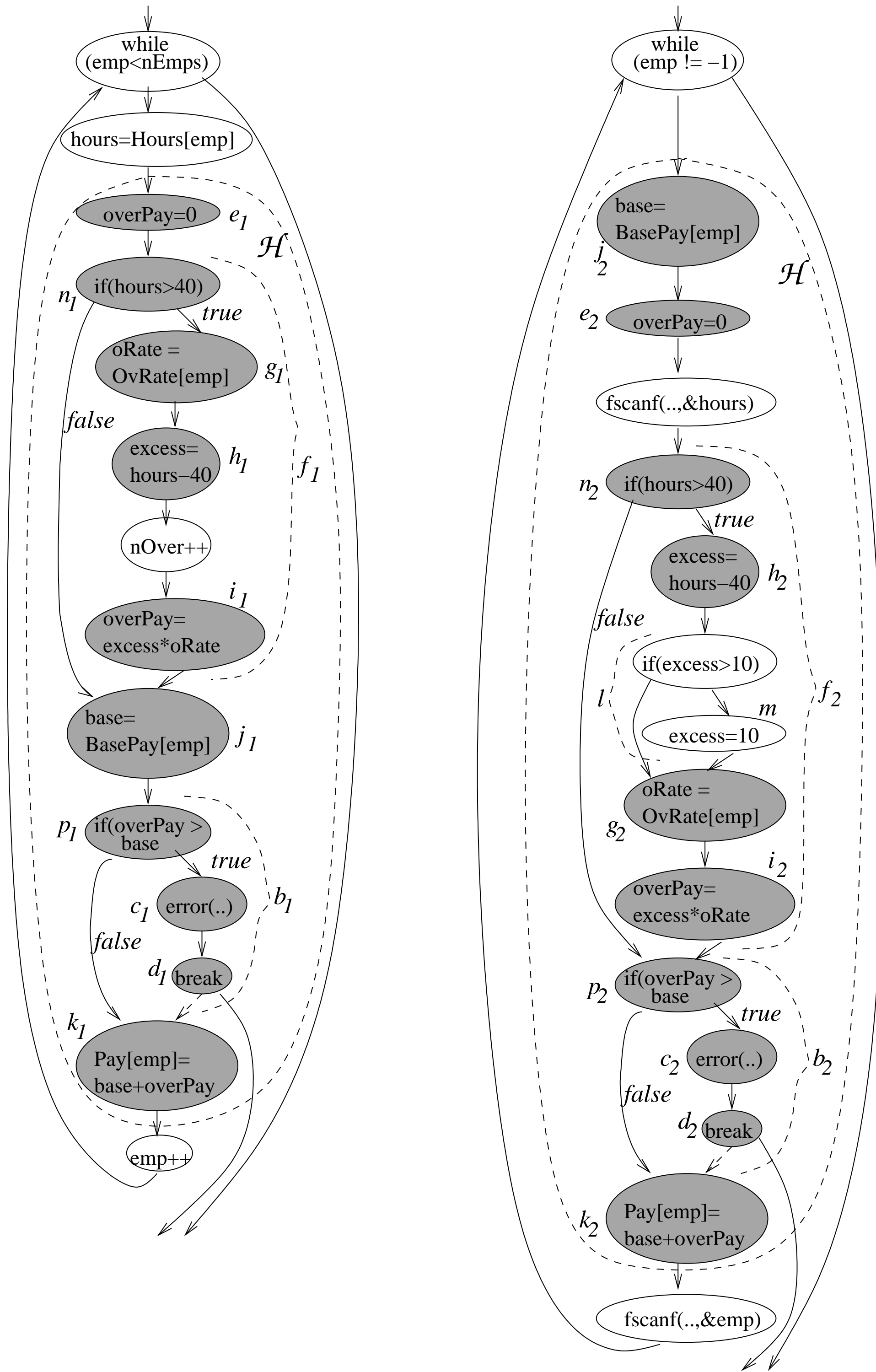
We allow the input programs to make use of features such as pointers, address-of operators, structures, and global variables. We do assume, however, that the appropriate static analyses, e.g., pointer analysis and inter-procedural GMOD/GREF analysis, have been done so that *use* and *def* sets (an over-approximation of the set of variables whose values may be used, and defined, respectively) are known for each CFG node. In particular, if a node in a procedure includes a procedure call or a dereference of a pointer, then the *use* and/or *def* sets of that node can include variables that do not occur literally in the node, and/or non-local variables. We assume that predicates have no side-effects (so every predicate node has an empty *def* set); this is not a severe restriction, because predicates that do have side-effects can be decomposed into one or more assignments/procedure calls, followed by a "pure" predicate.

We make use of the following (standard) definitions:

Definition 1 (Postdomination) A node p in a CFG postdominates a node q in the same CFG if every path from q to the exit node, including ones that involve non-executable edges, goes through p. Every node postdominates itself, by definition.

Definition 2 (Control dependence) A node p is C-control dependent on node q, where C is either *true* or *false*, iff q is a predicate node and p postdominates the C-successor of q but does *not* postdominate q itself. q is said to be a *control-dependence parent* of p, and p is said to be a *control-dependence child* of q.

The following definition is adapted from [KKP+81].



Clones are indicated using shaded nodes. Dashed ovals indicate the e-hammocks of the clones.

Figure 2.1 CFGs of fragments in Figure 1.1

Definition 3 (Flow dependence) A node p is *flow dependent* on a node q iff some variable v is defined by q, and used by p, and there is a v-*def*-free path P in the CFG from q to p that involves no non-executable edges[1]. We say that this flow dependence is *induced by* path P.

2.1 Program dependence graphs and slicing

Program dependence graphs (PDGs) were proposed in [FOW87] as a convenient program representation for several program analyses and transformations. The PDG for a procedure includes all of the nodes in the procedure's CFG, except for the *exit* node. The edges in the PDG represent the control dependences and flow dependences computed using the CFG; i.e., there is a control- (flow-) dependence edge from a node q to a node p in a PDG iff p is control- (flow-) dependent on p. As in the CFG, control-dependence edges are labeled *true* or *false*.

Example: Figure 2.2 shows an example program, its CFG, and its PDG. Ignore, for now, the distinction between bold and non-bold nodes in the PDG. Labels on control-dependence edges in the PDG are omitted for the sake of clarity (every such edge in this example is labeled `true`). (This example is taken directly from [KH02].) □

Our clone-detection approach makes use of PDGs, and a variation of an operation called *slicing*. Slicing was originally defined by Weiser [Wei84]. Informally, the *backward slice* of a procedure from a node S is the set of nodes in that procedure that might affect the execution of S, either by affecting some value used at S, or by affecting whether and/or how often S executes.

Weiser provided a CFG-based, dataflow-analysis-based algorithm for computing the backward slice from a node S in a procedure. Ottenstein and Ottenstein [OO84] provided a

[1] the original definition in [KKP+81] does not involve non-executable edges, because their CFGs have only executable edges

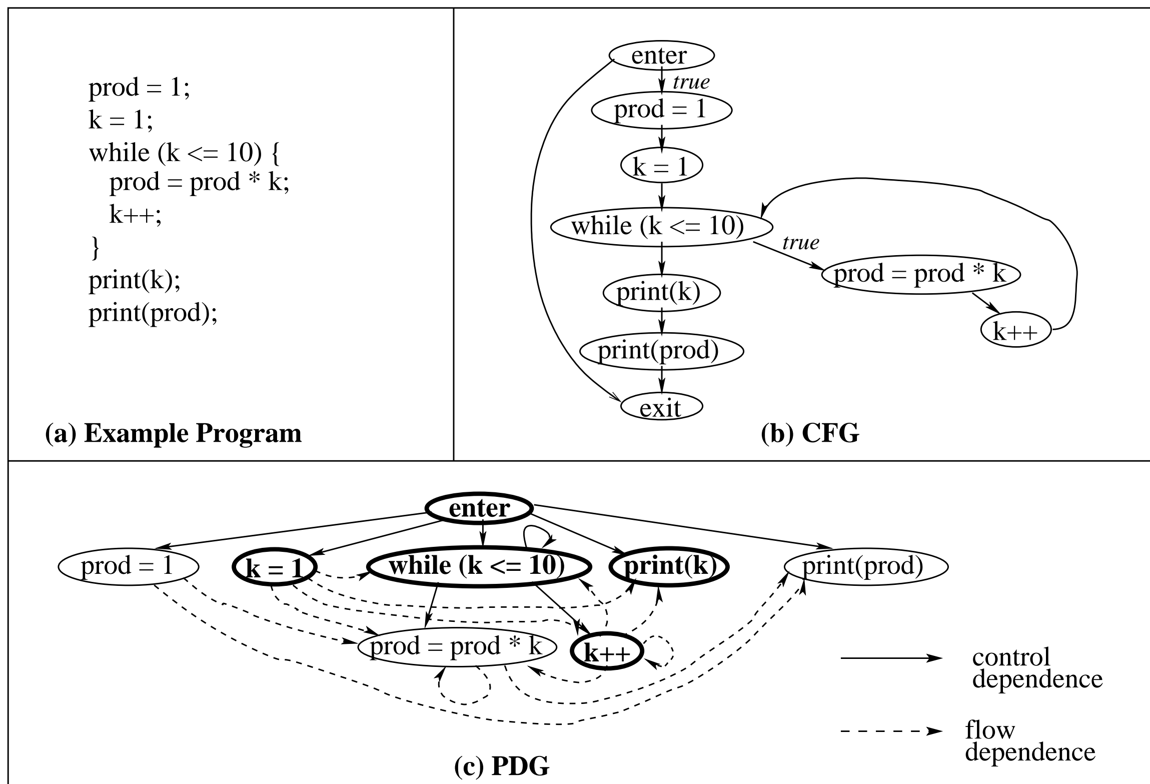


Figure 2.2 An example program, its CFG, and its PDG. The PDG nodes in the backward slice from "`print(k)`" are shown in bold.

more efficient algorithm that uses the PDG: Start from S and follow the control- and data-dependence edges backwards in the PDG. The nodes in the slice are all the nodes reached in this manner.

Example: The bold nodes in the PDG in Figure 2.2 constitute the backward slice from "`print(k)`". □

Analogous to a backward slice, the *forward slice* of a procedure from a node S is the set of nodes in the procedure that might be affected by S. The forward slice from S can be computed as the set of nodes reachable from S by following edges forward in the PDG.

Chapter 3

Duplication detection in source code

This chapter describes our clone-group detection algorithm. The algorithm performs three steps:

Step 1. Find all pairs of clones.

Step 2. Remove clone pairs that are *subsumed* by other clone pairs.

Step 3. Combine pairs of clones into larger groups.

This chapter is organized as follows. Section 3.1 introduces Step 1 of the algorithm – the slicing-based approach to finding pairs of clones. This step is the heart of the algorithm. The motivation behind the slicing-based approach, and the benefits of this approach, are provided in Section 3.2. Section 3.3 provides certain details about the clone-pairs detection approach that are omitted from Section 3.1; it also specifies Steps 2 and 3 of the algorithm. Section 3.4 provides examples of interesting clone groups found by the approach, and discusses some of the limitations of the approach.

3.1 Finding all pairs of clones

We first describe Step 1 of the algorithm informally, together with an illustration using an example. We then provide the formal description of this step in Section 3.1.2.

A high-level outline of Step 1 is:

1. Partition all nodes in all PDGs into equivalence classes, such that two nodes are in the same class if and only if they *match* (as defined below).

2. For each pair of matching nodes `(root1, root2)`, find two matching subgraphs of the PDGs that contain `root1` and `root2`, such that the subgraphs are "rooted" at `root1` and `root2`, using a variation of the slicing operation. The pair of subgraphs found is a pair of clones.

The notion of matching nodes is defined (recursively) as follows:

- Two expressions match if they have the same syntactic structure, ignoring variable names and literal values; e.g., "`b + 1`" matches "`d + 2`", but does not match "`d - 2`" or "`2 + d`". Array references match other array references iff the subscript expressions match.

  Because variable names are ignored while matching expressions, variable names in a clone may not map one-to-one with variable names in other corresponding clones; e.g., the node "`a + a + b`" matches "`p + q + q`", with `q` mapped both to `a` and to `b`, and with `a` mapped both to `p` and to `q`. We consider the implications of this in Section 3.4.3.

- Two function calls within expressions (or two procedure-call nodes) match if and only if both are calls to the same function, and corresponding actual parameters match (as expressions); e.g., "`f(a+1, b())`" matches "`f(c+2, b())`", but does not match "`f(c+2, d())`" or "`f(c-2, b())`".

- Two assignments match if and only if the left hand sides, as well as the right hand sides, match (as expressions).

- Two predicates match if and only if their expressions match, and both are of the same *kind* (`while, do-while, if`).

- Two jumps match if and only if they are of the same kind (`return`, `goto`, `break`, `continue`). For `return`s, their expressions must match, too.

The heart of the algorithm that finds two matching subgraphs is the use of backward slicing: Starting from `root1` and `root2` we slice backwards in lock step, adding a (flow-

or control-dependence) predecessor (and the connecting edge) to one slice iff there is a corresponding, matching predecessor in the other PDG (which is added to the other slice). The two predecessors just added to the slice-pair are said to be *mapped* to each other, as are the two connecting edges. Forward slicing is also used: whenever a pair of matching loop or `if` predicates $(p1, p2)$ is added to the pair of slices, we slice forward one step from $p1$ and $p2$, adding their matching control-dependence successors (and the connecting edges) to the two slices. Here again, the successors (connecting edges) just added to the slice-pair are said to be mapped to each other. Note that while lock-step backward slicing is done from *every* pair of matching nodes in the two slices, forward slicing is done only from matching predicates. When the process described above finishes, it will have identified two matching "partial" slices (PDG subgraphs) that represent a pair of clones. (Our motivation for using backward slicing and forward slicing is given, respectively, in Sections 3.2.2.2 and 3.2.2.3.)

3.1.1 Illustration using an example

Figure 3.1 shows a group of four clones (indicated by the "`++`" signs) identified by the implementation of the approach, on the source code of the Unix utility *bison*. The function of the duplicated code is to grow the buffer pointed to by `p` if needed, append the current character `c` to the buffer and then read the next character. The PDGs for Fragments 1 and 2 in Figure 3.1 are shown in Figure 3.2. We illustrate the process of finding a pair of matching partial slices, starting from matching nodes $3a$ and $3b$ in the PDGs. Slicing backward from nodes $3a$ and $3b$ along their incoming control-dependence edges we find nodes 5 and 8 (the two `while` nodes). However, these nodes do not match (they have different syntactic structure), so they are not added to the partial slices. Slicing backward from nodes $3a$ and $3b$ along their incoming flow-dependence edges we find nodes $2a$, $3a$, $4a$, and 7 in the first PDG, and nodes $2b$, $3b$, and $4b$ in the second PDG. Node $2a$ matches $2b$, and node $4a$ matches $4b$, so those nodes (and the edges just traversed to reach them) are added to the two partial slices. (Nodes $3a$ and $3b$ have already been added, so those nodes are not reconsidered.) Slicing backward from nodes $2a$ and $2b$, we find nodes $1a$ and $1b$, which match, so they (and the

Fragment 1

```
   while (isalpha(c) ||
          c == ’_’ || c == ’-’) {
++  if (p == token_buffer+maxtoken)
++   p = grow_token_buffer(p);
    if (c == ’-’) c = ’_’;
++  *p++ = c;
++  c = getc(finput);
   }
```

Fragment 2

```
   while (isdigit(c)) {
++  if (p == token_buffer+maxtoken)
++   p = grow_token_buffer(p);
++  *p++ = c;
    numval = numval*10 + c - ’0’;
++  c = getc(finput);
   }
```

Fragment 3

```
   while (c != ’>’)  {
    if (c == EOF) fatal();
    if (c == ’\n’) {
     warn("unterminated type name");
     ungetc(c, finput);
     break;
    }
++  if (p == token_buffer+maxtoken)
++   p = grow_token_buffer(p);
++  *p++ = c;
++  c = getc(finput);
   }
```

Fragment 4

```
   while (isalnum(c) ||
          c == ’_’ || c == ’.’) {
++  if (p == token_buffer+maxtoken)
++   p = grow_token_buffer(p);
++  *p++ = c;
++  c = getc(finput);
   }
```

Figure 3.1 Duplicated code from *bison*, with non-contiguous clones

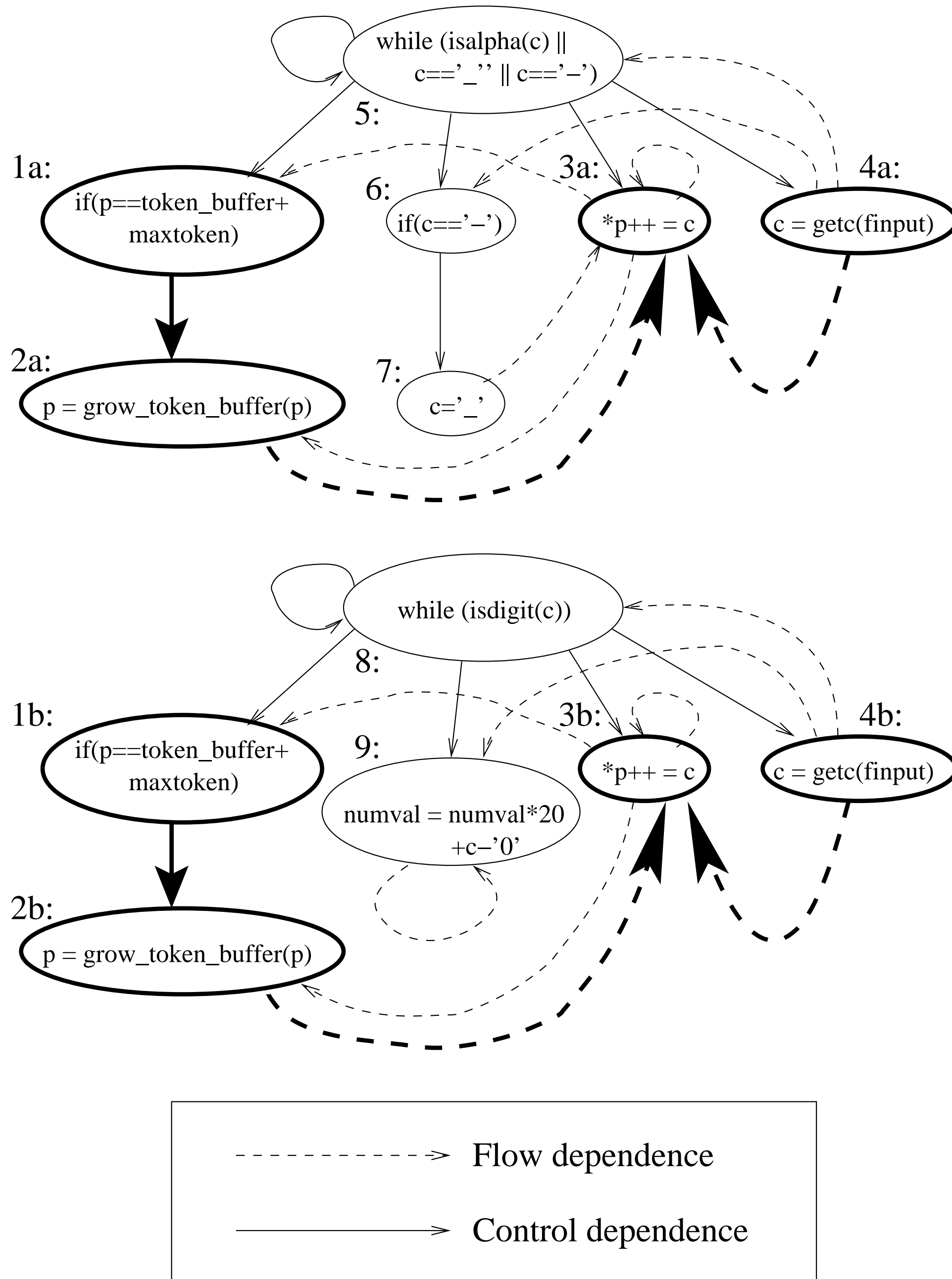


Figure 3.2 Matching partial slices starting from nodes 3a and 3b. The nodes and edges in the partial slices are shown in bold.

traversed edges) are added. Furthermore, nodes 1a and 1b represent `if` predicates; therefore we slice forward from those two nodes. We find nodes 2a and 2b, which are already in the slices, so they are not reconsidered. Slicing backward from nodes 4a and 4b, we find nodes 5 and 8, which do not match; the same two nodes are found when slicing backward from nodes 1a and 1b.

The partial slices are now complete. The nodes and edges in the two partial slices are shown in Figure 3.2 using bold font. These two partial slices correspond to the clones of Fragments 1 and 2 shown in Figure 1.1 using "++" signs.

3.1.2 Formal specification of Step 1 of the algorithm

Figures 3.3 and 3.4 specify the approach for finding clone pairs. Procedure `FindAllClonePairs` in Figure 3.3 is the "main" function. It partitions the set of all nodes in all PDGs into equivalence classes, and then invokes Subroutine `FindAClonePair` on *each* pair of matching nodes to find the two matching partial slices rooted at that pair. Actually, Procedure `FindAllClonePairs` incorporates some optimizations. Thus the above description is not entirely accurate; we postpone discussion of this matter to Section 3.3.

Procedure `FindAClonePair` is the one that finds a matching partial slice pair from a pair of matching starting nodes. This procedure maintains two data structures, `worklist` and `curr` (we discuss the other data structure, `globalHistory`, in Section 3.3). `curr` is the set of currently mapped pairs of PDG edges; therefore, when `FindAClonePair` finishes, `curr` contains the entire clone pair (the mapped edges also tell us which nodes are mapped). The pair of roots is the first node pair to be included in the slice pair; i.e., this pair is unlike all other matching node pairs, which are reached along PDG edges from other matching node pairs. To accommodate this exception we initialize `curr` (in the beginning of Procedure `FindAClonePair`) with the edge pair (`root1` $\rightarrow \diamond$, `root2` $\rightarrow \diamond$), where the $\diamond$ is some dummy node.

`worklist` is a set of pairs of nodes that have already been mapped and included in the clone pair, but from which we are yet to slice backward (or forward, for mapped predicates).

Procedure `FindAllClonePairs`.

1: For each PDG `p` in the program create `p.list`, the list of all nodes in `p` sorted in the reverse of the order in which nodes are visited in a depth-first traversal of `p` starting from its *entry* node.
2: Partition the set of all nodes in the program (i.e., in all PDGs) into *equivalence classes*, such that *matching* nodes are in the same class. Represent each class as a list, and sort the list such that the relative ordering of nodes from a single PDG `p` within the list is the same as their ordering in `p.list`.
3: Initialize `globalHistory` to empty.
4: **for all** PDGs `p` in the program **do**
5:     **for all** nodes `root1` in `p.list` **do**
6:         **for all** nodes `root2` in `root1`'s equivalence class (a list), not including `root1` and not including nodes following `root1` in the list **do**
7:             **if** `(root1, root2)` is not present in `globalHistory` **then**
8:                 Call `FindAClonePair (root1, root2)`.
9:             **end if**
10:         **end for**
11:     **end for**
12: **end for**

Procedure `FindAClonePair (root1, root2)`.

1: Initialize `curr` and `worklist` to empty.
2: Place the pair of edges (`root1` $\rightarrow \diamond$, `root2` $\rightarrow \diamond$) in `curr`. {*That is, map these two edges to each other and add them to the current clone pair.*} Place (`root1, root2`) in `worklist`, and in `globalHistory`.
3: **repeat**
4:     Call `GrowCurrentClonePair`.
5: **until** `worklist` is empty.
6: Write `curr` (the current clone pair) to output.

Figure 3.3 Algorithm to find pairs of clones

Procedure `GrowCurrentClonePair`.

1: Remove a node pair `(node1, node2)` from `worklist`.
{*Map flow-dependence parents of* `node1` *to flow-dependence parents of* `node2`*, as follows.*}
2: **for all** flow-dependence parents `p1` of `node1` in the PDG that are not an end-point of any edge in `curr` **do**
3: **if** there exists a flow-dependence parent `p2` of `node2` such that:
• `p2` is in the same equivalence class as `p1`, and `p2` is not an end-point of any edge in `curr`, and
• the two flow-dependence edges `p1` → `node1` and `p2` → `node2` are either both *loop carried*, or are both *loop independent* (defined in Section 3.3.2.1), and
• the predicates of the loops *crossed* by the flow-dependence edge `p1` → `node1` are in the same equivalence class as the corresponding predicates of the loops *crossed* by the flow-dependence edge `p2` → `node2`
then
4: Place (`p1` → `node1`, `p2` → `node2`) in `curr`. {*That is, map the two edges to each other and add them to the current clone pair.*} Place (`p1, p2`) in `worklist`, and in `globalHistory`.
5: **end if**
6: **end for**
{*Map control-dependence parents of* `node1` *to control-dependence parents of* `node2`*, as follows.*}
7: **for all** control-dependence parents `p1` of `node1` in the PDG that are not an end-point of any edge in `curr` **do**
8: **if** there exists a control-dependence parent `p2` of `node2` such that:
• `p2` is in the same equivalence class as `p1`, and `p2` is not an end-point of any edge in `curr`, and
• the two control-dependence edges `p1` → `node1` and `p2` → `node2` have the same label (*true* or *false*)
then
9: Place (`p1` → `node1`, `p2` → `node2`) in `curr`. Place (`p1, p2`) in `worklist`, and in `globalHistory`.
10: **end if**
11: **end for**
12: **if** `node1` and `node2` are predicates **then**
13: {*Map control-dependence children of* `node1` *to control-dependence children of* `node2`*, as follows.*}
14: **for all** control-dependence children `c1` of `node1` in the PDG that are not an end-point of any edge in `curr` **do**
15: **if** there exists a control-dependence child `c2` of `node2` such that:
• `c2` is in the same equivalence class as `c1`, and `c2` is not an end-point of any edge in `curr`, and
• the two control-dependence edges `node1` → `c1` and `node2` → `c2` have the same label (*true* or *false*).
then
16: Place (`node1` → `c1`, `node2` → `c2`) in `curr`. Place (`c1, c2`) in `worklist`, and in `globalHistory`.
17: **end if**
18: **end for**
19: **end if**

Figure 3.4 Subroutine for growing current clone pair, invoked from Figure 3.3

`worklist` is initialized to (`root1, root2`), and `FindAClonePair` finishes when `worklist` becomes empty.

`FindAClonePair` calls a subroutine `GrowCurrentClonePair`, shown in Figure 3.4. `GrowCurrentClonePair` removes a node-pair (`node1, node2`) from `worklist`, and maps matching flow-dependence and control-dependence predecessors of these two nodes, as well as matching control-dependence successors, if `node1` and `node2` are predicates. Note that for two flow-dependence predecessors to be mapped to each other, we have two restrictions, one to do with loop-carried and loop-independent edges, and the other to do with predicates of the loops *crossed* by the two edges. We discuss these two restrictions in Section 3.3. Note that in general there may be multiple ways of mapping predecessors (or successors) of (`node1, node2`) to each other; when there are several choices the algorithm chooses one among them, while ensuring that the maximum possible number of predecessors (successors) are mapped to each other.

3.2 Motivation behind the approach, and its benefits

Previous approaches for clone detection either work on the source text of the program, or on the abstract syntax tree (AST) representation, or on the control-flow graph (CFG) representation. Source code is a linear representation, which forces the programmer to arbitrarily pick one particular ordering of statements from among several potential choices that are all semantically equivalent. The other two representations are closely tied to the source; in particular, both representations reflect the ordering of statements in the source code (although they do abstract away certain lexical aspects of the source). As a result, these previous approaches cannot find duplicated fragments that match inexactly at the source-code level, but where the inexactness is purely an artifact of differing arbitrary choices the programmer made for the different copies. The key hypothesis we make is that if a group of fragments have similar functionality (and match syntactically at the statement level), then flow- and control-dependences between statements are identical (or very similar) in all the fragments, even if the fragments match inexactly considering the ordering of statements.

Fragment 1	Fragment 2

<pre>   fp3 = lookaheadset + tokensetsize;
   for (i=lookaheads[state];
        i < k; i++) {
++   fp1 = LA + i*tokensetsize;
++   fp2 = lookaheadset;
++   while (fp2 < fp3)
++   *fp2++ |= *fp1++;
   }</pre>
	<pre> fp3 = base + tokensetsize;
	   ...
	   if (rp) {
	    while ((j = *rp++) >= 0) {
	      ...
	++ fp1 = base;
	++ fp2 = F + j*tokensetsize;
	++ while (fp1 < fp3)
	++ *fp1++ |= *fp2++;
	    }
	   }</pre>

Figure 3.5 Duplicated, out-of-order code from *bison*

This hypothesis forms the basis for our approach: use PDGs, which abstract away irrelevant aspects of the ordering among statements and reflect only flow- and control-dependences, and find clones by finding matching subgraphs in the PDGs. As stated in the Introduction, the approach has two major benefits – finding inexact matches, and finding clone groups that are good candidates for extraction. We discuss these benefits in the following subsections.

3.2.1 Finding non-contiguous, out-of-order, and intertwined clones

One example of non-contiguous clones identified by the implementation of the approach in the source code of *bison* was given in Figure 3.1. By running the tool on a set of real programs, we have observed that non-contiguous clones that are good candidates for extraction (like the ones in Figure 3.1) occur frequently (see Section 8 for further discussion). Therefore, the fact that the approach can find such clones is a significant advantage over most previous approaches to clone detection.

Non-contiguous clones are one kind of inexact matching. Another kind of inexact matching occurs when the ordering of matching statements is different in the different clones. The two clones shown in Figure 3.5 (again from *bison*) illustrate this. The clone in Fragment 2

```
++ tmpa = UCHAR(*a),
xx tmpb = UCHAR(*b);
++ while (blanks[tmpa])
++   tmpa = UCHAR(*++a);
xx while (blanks[tmpb])
xx   tmpb = UCHAR(*++b);
++ if (tmpa == '-') {
     tmpa = UCHAR(*++a);
     ...
   }
xx else if (tmpb == '-') {
     if (...UCHAR(*++b)...) ...
```

Figure 3.6 An intertwined clone pair from *sort*.

differs from the one in Fragment 1 in two ways: the variables have been renamed (including renaming `fp1` to `fp2` and vice versa), and the order of the first and second statements (in the clones, not in the fragments) has been reversed. This renaming and reordering does not affect the flow or control dependences; therefore, the approach finds the clones as shown in the figure, with the first and second statements in Fragment 1 that are marked with "++" signs matching the second and first statements in Fragment 2 that are marked with "++" signs.

Our approach is also effective in finding intertwined clones. An example of such clones in the Unix utility *sort* is given in Figure 3.6. In this example, one clone is indicated by "++" signs while the other clone is indicated by "xx" signs. The clones take a character pointer (`a`/`b`) and advance the pointer past all blank characters, also setting a temporary variable (`tmpa/tmpb`) to point to the first non-blank character. The final component of each clone is an `if` predicate that uses the temporary. The predicates were the roots of the two matching partial slices (the second one – the second-to-last line of code in the figure – occurs 43 lines further down in the code).

3.2.2 Finding good candidates for extraction

A key goal of ours, which has driven several design decisions, is to find groups of clones that can be extracted into separate procedures. A group of clones is a good candidate for extraction if:

- The group is *extractable*; i.e., it is possible to create a separate procedure and to replace each clone with a call to this procedure, such that the semantics of the program is preserved.
- The group would be considered interesting by a programmer.
- The clones are not too small.

Intuitively, a group of clones is interesting if:

- Each clone in the group performs a meaningful computation that makes sense as a separate procedure; i.e., the functionality of the clone can be easily explained in English, or equivalently, the new procedure obtained from the clone can be given a meaningful name.
- The clones in the group all have similar functionality; i.e., the English explanations of the functionalities of the clones are similar.

An example of a pair of interesting clones is the one shown in Figure 1.1. The functionality of the cloned code in that example can be explained as follows: Compute the base pay and overtime pay of an employee; then, if overtime pay does not exceed base pay compute the total pay, else report an error. Another example of an interesting clone pair is the one shown in Figure 3.1. As we mentioned earlier, the function of the cloned code in that example is to grow the buffer pointed to by `p` if needed, append the current character `c` to the buffer and then read the next character. Sections 3.4.1 and 3.4.2 present several other examples of interesting clone groups found in real programs by the implementation of the approach

(any clone group that does not satisfy the informal characterization presented above is called "uninteresting").

One aspect of meaningfulness is that the cloned code performs a single conceptual operation (is highly cohesive [SMC74]):

- it computes a small number of outputs (every variable for which there is a flow-dependence edge from inside the clone to outside is an output of the clone).
- all the cloned code is relevant to the computation of the outputs (i.e., the backward slices from the statements that assign to the outputs should include the entire clone).

Another aspect of meaningfulness is that the cloned code represent a "logically complete" computation (we return to this later).

We now discuss how the characteristics required of good clone groups are likely to be satisfied by the clone pairs found by the algorithm.

3.2.2.1 Similar functionality due to matching dependences

Two clones are likely to have similar functionality if for every flow- (control-) dependence edge between two nodes in one of the clones, there is a flow- (control-) dependence edge between the matching two nodes in the other clone. Although the way we construct matching partial slice pairs does not entirely guarantee this property, because our (sole) mechanism for growing a slice pair is adding matching pairs of dependence edges to it, many dependence edges within an identified clone have matching dependence edges in the other clone. This makes it likely that clone pairs identified have similar functionality.

3.2.2.2 Cohesion due to backward slicing

The heart of the algorithm is backward slicing. A backward slice from a starting node automatically satisfies one of the two aspects of cohesiveness – every statement in the slice is relevant to the outputs computed at the starting node. Therefore, a backward slice is likely

Fragment 1	Fragment 2

`if (tmp->nbytes == -1) {`
` error (0, errno, "%s", filename);`
` errors = 1;`
` free((char *) tmp);`
` goto free_lbuffers;`
`}`
	`if (tmp->nbytes == -1) {`
	` error (0, errno, "%s", filename);`
	` errors = 1;`
	` free((char *) tmp);`
	` goto free_cbuffers;`
	`}`

Figure 3.7 Error-handling code from *tail* that motivates the use of forward slicing.

to be cohesive. For the same reason, when a pair of matching partial slices is obtained by lock-step backward slicing, both partial slices are likely to be cohesive.

3.2.2.3 Complete computations due to forward slicing

In many situations, backward slicing by itself only identifies clones that are subsets of "logically complete" clones that would make sense as a separate procedure. In particular, conditionals and loops sometimes contain code that forms one logical operation, but that is not the result of a backward slice from any single node.

One example of this situation is error-handling code, such as the two fragments in Figure 3.7 from the Unix utility *tail*. The two fragments are identical except for the target of the final `goto`, and are reasonable candidates for extraction. They both check for the same error condition, and if it holds, they both perform the same sequence of actions: calling the `error` procedure, setting the global `errors` variable, and freeing variable `tmp`. Each of the two fragments is a "logically complete" computation, as the entire sequence of actions is conditional on, and related to, the controlling condition. (The final `goto` cannot be part of the extracted procedure; instead, that procedure would need to return a boolean value to specify whether or not the `goto` should be executed. This is described in detail in Section 5.6.)

Note that the two fragments cannot be identified as clones using only backward slicing, since the backward slice from any statement inside the `if` fails to include any of the other

statements in the `if`. Thus, with backward slicing only, we would identify four clone pairs – each one containing one of the pairs of matching statements inside the `if` statements, plus the `if` predicates. The forward-slicing step from the pair of matched `if` predicates allows us to identify just a single clone pair – the two entire `if` statements, which are logically complete computations.

Another example where forward slicing is needed is a loop that sets the values of two related but distinct variables (e.g., the head and tail pointers of a linked list). In such examples, although the entire loop corresponds to a single logical operation, backward slicing alone is not sufficient to identify the whole loop as a clone.

Note that we do forward slicing only in a restricted manner. While we do backward slicing from every pair of mapped nodes in the clone pair being currently built, along flow- and control-dependence edges, we do forward slicing only from mapped predicates along control-dependence edges. Forward slicing along control-dependence edges makes sense for the reason mentioned above: we want to find groups of statements that form logically atomic units because of control dependence on a common condition. However, in our experience, forward slicing along flow-dependence edges gives bad results: many separate computations (each with its own outputs) can be flow dependent on an assignment statement, and including them all in the clone pair (by forward slicing from the assignment) destroys cohesiveness.

Extractability is the remaining desirable characteristic; it is discussed in Section 3.3.2.

3.3 Some details concerning the algorithm

3.3.1 Step 2 of the algorithm: Eliminating subsumed clones

A clone pair P_1 *subsumes* another clone pair P_2 iff, treating each clone as a set of nodes, the two clones in P_1 are supersets of the two clones in P_2. There is no reason to report subsumed clone pairs; it is better to reduce the number of clone pairs reported, and to let the user split large clones if there is some reason to do so. Step 2 of the algorithm finds and deletes all clone pairs that are subsumed by other clone pairs.

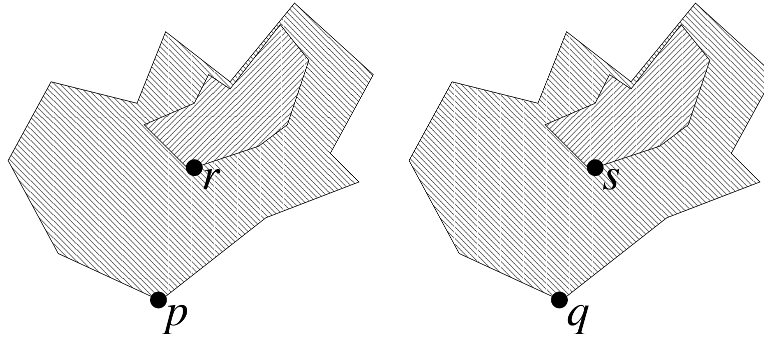


Figure 3.8 A slice-pair subsumed by another slice-pair

We employ two heuristics in Step 1 of the algorithm in an effort to avoid producing subsumed clone pairs in the first place. The motivation for doing so is to reduce the running time of Step 1, which is proportional to the number of clone pairs produced (Step 2 is quite efficient, comparatively).

3.3.1.1 Heuristic 1: do not treat previously mapped nodes as roots

Assume procedure `FindAClonePair` is called with two nodes p and q as roots, and assume two nodes r and s get mapped to each other during the process of finding the matching slice-pair starting from (p, q). Clearly (r, s) will be placed in the worklist after they are mapped to each other (see procedure `GrowCurrentClonePair` in Figure 3.4); therefore backward slicing will take place from r and s during the production of the slice-pair rooted at (p, q) (forward slicing will also take place, if r and s are predicates). This means that if we later try to produce a slice-pair with r and s as roots, we are likely to produce a slice-pair that is subsumed by the slice-pair rooted at (p, q) found earlier.

The situation just described is depicted pictorially in Figure 3.8. The slice-pair rooted at (p, q) is shown using one kind of shading (r and s are mapped to each other within this slice-pair). The subsequently generated slice-pair rooted at (r, s) is shown using a different kind of shading.

Our heuristic, based on the observation just made, is: remember each pair of nodes that has so far been mapped to each other in some slice-pair, and do not consider this pair of nodes

as roots in the future. The set `globalHistory` is used for this purpose; pairs are inserted into it in Procedure `GrowCurrentClonePair` (Figure 3.4), while Procedure `FindAllClonePairs` (Figure 3.3) searches it to determine which node pairs should be not considered as root pairs.

3.3.1.2 Heuristic 2: try nodes as roots in a good order

The purpose of this heuristic is to make the earlier heuristic more effective. Consider the example in Figure 3.8: the earlier heuristic will take effect, and the subsumed slice-pair will be not produced, only if the algorithm tries (p, q) as a pair of roots *before* trying (r, s) as a pair of roots. The second heuristic therefore attempts to find a good order in which to try the nodes as roots. Intuitively, the idea is to try nodes that are closer to the "leaves" of a PDG (assuming PDGs were acyclic graphs) as roots before nodes that are farther away from the "leaves". This heuristic is incorporated into the approach as follows: For each PDG `p`, do a depth-first traversal of `p` starting from its *entry* node. Then, for any two nodes p and r in `p`, look at the order in which these two nodes were visited (for the first time) during the depth-first traversal, and try the later-visited node as root first. Procedure `FindAllClonePairs` (in Figure 3.3) implements this heuristic.

Note that if there are no cycles in a PDG then this DFS-based approach guarantees that nodes that come "afterwards" in a PDG are tried as roots first. In the presence of cycles, although the notion of "afterwards" is not well defined, the approach still imposes some order on the nodes in the cycle.

3.3.2 Improving the chances of identifying extractable clones

Recall that one of the goals for the approach is to find groups of clones that are extractable (Section 3.2.2); i.e., we want to find groups of clones such that all the clones can be replaced (say, with the help of our extraction algorithms) by a call to a single, separate procedure without any change in the program's semantics. We now describe two aspects of Step 1 of the clone-detection algorithm that improve the likelihood that extractable clone groups are found.

Fragment 1	Fragment 2

`sum = 0`
`i = 0;`
`while(i <= 10) {`
`  sum = sum + i;`
`  i++;`
`}`
	`sum = 0;`
	`i = 0;`
	`while(i < 10) {`
	`  i++;`
	`  sum = sum + i;`
	`}`

Figure 3.9 Example illustrating treatment of loops by algorithm

3.3.2.1 Distinguishing loop-carried flow dependences from loop-independent flow dependences

Consider the example pair of code fragments in Figure 3.9. Consider the clone pair identified by the algorithm, starting from the two matching statements "`sum = sum + i`". In both fragments this statement is the target of a flow-dependence edge from the statement "`i++`"; however, the flow occurs from a previous iteration to a successive iteration in the first fragment, whereas in the second fragment the flow occurs within the same iteration. If the algorithm did not distinguish these two kinds of flows, it would map the two statements "`i++`" to each other in the backward slicing step (it would not map any other nodes to each other, as explained later). In other words, each clone would consist of the two statements "`sum = sum + i`" and "`i++`". However, this pair of clones is not extractable in any obvious way (our clone-group extraction algorithm would fail on this pair). The problem is that the two matching statements cannot be reordered, without affecting semantics, in either fragment.

To avoid such problems, the algorithm distinguishes *loop-independent* flows from *loop-carried* flows, and maps only flows of like kind to each other (see the second bullet-point condition within the statement labeled 3 in Procedure `GrowCurrentClonePair`, Figure 3.4). Formally, a loop-independent flow from a node n to a node m is a flow from n to m via a path that involves no *back edges* of loops that contain both n and m. A back edge of a loop

is a CFG edge from a node inside the loop to the loop's header (entry node). Any flow that is not loop-carried is loop-independent.

Therefore, in the example in Figure 3.9, starting from the two statements "`sum = sum + i`" as roots, the algorithm actually maps no other nodes to each other.

3.3.2.2 Loops crossed by mapped flow-dependence edges must have matching predicates

We describe here the *loop-crossing* condition – the third bullet-point condition within the statement labeled 3 in Figure 3.4. Consider again the example in Figure 3.9. If the loop-crossing condition were not part of the algorithm, then starting from the two statements "`sum = sum + i`" as roots, the algorithm would follow flow-dependence edges backward and map the two statements "`i = 0`" to each other, and also the two statements "`sum = 0`" to each other. In other words, each clone would consist of three statements: "`sum = 0`", "`i = 0`", and "`sum = sum + i`". Once again, there is no obvious way to extract these two clones: extracting the loops themselves is not possible, because the two loop predicates are non-matching, while the only other option – moving the first two statements in each clone to inside the loop – changes the semantics of the program. The clone-group extraction algorithm would, once again, fail on this pair of clones. The loop-crossing condition, which we explain below, prevents the flow-dependence edges from "`i = 0`" to "`sum = sum + i`" in the two fragments from being mapped to each other, and likewise for the flow-dependence edges from "`sum = 0`" to "`sum = sum + i`".

The loop-crossing condition is as follows. For each flow-dependence edge $n \to m$, we build a pair of lists: the first list contains the predicates of the loops that contain n, and the second list contains the predicates of the loops that contain m. Both lists are sorted from innermost loop to outermost loop, and both lists omit the predicates of loops that contain both n and m. Two flow-dependence edges are mapped to each other only if the first lists of both edges match, and the second lists of both edges match. Two lists match if both contain the same number of predicates, and corresponding predicates in the two lists match.

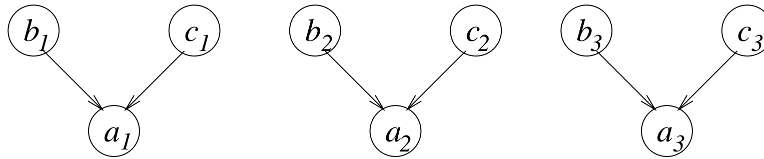


Figure 3.10 Illustration of clone-pairs grouping

In the example under discussion, the pair of lists for the dependence edge "`i = 0`" → "`sum = sum + i`" in the first fragment is (*empty list*, ("`while (i <= 10)`")), while the pair for the corresponding dependence edge in the second fragment is (*empty list*, ("`while (i < 10)`")). The second lists of the two edges do not match (the first lists match trivially), and so the two instances of "`i = 0`" are not mapped to each other.

3.3.3 Step 3 of the algorithm: Grouping pairs of clones

We describe here Step 3 of the algorithm – combining pairs of clones (generated in Step 1 and filtered by Step 2) into groups. The example in Figure 3.10, which shows three PDG subgraphs, illustrates Step 3. Say a_1, a_2, a_3 match each other, b_1, b_2, b_3 match each other, and c_1, c_2, c_3 match each other. Considering the a_i's as roots, Step 1 finds three pairs of clones; the first pair is $\{b_1 \rightarrow a_1, c_1 \rightarrow a_1\}, \{b_2 \rightarrow a_2, c_2 \rightarrow a_2\}$, the second pair is $\{b_1 \rightarrow a_1, c_1 \rightarrow a_1\}, \{b_3 \rightarrow a_3, c_3 \rightarrow a_3\}$, while the third pair is $\{b_2 \rightarrow a_2, c_2 \rightarrow a_2\}, \{b_3 \rightarrow a_3, c_3 \rightarrow a_3\}$. Notice that the first clone in the first pair is identical to the first clone in the second pair, that the second clone in the first pair is identical to the first clone in the third pair, and that the second clone in the first pair is identical to the second clone in the second pair. Therefore, Step 3 merges the three clone pairs into a single group of three clones: $\{b_1 \rightarrow a_1, c_1 \rightarrow a_1\}, \{b_2 \rightarrow a_2, c_2 \rightarrow a_2\}, \{b_3 \rightarrow a_3, c_3 \rightarrow a_3\}$.

We now formally specify the algorithm for Step 3, beginning with two definitions. A clone is said to be "identical" to another clone iff the two of them, treated as sets of dependence edges, are equal; two clone pairs are said to "overlap" iff a clone in one of the pairs is identical to a clone in the other. The approach for this step is: Partition the set of all clone pairs (that remain after Step 2) based on the transitive closure of the overlap relation (i.e., place

two clone pairs in the same partition if the two are related by the transitive closure of the overlap relation). Finally, generate one group of clones from each partition, by unioning the pairs of clones in the partition and eliminating duplicate copies of identical clones.

3.4 Discussion

Based on experiments (see Chapter 7) with the implementation of the clone-detection approach, we infer that the approach is likely to find most interesting clone groups, including ones that involve non-contiguous clones and out-of-order matches. It finds a few uninteresting clones, but mainly at small sizes. (For this reason, and for other reasons discussed in Section 3.4.4, the programmer needs to examine the output of the tool, and determine from that the clone groups that are good candidates for extraction.) We provide quantitative evidence for these claims in Chapter 7. In this section we present a discussion of the merits and drawbacks of the approach, in a qualitative sense.

3.4.1 Examples of interesting, extractable clones identified

Some examples of interesting clone groups identified by the tool we implemented are listed below. Each example, except the one involving intertwined clones, is from the source code of the Unix utility *bison*; the intertwined clones are from the source code of the Unix utility *sort*. These clone groups can all be extracted into separate procedures in a fairly straightforward manner, and all make sense as separate procedures.

- The four-clone group shown in Figure 3.1; note that two of the clones are non-contiguous.

- The two clones shown in Figure 3.11 using the "`++`" signs. Both clones are from the module of *bison* that reads the input grammar (*bison* is a parser generator). Each `case` block shown performs the following task: if the current character is a "$", it reads the next character and based on what that is, it reads the rest of the token in an appropriate manner.

In this example the tool did not identify the ideal clone pair. Ideally, the two statements labeled `(1)` should have been mapped to each other; however, the statement labeled `(1)` in the first clone is actually mapped to the statement labeled `(10)` in the second clone. Also, the calls to "`getc`" outside the two `case` blocks, and the `switch` predicates, should have been excluded from the clone pair. (The calls to `getc` cannot be extracted together with the rest of the clone because of certain flow dependences from these nodes to other nodes that are not shown in the figure. The `switch` can technically be extracted out, by making a duplicate copy of it, but the extracted procedure is more readable if it contains only the contents of the "`case '$'`" block.)

In other words, the tool identified a *variant* of the ideal clone pair. In fact the tool identified several other variants of this same ideal clone pair (we discuss this further in Section 3.4.4). Note that even the ideal clones in this example are non-contiguous (because the two statements labeled `(3)` and the statement labeled `(2)` have to remain unmapped).

- The two clones shown in Figure 3.5. These were part of a three-clone group. The third clone involved a different renaming of variables, and used the same statement ordering as the clone in Fragment 1.

- The pair of intertwined clones shown in Figure 3.6 from *sort*. Here too the tool found a variant of the ideal clone pair – ideally the `if` predicates should not have been included in the clone pair.

- A group of seven clones, identical except for variable names. Two of the clones are shown in Figure 3.12. This code prints the contents of an array (`check` / `rrhs`), ten entries to a line, separated by commas.

```
++ switch(c) {
   ...
++ case '$':
++  c = getc(finput);                         (1)
++  type_name = NULL;
++  if (c == '<') {
++    register char *cp = token_buffer;
++    while ((c = getc(finput)) != '>'
++           && c > 0)
++      *cp++ = c;
++    *cp = 0;
++    type_name = token_buffer;
++    c = getc(finput);
++  }
++  if (c == '$') {
++    fprintf(fguard, "yyval");
++    if (!type_name)
        type_name = rule->sym->type_name;   (3)
++    if (type_name)
++      fprintf(fguard, ".%s", type_name);
++    if(!type_name && typed)                 (4)
++      warns("$$ of `%s' has no declared
++             type", rule->sym->tag);
++  }
++  else if (isdigit(c) || c == '-') {
++    ungetc (c, finput);
++    n = read_signed_integer(finput);
++    c = getc(finput);
++    if (!type_name && n > 0)
++      type_name = get_type_name(n, rule);
++    fprintf(fguard, "yyvsp[%d]",
++            n - stack_offset);
++    if (type_name)
++      fprintf(fguard, ".%s", type_name);
++    if(!type_name && typed)                 (5)
++      warnss("$%s of `%s' has no declared
++              type", int_to_string(n),
++              rule->sym->tag);
++    continue;
++  }
++  else
++    warni("$%s is invalid",                 (8)
++          printable_version(c));
++  break;
    ...
   }
++ c = getc(finput);                          (9)
++ c = getc(finput);                         (10)
   c = getc(finput);                         (11)
++ c = getc(finput);                         (12)
```

```
++ switch(c) {
   ...
++ case '$':
    c = getc(finput);                         (1)
++  type_name = NULL;
++  if (c == '<') {
++    register char *cp = token_buffer;
++    while ((c = getc(finput)) != '>'
++           && c > 0)
++      *cp++ = c;
++    *cp = 0;
++    type_name = token_buffer;
      value_components_used = 1;            (2)
++    c = getc(finput);
++  }
++  if (c == '$') {
++    fprintf(fguard, "yyval");
++    if (!type_name)
        type_name = get_type_name(n, rule); (3)
++    if (type_name)
++      fprintf(fguard, ".%s", type_name);
++    if(!type_name && typed)                 (4)
++      warns("$$ of `%s' has no declared
++             type", rule->sym->tag);
++  }
++  else if (isdigit(c) || c == '-') {
++    ungetc (c, finput);
++    n = read_signed_integer(finput);
++    c = getc(finput);
++    if (!type_name && n > 0)               (13)
++      type_name = get_type_name(n, rule);
++    fprintf(fguard, "yyvsp[%d]",
++            n - stack_offset);
++    if (type_name)
++      fprintf(fguard, ".%s", type_name);
++    if(!type_name && typed)                 (5)
++      warnss("$%s of `%s' has no declared
++              type", int_to_string(n),
++              rule->sym->tag);
++    continue;
++  }
++  else
++    warni("$%s is invalid",                 (8)
++          printable_version(c));
++  break;
    ...
   }
++ c = getc(finput);                          (9)
++ c = getc(finput);                         (10)
++ c = getc(finput);                         (11)
++ c = getc(finput);                         (12)
```

(The four final calls to `getc` actually occur non-contiguously in both fragments.)

Figure 3.11 Two non-contiguous clones identified by the tool in *bison*

`++ j = 10;`
`++ for (i = 1; i <= high; i++) {`
`++  putc(',', ftable);`
`++  if (j >= 10) {`
`++    putc('\n', ftable);`
`++    j = 1;`
`++  }`
`++  else`
`++    j++;`
`++  fprintf(ftable, "%6d", check[i]);`
`++ }`
	`++ j = 10;`
	`++ for (i = 1; i <= nrules; i++) {`
	`++  putc(',', ftable);`
	`++  if (j >= 10) {`
	`++    putc('\n', ftable);`
	`++    j = 1;`
	`++  }`
	`++  else`
	`++    j++;`
	`++  fprintf(ftable, "%6d", rrhs[i]);`
	`++ }`

Figure 3.12 Seven copies of this clone were found in *bison*

3.4.2 Examples of interesting, not-easily-extractable clones identified

The tool also finds some clone groups that are not as readily extractable, but are still interesting. Some of these clone groups *can* be extracted by a programmer (although they would pose challenges to automatic clone-group extraction algorithms, such as our own); such groups are clearly worth reporting. However, there are some interesting groups identified that are not easily extractable even by a programmer, but are still worth identifying and reporting, for a variety of reasons: Mere knowledge of the existence of such clone groups may improve program understanding. Bug-fixes and enhancements done on one fragment can be more reliably propagated to other similar fragments that need the same change. Finally, if the program is ever re-implemented (for any reason), knowledge of the existence of clones will assist in the creation of a better new design that obviates the need for the clones. Interesting but not-easily-extractable clone groups identified by the tool fall into several categories, as discussed below.

3.4.2.1 Category 1: extraction costs outweigh benefits

In this category the clones perform meaningful computations, but the cloned code is not large enough for the benefits of extraction to outweigh the costs; e.g., the extracted procedure might need too many parameters, or too many predicates (relative to the size of the clones) might need to be duplicated to make a group extractable.

3.4.2.2 Category 2: extraction requires appropriate language support

For the clone groups in this category, the clones within a group differ such that extraction can be performed only with appropriate language support. Examples of clone groups that fall into this category are ones that involve slight differences in operators. For example, two clones identified by the tool were mostly identical, but a "+" in one of the clones corresponded to a "-" in the other. In other cases clones in a group differ in the types of the variables

they involve. For example, there are three `for` loops in a file in *bison*, such that each loop iterates over a linked list, and all three loops are syntactically identical to each other; the list elements are of different `struct` types, but each of the types has a `next` field and a `number` field, with the loop doing a computation on the `number` fields. Procedure extraction is problematic for both these examples; however with a language like C, macros can be used to eliminate the clones in both examples. The second example can also be handled using a template mechanism (as in C++).

3.4.2.3 Category 3: extraction requires sophisticated transformations

For the clone groups in this category, the clones within a group differ such that sophisticated semantics-preserving transformations are required for extraction.

For example, consider the clone pair shown in Figure 3.13 that is identified by the tool in *bison*. The two
`case '\'': case '"'` blocks have the same functionality: they read a string (delimited by single/double quotes) from the input stream `finput`. Notice that an "`if .. else if ..`" statement in the first clone (the first statement in the `while` loop's body) is replaced with a sequence of two `if` statements (in the opposite order) in the second clone. This difference is a matter of structure, not of semantics. Although our automatic clone-group extraction algorithm is not sophisticated enough to recognize that one of these clones can be restructured to make it syntactically identical to the other (and therefore cannot handle this clone pair), a programmer ought to be able to do this restructuring, and the subsequent extraction. This extraction would improve the program. Therefore, it is beneficial for clone groups such as this one to be identified; the tool identifies this clone group, in spite of the difficult characteristics it exhibits. It does so by:

1. starting from the two matching calls `fatal("unterminated string")`
2. slicing back from the roots along control-dependence edges to reach the two "`if (c == EOF)`" predicates

3. slicing back from there along loop-carried flow-dependence edges to reach the two "`c = match`" statements (these flow dependences occur due to a `while` loop, not shown in Figure 3.13, that surrounds each `switch` statement)

4. slicing backward from there to reach the rest of the clone pair (with forward slicing also, from predicates reached).

Note that here the tool identified a variant of the ideal clone pair – the ideal clone pair does not include the `switch` predicates, and does not include the three other `case` blocks (see Figure 3.13).

Another example in this category is a pair of similar fragments, shown in Figure 3.14, with each fragment reading in a comment from the input grammar. While the two fragments have many matching statements, there are significant differences between the two:

- The `while` loop in the first clone has the predicate "`while(!ended)`", while in the second clone the corresponding predicate is "`while(in_comment)`". The first clone sets `ended` to 1 inside the loop when it sees the end of the comment, whereas the second one sets `in_comment` to 0 when it sees the end of the comment.

- The first clone writes out every character in the comment to one output stream, `faction`, whereas the second clone writes out every character to one stream `fattrs` and also to another stream `fdefines`, if `fdefines` is not null.

- The single "`if..else if..else..`" statement inside the `while` loop in the first clone is replaced by a sequence of three separate `if` statements in the second clone, and the matching is out-of-order.

Here again, the tool is able to identify (variants of) this clone pair. It is useful to identify this clone pair, because a programmer could restructure one of the two fragments to make it identical to the other, and then derive benefit by extracting the clones into a separate procedure.

```
++ switch(c) {
   ...
++ case '\'':
++ case '"':
++   match = c;
++   putc(c, faction);
++   c = getc(finput);
++   while (c != match) {
++     if (c == '\n') {
++       warn("unterminated string");
++       ungetc(c, finput);
++       c = match;
++       continue;
++     }
++     else if (c == EOF)
++       fatal("unterminated string at
++              end of file");
++     putc(c, faction);
++     if (c == '\\') {
++       c = getc(finput);
++       if (c == EOF)
++         fatal("unterminated string");
++       putc(c, faction);
++       if (c == '\n')
++         lineno++;
++     }
++     c = getc(finput);
++   }
++   putc(c, faction);
++   break;
   ...
++ ... three other case blocks ...
```

```
++ switch(c) {
   ...
++ case '\'':
++ case '"':
++   match = c;
++   putc(c, fguard);
++   c = getc(finput);
++   while (c != match) {
++     if (c == EOF)
++       fatal("unterminated string at
++              end of file");
++     if (c == '\n') {
++       warn("unterminated string");
++       ungetc(c, finput);
++       c = match;
++       continue;
++     }
++     putc(c, fguard);
++     if (c == '\\') {
++       c = getc(finput);
++       if (c == EOF)
++         fatal("unterminated string");
++       putc(c, fguard);
++       if (c == '\n')
++         lineno++;
++     }
++     c = getc(finput);
++   }
++   putc(c, fguard);
++   break;
   ...
++ ... three other case blocks ...
```

Figure 3.13 Two clones in *bison* that are semantically identical but structurally different

```
case '/':
  putc(c, faction);
  c = getc(finput);
  if (c != '*' && c != '/')
    continue;
  cplus_comment = (c == '/');
  putc(c, faction);
  c = getc(finput);

  ended = 0;
  while (!ended) {
    if (!cplus_comment && c == '*') {
      while (c == '*') {
        putc(c, faction);
        c = getc(finput);
      }

      if (c == '/') {
        putc(c, faction);
        ended = 1;
      }
    }
    else if (c == '\n') {
      lineno++;
      putc(c, faction);
      if (cplus_comment)
        ended = 1;
      else
        c = getc(finput);
    }
    else if (c == EOF)
      fatal("unterminated comment");
    else {
      putc(c, faction);
      c = getc(finput);
    }
  }
  break;
```

```
case '/':
  c = getc(finput);
  if (c != '*' && c != '/')
    ungetc(c, finput);
  else {
    putc(c, fattrs);
    if (fdefines)
      putc(c, fdefines);
    cplus_comment = (c == '/');
    in_comment = 1;
    c = getc(finput);

    while (in_comment) {
      putc(c, fattrs);
      if (fdefines)
        putc(c, fdefines);

      if (c == '\n') {
        lineno++;
        if (cplus_comment) {
          in_comment = 0;
          break;
        }
      }
      if (c == EOF)
        fatal("unterminated comment at
              end of file");

      if (!cplus_comment && c == '*') {
        c = getc(finput);
        if (c == '/') {
          putc('/', fattrs);
          if (fdefines)
            putc('/', fdefines);
          in_comment = 0;
        }
      }
      else
        c = getc(finput);
    }
  }
  break;
```

Figure 3.14 Two fragments in *bison* that each read a comment from the input grammar

Note that although this example clone pair and the previous one were identified by the tool despite differences in structure between the clones, the tool cannot in general find *all* clone pairs where the clones have similar (or identical) semantics, irrespective of structure and irrespective of syntactic similarity (that would require solving the program equivalence problem, which is undecidable). Our approach can find clones only when the statements in one clone match the statements in the other clone (syntactically) and when dependences match.

3.4.2.4 Category 4: multiple options for extraction

The clone groups in this category are challenging in the sense that there are multiple options for extraction, none of which is clearly superior to the others. Consider the example clone pair in Figure 3.15, identified by the tool in *bison*. The two fragments each compute a set (implemented as a bit vector), by adding items to the set in each iteration of the outer loop. Each fragment actually uses two bit vectors, one to hold the current set and the other to hold the set as it was at the end of the previous iteration. The body of the outer loop in the first fragment performs the following tasks:

1. Copy the current set, which is in the bit vector pointed to by `N`, into the bit vector pointed to by `Np`.

2. Turn on additional bits in the bit vector pointed to by `Np` (this happens in the second `for` loop – its actual functionality can be ignored for our purposes).

3. Check if the bit vectors pointed to `Np` and `N` are identical. If yes, then nothing was added to the bit vector in this iteration; therefore, quit the loop. Otherwise, let `N` point to the bit vector that `Np` points to, and vice versa. Therefore `N` ends up pointing to the latest version of the set, whereas `Np` ends up pointing to the set as it was at the beginning of the current iteration. Then start the next iteration.

The second fragment has similar functionality, except that its second step differs in what it does (the clones also differ in variable names). Note that both clones identified by the tool

```
++ while (1) {
++  for (i = WORDSIZE(nvars) - 1; i >= 0;
++       i--)
++   Np[i] = N[i];
++  for (i = 1; i <= nrules; i++) {
     if (!BITISSET(P, i)) {
      if (useful_production(i, N)) {
       SETBIT(Np, rlhs[i] - ntokens);
       SETBIT(P, i);
      }
     }
++  }
++  if(bits_equal(N, Np, WORDSIZE(nvars)))
++   break;
++  Ns = Np;
++  Np = N;
++  N = Ns;
++ }
++ FREE(N);
```

```
++ while (1) {
++  for (i = WORDSIZE(nsyms) - 1; i >= 0;
++       i--)
++   Vp[i] = V[i];
++  for (i = 1; i <= nrules; i++) {
     if (!BITISSET(Pp, i) && BITISSET(P, i)
         && BITISSET(V, rlhs[i])) {
      for (r=&ritem[rrhs[i]]; *r>=0; r++) {
       if (ISTOKEN(t = *r)
           || BITISSET(N, t - ntokens)) {
        SETBIT(Vp, t);
       }
      }
      SETBIT(Pp, i);
     }
++  }
++  if(bits_equal(V, Vp, WORDSIZE(nsyms)))
++   break;
++  Vs = Vp;
++  Vp = V;
++  V = Vs;
++ }
 end_iteration:
++ FREE(V);
```

Figure 3.15 Two non-contiguous clones identified by the tool in *bison*

are non-contiguous. In this example, unlike in the previous two examples, there is no single obvious strategy for extraction. There are a number of extraction options for this example, each with its own pros and cons:

- The intervening mismatching code from each fragment can be placed in the extracted procedure, guarded by boolean parameters. This option is probably not desirable, because the mismatching code fragments are of large size. However, this is the option that would be adopted by our clone-group extraction algorithm on this example.

- Each of the two intervening mismatching fragments can be made into a separate procedure; the matching code can then be extracted such that it contains an indirect call `(*fp)`, where `fp` is a function pointer parameter that is set at each call site to the appropriate procedure.

- The clone pair identified by the tool can be split into two, with the initial matching chunks being extracted into one procedure and the later matching chunks being extracted into another.

  This option, although advantageous in the sense that it involves no guarded code or calls through function pointers, does have the disadvantage of separating the two cloned chunks, which *are* logically related, into two (independent) procedures. However, in an object-oriented language, this disadvantage can be somewhat overcome by making the two extracted procedures methods of a common class.

Whereas other previous approaches to clone detection would report each of the two pairs of matching chunks in this example as a separate clone pair, our approach reports them as a single clone pair. This is beneficial in two ways:

- As the two cloned chunks are logically related, from the program-understanding perspective it helps if they are shown together as a single clone pair.

- Although the final decision might very well be to split the clone pair into two, presenting them as one clone pair allows the user to consider all the extraction options listed above. In other words there is greater flexibility.

3.4.3 Non-one-to-one variable-name mappings

Most of the clone groups identified by the clone-detection approach have the property that variable names in the clones in a group are involved in a one-to-one mapping; i.e., if a variable v_1 occurs in some position in a clone such that a variable v_2 occurs in the corresponding position in some other clone in the group, then for every position in the first clone where v_1 occurs the corresponding position in the second clone has v_2, and vice versa. (All the example clone groups shown so far in this dissertation have this property.) This property is desirable because it makes the task of determining the parameters and local variables of the extracted procedure straightforward (every group of mapped variables becomes a parameter/local variable of the extracted procedure). However, the clone-detection approach does not explicitly guarantee this property (recall that, as stated early in this chapter, two nodes are regarded as matching if their internal expressions are identical *ignoring* all variable names and literal values); in fact our implementation did find a few interesting clone groups that did not satisfy this property. Most such clone groups were, however, still extractable, although determining the local variables and parameters for the extracted procedure is non-trivial.

An example of such a clone pair identified by the tool is shown in Figure 3.16. The two clones are identical, except in the lines labeled `(1)` and `(2)`; in particular every occurrence of `p` (`p2`) in the first clone corresponds to an occurrence of `p` (`p2`) in the second clone, except that the occurrence of `p` in the line labeled `(1)` in the first clone corresponds to `p2` in the second clone. In other words, the variable renaming is not one-to-one (the variable-name mismatch on line `(2)`, where `o_file` matches `o_override`, is not a problem, since these two variables do not occur anywhere else in the two clones). However, this clone group can be extracted: The line labeled `(1)` will, in the extracted procedure, be "`p2 = next_token (t + 6)`", where

```
if (ignoring)
  in_ignored_define = 1;
else {
  p2 = next_token (p + 6);              (1)
  if (*p2 == '\0')
    fatal (&fileinfo,
           _("empty variable name"));
  p = strchr (p2, '\0');
  while (isblank ((unsigned char)p[-1]))
    --p;
  do_define(p2, p - p2, o_file,         (2)
            infile, &fileinfo);
}
```

```
if (ignoring)
  in_ignored_define = 1;
else {
  p2 = next_token (p2 + 6);             (1)
  if (*p2 == '\0')
    fatal (&fileinfo,
           _("empty variable name"));
  p = strchr (p2, '\0');
  while (isblank ((unsigned char)p[-1]))
    --p;
  do_define(p2, p - p2, o_override,     (2)
            infile, &fileinfo);
}
```

Figure 3.16 Example clone pair from *make* illustrating non-one-to-one variable renaming

++ p = filename;
   ...
++ ...p...
   ....
++ ..p...
	++ p = filename;
	   ...
	++ ...filename...
	   ....
	++ ..filename...

Figure 3.17 Skeleton of a clone pair in *make* that illustrates non-one-to-one variable renaming

`t` is a parameter that is given the values of the variables `p` and `p2` from the two call sites, respectively. The other variable occurrences in the extracted procedure are identical to the corresponding occurrences in the two clones, with both `p` and `p2` being "output" parameters of the extracted procedure.

Figure 3.17 shows (the skeleton of) another clone pair identified by the tool in *make* that exhibits non-one-to-one variable renaming. The renaming is not one-to-one in this example because `p` in the first clone is mapped to `p` in the second clone in the first line, but to `filename` in the other lines. In this example extraction can be performed by replacing the second and third occurrences of `p` in the first clone by `filename`; this *copy propagation* step makes the two clones identical, thereby enabling the creation of the new procedure.

In both the examples above, which were found in real programs by the tool, there was a clean way to produce the extracted procedure in spite of the non-to-one variable renaming. It is a strength of our approach that it is able to identify clone groups such as these that are interesting and extractable, but exhibit non-one-to-one renaming.

Parts (a) and (b) in Figure 3.18 show an (artificial) example clone pair that *can* be identified by the approach, but that exhibits characteristics that almost never occur in practice; the approach can identify this pair of clones by starting from the second pair of matching nodes (involving the "`*`" operator), and slicing back from the right-hand-side operands of the "`*`" operations to reach the first pair of matching nodes. The variable renaming in this pair of clones is clearly non-one-to-one; moreover, assuming that variables `b` and `c` are live at the exit of the first clone, and variables `e` and `f` are live the exit of the

(a) Clone 1

```
++ b = a + 1;
++ c = a * b;
```

(c) Rewritten Clone 1

```
f(&b, &a, &c, &a);
```

(b) Clone 2

```
++ e = d + 1;
++ f = e * e;
```

(d) Rewritten Clone 2

```
f(&e, &d, &f, &e);
```

(e) Extracted Procedure

```
void
f(int *p1, int *p2,
  int *p3, int *p4) {

  (*p1) = (*p2) + 1;
  (*p3) = (*p4) * (*p1);
}
```

Figure 3.18 Extraction of a difficult-to-extract pair of clones

second clone, because of the mismatching data flows (the left-hand-side operand of the "*" operator uses a value defined outside the clone in the first clone, but a value defined inside the clone in the second clone), the solutions discussed in earlier examples do not work. An extraction solution that does work for this example is shown in parts (c), (d), and (e) of Figure 3.18; notice that every variable occurrence in the extracted procedure is a pointer dereference, and that appropriate addresses are passed in the two call sites to ensure that data flows remain identical before and after extraction. This transformation is not a very good one, because it is likely to worsen the understandability of the program. However, as we said earlier, clone groups that we observed in practice almost never exhibit such mismatching data flows; the few clone groups in real programs that did exhibit non-one-to-one variable renamings were, in the vast majority of cases, cleanly extractable, as illustrated by the previous two examples in this section.

3.4.4 A drawback of the approach: variants

An ideal clone group is a clone group that a programmer would consider interesting. A *variant* is a clone group that resembles, but is not identical to, an ideal clone pair. The difference between the variant and the ideal could be in terms of which nodes are included in the clone group, and/or how the mappings between the nodes in the clones are defined. Our approach to clone-group detection does have the drawback of often identifying, instead

of an ideal clone group, multiple (overlapping) clone groups that are all variants of the ideal group.

We use the clone pair in Figure 3.11, which is identified by the tool in the source code of *bison*, and which is a variant of an ideal clone pair, to describe why variants are identified. The ideal clone pair in this case is the two entire `case` blocks, excluding the statement labeled `(2)` and the two statements labeled `(3)`, and nothing more. The variant clone pair identified by the tool (shown in Figure 3.11) was obtained by starting from the two matching calls to `warni`. Here is how (and why) the variant pair identified by the tool differs from the ideal pair:

- The two `switch` predicates are included in the variant identified; they were reached by backward slicing (along control-dependence edges) from the the pair of `case` labels.

- Statements `(9)`, `(10)` and `(12)` in the first fragment are mapped to statements `(9)`, `(11)` and `(12)` in the second fragment, respectively; these pairs of nodes are reached by slicing back from the two `switch` predicates along loop-carried flow-dependence edges (these flow dependences occur due to a `while` loop, not shown in Figure 3.11, that surrounds each `switch` statement).

- Statement `(1)` in the first clone is mapped to statement `(10)` in the second clone, instead of being mapped to statement `(1)` in the second fragment. Here is why this happened: There is a loop-carried flow-dependence edge in the first fragment from statement `(1)` to the `switch` predicate. There is *no* flow dependence from `(11)` to the `switch` predicate (the path between these two nodes is blocked by a redefinition of `c` not shown in the figure). In the second fragment, there are loop-carried flow-dependence edges from both statements, `(1)` and `(10)`, to the `switch` predicate. Therefore there is non-determinism in the algorithm – when slicing backward from the `switch` predicates, `(1)` in the first fragment could be mapped to either `(1)` or `(10)` in the second fragment; the algorithm ended up making the second (and less desirable) choice.

The tool also identifies *other* clone pairs, besides the one shown in Figure 3.11, that are variants of the same ideal clone pair. One of these other clone pairs is shown in Figure 3.19. This pair was identified by starting the slicing from predicate `(4)` in the first fragment and predicate `(5)` in the second fragment. In fact the tool identifies yet another clone pair that is very similar to the one shown in Figure 3.19; the only difference is that this pair was obtained by starting the slicing from `(5)` in the first fragment and `(4)` in the second fragment. Note that statement `(11)` in the first fragment is present in the clone pair in Figure 3.19 but not in the clone pair in Figure 3.11; if this were not true then the clone pair in Figure 3.19 would have been subsumed by the clone pair in Figure 3.11, and therefore would have been removed from the output of the tool (see Step 2 of the algorithm, at the beginning of this chapter).

The production of multiple variants of an ideal clone group, instead of just the ideal group, causes some problems:

- The running time of the tool is proportional to the number of clone groups produced; therefore, every additional variant produced increases the time requirement.
- Users have to view more clone groups than they would have to if just the ideal groups were reported. Moreover, they need to recognize which reported variants correspond to the same ideal group, and they need to determine the ideal groups.
- The operation of Step 3 of the algorithm – grouping – can be affected. Say there is a group of many (more than two) clones in a program. The algorithm begins by identifying several pairs of clones (in Step 1). If each clone pair identified is exactly the ideal one, then all the clone pairs are related to each other via the "overlap" relation (see Section 3.3.3). Therefore all these pairs would be merged in Step 3 into a single clone group, which is the ideal outcome. However, due to the variants problem, Step 1 could end up identifying pairs of clones such that no clone in a pair is identical to any other clone in another pair. If this happens then Step 3 of the algorithm would merge nothing, therefore leaving all the clone pairs in the final output (which is undesirable).

```
++ switch(c) {
   ...
++ case '$':
++  c = getc(finput);                          (1)
++  type_name = NULL;
++  if (c == '<') {
++    register char *cp = token_buffer;
++    while ((c = getc(finput)) != '>'
++           && c > 0)
++      *cp++ = c;
++    *cp = 0;
++    type_name = token_buffer;
++    c = getc(finput);
++  }
    if (c == '$') {
      fprintf(fguard, "yyval");
      if (!type_name)
        type_name = rule->sym->type_name;      (3)
      if (type_name)
        fprintf(fguard, ".%s", type_name);
++    if(!type_name && typed)                  (4)
        warns("$$ of `%s' has no declared
               type", rule->sym->tag);
    }
    else if (isdigit(c) || c == '-') {
      ungetc (c, finput);
      n = read_signed_integer(finput);
      c = getc(finput);
      if (!type_name && n > 0)
        type_name = get_type_name(n, rule);
      fprintf(fguard, "yyvsp[%d]",
              n - stack_offset);
      if (type_name)
        fprintf(fguard, ".%s", type_name);
      if(!type_name && typed)                  (5)
        warnss("$%s of `%s' has no declared
                type", int_to_string(n),
                rule->sym->tag);
      continue;
    }
    else
      warni("$%s is invalid",                  (8)
            printable_version(c));
    break;
    ...
   }
++ c = getc(finput);                           (9)
++ c = getc(finput);                          (10)
++ c = getc(finput);                          (11)
++ c = getc(finput);                          (12)
```

```
++ switch(c) {
   ...
++ case '$':
    c = getc(finput);                          (1)
++  type_name = NULL;
++  if (c == '<') {
++    register char *cp = token_buffer;
++    while ((c = getc(finput)) != '>'
++           && c > 0)
++      *cp++ = c;
++    *cp = 0;
++    type_name = token_buffer;
      value_components_used = 1;               (2)
++    c = getc(finput);
++  }
    if (c == '$') {
      fprintf(fguard, "yyval");
      if (!type_name)
        type_name = get_type_name(n, rule); (3)
      if (type_name)
        fprintf(fguard, ".%s", type_name);
      if(!type_name && typed)                  (4)
        warns("$$ of `%s' has no declared
               type", rule->sym->tag);
    }
    else if (isdigit(c) || c == '-') {
      ungetc (c, finput);
      n = read_signed_integer(finput);
++    c = getc(finput);
      if (!type_name && n > 0)
        type_name = get_type_name(n, rule);
      fprintf(fguard, "yyvsp[%d]",
              n - stack_offset);
      if (type_name)
        fprintf(fguard, ".%s", type_name);
++    if(!type_name && typed)                  (5)
        warnss("$%s of `%s' has no declared
                type", int_to_string(n),
                rule->sym->tag);
      continue;
    }
    else
      warni("$%s is invalid",                  (8)
            printable_version(c));
    break;
    ...
   }
++ c = getc(finput);                           (9)
++ c = getc(finput);                          (10)
++ c = getc(finput);                          (11)
++ c = getc(finput);                          (12)
```

(The four final `getc()`s actually occur non-contiguously, in both fragments.)

Figure 3.19 A clone pair identified by the tool – this pair and the pair in Figure 3.11 are variants of the same ideal pair

As discussed in Section 3.2.2.3, the use of forward slicing in the algorithm helps suppress the production of variants. In addition, the approach incorporates two other heuristics, distinguishing loop-carried flow dependences from loop-independent ones (Section 3.3.2.1), and requiring that the loops crossed by mapped flow dependences have matching predicates (Section 3.3.2.2), for the same purpose. These heuristics increase the likelihood that nodes that should not be mapped to each other ideally are not mapped to each other by the algorithm. These heuristics, together with forward slicing, suppress (but do not eliminate) the production of variants. Therefore, rather than supplying every clone group identified by the tool directly to an automatic clone-group extraction algorithm, a programmer should look at the clone groups reported, and adjust them to make them ideal. These adjustments may include:

- Removing unnecessary nodes from a clone group; e.g., statements `(9), (10), (11)` and `(12)` should be removed from both clones shown in Figure 3.11.
- Adjusting how the nodes in a clone are mapped to nodes in other clones; e.g., the two statements labeled `(1)` in Figure 3.11 should be mapped to each other.
- Adding nodes that are not part of the reported clone group, but should be.
- Splitting a large reported clone group into several smaller ones. For instance, the clone pair shown in Figure 3.13 should be split into four clone pairs, one for each `case` block. As another example, the clone pair in Figure 3.15 should perhaps be split into two clone pairs.

In addition, when the tool reports multiple variant clone groups for the same ideal group (as illustrated in Figures 3.11 and 3.19), the programmer needs to recognize this, and avoid redundant work. To support this it might be possible to devise simple, automatic heuristics that group together related clone groups produced by the tool, and pick and show one or a few promising candidates from each group to the user.

In conclusion, the approach does suffer from the problem of variants. However, this is not an overwhelming problem. In fact, after we ran the tool on the source code of the GNU utility *make*, one person (the author) was able to look at all clone groups reported by the tool whose clones had 30 or more nodes (127 groups), and determine the corresponding ideal clone groups that were worthy of extraction (11 groups), in less than 4 hours time (we chose thirty nodes as a size threshold to save manual effort – in our experience interesting clones are often large). The source code of *make* is over 30,000 lines long; manual examination of this much source code is likely to take far more time, and would involve a fair of amount of risk in terms of missed clones.

3.4.5 Other problems with the approach

Our approach has two other drawbacks, besides identifying variants, that we discuss in this section.

3.4.5.1 Uninteresting clones are identified

The approach sometimes finds uninteresting clones (clones that do not represent a meaningful computation that makes sense as a separate procedure). An example of a pair of uninteresting clones identified by the tool in *bison* is shown in Figure 3.20; these two clones were identified by starting from the two matching calls to `putc`, and by following data-dependence edges backward to reach the matching assignments to `c`. The two clones are not meaningful computations, and are not extractable either, even by a programmer.

Although the approach does find uninteresting clones, in our experience the problem is not a severe one. We found in our experiments (Chapter 7) that most uninteresting clone groups found by the tool are rather small in size. Conversely, we also found that most interesting clones were somewhat large in size. Therefore a good heuristic to get around the problem of uninteresting clones is to pick some reasonable size-threshold and look only at reported clones of size larger than that threshold.

```
while(brace_flag ? ...) {
  switch(c) {
    ...
    case '}':
++    putc(c, fguard);
      ...
++    c = getc(finput);
    case '\'': case '"':
      ...
++    c = getc(finput);
      ...
    case '/':
      ...
++    c = getc(finput);
      ...
++    c = getc(finput);
      ...
    case '$':
      ...
++    c = getc(finput);
      ...
  }
  ...
}
```

```
while (c != '}') {
  switch(c) {
    ...
    case '/':
      ...
++    c = getc(finput);
      ...
    case '$':
      ...
++    c = getc(finput);
      ...
    case '@':
++    c = getc(finput);
      ...
++    c = getc(finput);
      ...
    default:
++    putc(c, faction);
  }
++ c = getc(finput);
}
```

Figure 3.20 A pair of uninteresting clones identified by the tool in *bison*

`1a: c = 2;`
`2a: d = a * b;`
`3a: e = b / 2;`
`4a: f = c + d  + e;`
	`1b: c = a - 2;`
	`2b: d = a * b;`
	`3b: e = b / 2;`
	`4b: f = c + d + e;`
		`1c: c = a - 2;`
		`2c: d = 2 + a;`
		`3c: e = b / 2;`
		`4c: f = c + d + e;`

Figure 3.21 Example illustrating a group of clones missed by the tool

3.4.5.2 Some groups of clones are missed

The algorithm sometimes misses groups that consist of more than two clones because it first finds clone pairs, and then combines them into groups. Consider the (toy) example shown in Figure 3.21. Assuming that the nodes labeled `4` are tried first as roots, the approach identifies three clone pairs for this example:

- The pair rooted at (`4a, 4b`) that also contains the node pairs (`2a, 2b`) and (`3a, 3b`).
- The pair rooted at (`4a, 4c`) that also contains the node pair (`3a, 3c`).
- The pair rooted at (`4b, 4c`) that also contains the node pairs (`1b, 1c`) and (`3b, 3c`).

Note that none of the six individual clones listed above are identical to each other. Therefore Step 3 of the algorithm (Section 3.3.3) would do nothing on this example, and the final output would contain all three clone pairs listed above. Note that the ideal group of three clones, with each clone consisting of a node labeled `3` and a node labeled `4`, should have been identified but was not. In essence, the problem is that when we have multiple corresponding fragments, matches between an individual pair of fragments may be larger than the global match, and may be different from matches between other pairs of fragments. This problem, to some extent, is unavoidable with our current approach. The solution to this problem would be to modify the approach to find groups of clones directly.

3.4.6 Time complexity of the approach

In this section we discuss the time complexity of the clone-detection algorithm. We assume that the PDGs, CFGs and ASTs of all procedures have already been built, and that each flow-dependence edge has already been marked *loop-carried* or *loop-independent*. We assume that hash table lookups can be done in constant time, and that the maximum nesting depth of any node in any AST is bounded by a constant.

The worst-case time complexity of the algorithm is $O(N^2E)$, where N is the total number of nodes in the given program, and E is the maximum number of edges in the PDG of any procedure in the program. We explain this result below, with reference to the procedures in Figures 3.3 and 3.4:

Procedure `FindAllClonePairs`: The initial step of partitioning all nodes into equivalence classes basically involves doing a depth-first search on each CFG in the program. This takes $O(N)$ time.

Then, procedure `FindAllClonePairs` calls procedure `FindAClonePair` $O(N^2)$ times. That procedure is basically a wrapper around procedure `GrowCurrentClonePair`, which we discuss next.

Procedure `GrowCurrentClonePair`: This procedure repeatedly removes node pairs from the `worklist`, and for each such pair (`node1`, `node2`), maps flow-dependence parents of `node1` to matching flow-dependence parents of `node2`, control-dependence parents of `node1` to matching control-dependence parents of `node2`, and control-dependence children of `node1` to matching control-dependence children of `node2` (if `node1` and `node2` are predicates).

We first consider the mapping of the control-dependence parents. Using hash tables whose keys are a combination of the expressions inside the predicates and the labels on the control-dependence edges, the control-dependence parents of `node1` can be mapped to the control-dependence parents of `node2` in time $O(p)$, where p is the total number of control-dependence parents of the two nodes.

Similarly, the control-dependence children (and flow-dependence parents) of `node1` and `node2` can be mapped to each other in time proportional to the total number of such children (parents).

In other words, for each pair of nodes (`node1`, `node2`) removed from the worklist, processing takes time proportional to the total number of PDG edges incident on these two nodes. Also, since each node is a member of at most one pair of nodes removed from the worklist, no PDG edge is visited more than twice during any invocation of procedure `GrowCurrentClonePair` (flow-dependence edges are visited only once; control-dependence edges are visited at most twice – once from the parent and once from the child). Therefore the worst-case time complexity of one invocation of this procedure is $O(E)$.

That means that the worst-case time complexity of the entire clone-detection algorithm is $O(N^2E)$.

Note that the *expected* number of edges in a PDG is at most Me, where M is the maximum number of nodes in any procedure (in any program), and e is the expected value of the average number of PDG edges incident on a node in a PDG. We expect M to be independent of N (the program size); in other words, we expect that procedure sizes have a constant upper bound, albeit some large constant. Furthermore, we expect e to be a small constant; in experiments we did using PDGs built by the tool CodeSurfer [Csu], for large example programs such as *bison* and *make*, we found that the average number of intra-procedural PDG edges per node in a program varied between 2.3 and 3.8.

Although the worst-case time bound of the algorithm is $O(N^2E)$, the running time in practice would be less than that, because the time complexity we have derived here does not take into account the heuristic that a pair of nodes that gets mapped in some clone pair be not used a root pair at all in the future (see Section 3.3.1.1). We expect that this heuristic will let the running time of the approach be much less than what the theoretical result of this section suggests. Chapter 7 presents actual running times of the implementation of

the algorithm on three real programs; those numbers indicate that in practice the running time of the tool grows faster than linearly with the size of the program, but much below a quadratic rate.

Chapter 4

Terminology for extraction algorithms

In this chapter we introduce terminology that is needed by the individual-clone and clone-group extraction algorithms (in addition to the terminology introduced in Chapter 2). A *block* is a subgraph of a CFG that corresponds to a single (simple or compound) source-level statement. Therefore there are several kinds of blocks: assignment, jump (one kind for each kind of jump), procedure call, `if`, `while`, and `do while`. Each block has a unique *entry* node that is the target of all non-jump edges whose sources are outside the block and whose targets are inside. Each block also has a unique *fall-through exit* node (outside the block) such that all non-jump edges whose sources are inside the block and whose targets are outside have this node as their target.

A *block sequence* b is a sequence of blocks $B_1, B_2, \ldots, B_n$ (where $n \geq 1$) such that the entry node of block B_i is the fall-through exit node of block B_{i-1}, for each i in the range 2 through n. The entry node of B_1 is the entry node of the entire block sequence b, while the fall-through exit node of B_n is the fall-through exit of b. Each of the blocks B_i is said to be a *constituent* of the block sequence b. Any block sequence obtained by dropping zero or more leading blocks and zero or more trailing blocks from b is said to be a *sub-sequence* of b.

A *maximal block sequence* is one that is not a sub-sequence of any other block sequence.

Nodes are *nesting children* of predicates in the usual sense (e.g., a node in the "then" part of an `if` statement is a *true*-nesting child of that `if` statement's predicate). Block sequences are nesting children of the blocks that contain them, and of the corresponding predicates. For example, the "then" and "else" parts of an `if` statement are maximal block sequences

that are the *true* and *false* nesting children, respectively, of the `if` block that corresponds to that `if` statement; these two block sequences are also nesting children of the `if` predicate that corresponds to that `if` statement. A loop-block has just one maximal block sequence as its nesting child – the block sequence that constitutes the body of the loop. Assignment, jump, and procedure-call blocks have no nesting children.

Example: Consider the second CFG fragment in Figure 2.1. The entire fragment is a `while` block. The region marked $\mathcal{H}$ plus the following `fscanf` statement is a maximal block sequence that is the *true* nesting child of the outer `while` block. j_2, e_2, the `fscanf` statement, and f_2 are the first four constituent blocks of this maximal block sequence. Of these blocks, f_2 is the only one that has a maximal block sequence nested inside it (the "then" part of the `if` statement). □

A *hammock* is a subgraph of a CFG that has a single *entry node* (a node that is the target of all edges from outside the hammock that enter the hammock), and from which control flows out to a single *fall-through exit node* (a node that is outside the hammock that is the target of all edges leaving the hammock). More formally, given a CFG G with nodes $\mathcal{N}(G)$ and edges $\mathcal{E}(G)$, a hammock in G is the subgraph of G induced by a set of nodes $H \subseteq \mathcal{N}(G)$ such that:

1. There is a unique entry node e in H such that:
   $(m \notin H) \wedge (n \in H) \wedge ((m, n) \in \mathcal{E}(G)) \Rightarrow (n = e)$.

2. There is a unique fall-through exit node t in $\mathcal{N}(G) - H$ such that:
   $(m \in H) \wedge (n \notin H) \wedge ((m, n) \in \mathcal{E}(G)) \Rightarrow (n = t)$.

An *e-hammock* (a hammock with exiting jumps) is a subgraph of a CFG that has a single *entry node*, and, if all jumps are replaced by no-ops, a single *fall-through exit node*; i.e., an e-hammock is a hammock that is allowed to include one or more exiting jumps (jumps whose targets are not inside the hammock and are not the hammock's fall-through exit node).

It can be shown that a CFG subgraph is an e-hammock iff the subgraph is a block sequence having the additional property that its entry node is the target of all incoming

jump edges (those whose sources are outside the block sequence). The block sequence is a hammock if, additionally, all outgoing jump edges go to its fall-through exit node.

Example: Every block sequence (including the non-maximal ones) in the second CFG in Figure 2.1 is an e-hammock; e.g., the circled block sequence labeled $\mathcal{H}$. The two blocks "`fscanf(..,&hours)`" and f_2 together form a block sequence that is a hammock. □

The following definitions are adapted from [KKP+81].

Definition 4 (Anti dependence) A node p is *anti dependent* on a node q iff q uses some variable v, p defines v, and there is a path P in the CFG from q to p that involves no non-executable edges. We say that this anti dependence is induced by path P.

Definition 5 (Output dependence) A node p is *output dependent* on a node q iff both p and q define some variable v, and there is a path P in the CFG from q to p that involves no non-executable edges. We say that this output dependence is induced by path P.

Flow, anti, and output dependences are collectively known as *data dependences.* A data dependence between two nodes can be induced by more than one path, and one or more kinds of data dependences may exist between two nodes.

Chapter 5

Individual-clone extraction algorithm

In this chapter we present the individual-clone extraction algorithm, which can be summarized as follows:

Given: The set of nodes that are to be extracted into a separate procedure (the nodes in a single clone), as well as the CFG of the procedure that contains these nodes. The given nodes are referred to in this chapter as the *marked* nodes.

Do: Find the smallest e-hammock (single-entry CFG subgraph that corresponds to a sequence of source-level statements) that contains the marked nodes and that contains no *backward exiting jumps* (defined in Section 5.1). Transform this e-hammock in a semantics preserving manner such that:

- As many of the unmarked nodes in the e-hammock as possible are moved out of the e-hammock, and
- The e-hammock becomes a hammock (which is a single-entry single-outside-exit structure).

The transformation done by the individual-clone algorithm is illustrated using the schematic in Figure 5.1. Part (a) of that figure shows the original clone (the shaded nodes are the marked nodes); notice that the e-hammock of the clone (the first four nodes) contains an unmarked node, in addition to the marked nodes. In other words, the clone is non-contiguous. The e-hammock also contains an exiting jump.

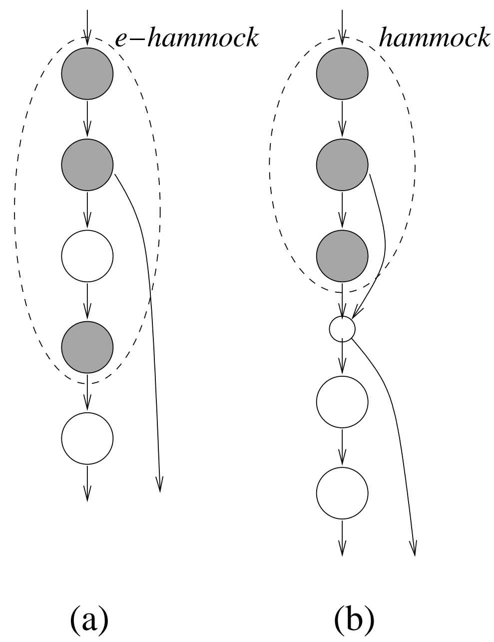


Figure 5.1 Transformation done by the individual-clone algorithm

Non-contiguous clones are a problem because it is not clear which of the several "holes" that are left behind after the marked nodes are removed should contain the call to the new procedure (in Figure 5.1(a) there were two such holes if the marked nodes were removed). Exiting jumps are a problem, too; a clone that involves an exiting jump cannot be extracted as such because, after extraction, control returns from the new procedure to a single node in the remaining code (the node that immediately follows the call). Figure 5.1(b) contains the transformed output of the algorithm. The intervening unmarked node has been moved out of the way of the marked nodes (in general only the intervening unmarked nodes that can be moved out without affecting semantics are moved out; the others are left behind in the e-hammock of the clone). The e-hammock has also been converted into a hammock, which, being a single-entry single-outside-exit structure, is easy to replace by a call; this conversion is done by converting the exiting jump into a non-exiting jump to the "fall-through exit" of the clone, and by placing a new copy of the exiting jump outside the clone, controlled by an appropriate condition.

The algorithm runs in polynomial time (in the size of the e-hammock that contains the marked code), always succeeds in converting the e-hammock into a hammock (which is not the case for some previous approaches), and is provably semantics preserving (proofs are given in Appendices A and B). It performs the following steps:

Step 1:

Find the smallest e-hammock $\mathcal{H}$ that contains the marked nodes, and contains no *backward exiting jumps*. Because the marked nodes can be non-contiguous, the e-hammock can contain unmarked nodes in addition to marked nodes. Also, the e-hammock can contain exiting jumps (whenever we say "exiting jump" in this chapter, we mean an exiting jump of the e-hammock identified in this step).

(Note: the algorithm transforms $\mathcal{H}$, leaving the rest of the CFG unchanged.)

Step 2:

Determine a set of ordering constraints among the nodes in the e-hammock based on data dependences, control dependences, and the presence of exiting jumps.

Step 3:

Promote any unmarked nodes in the e-hammock that cannot be moved out of the way of the marked nodes without violating ordering constraints. The promoted nodes will be present in the extracted procedure in guarded form (as indicated in the example in Figure 1.4(b)).

From this point on, the promoted nodes are regarded as marked.

Step 4:

Partition the nodes in the e-hammock into three "buckets": *before*, *marked*, and *after*. The *marked* bucket contains all the marked nodes. The *before* and *after* buckets contain intervening unmarked nodes that were moved out of the way. Nodes that are forced by some constraint to precede some node in the *marked* bucket are placed in the *before* bucket; nodes that are forced by some constraint to follow some node in the *marked* bucket are placed in the *after* bucket; each other intervening node is placed arbitrarily in *before* or *after*.

An assignment or procedure-call node in the e-hammock is assigned to exactly one of the three buckets during the partitioning. However, whenever a node is placed in a bucket, all its control-dependence ancestors in $\mathcal{H}$ are also placed in the same bucket; if those ancestors (predicates or jumps) are already present in other buckets, the algorithm creates new copies for the current bucket. In other words, predicates and jumps may be duplicated (therefore, strictly speaking, this step *partitions* only the set of non-predicate and non-jump nodes). However, any individual bucket will contain only one copy of any node (a bucket is a set of nodes).

Step 5:

Create three e-hammocks from the nodes in the *before*, *marked*, and *after* buckets, respectively. Let the relative ordering of nodes within each e-hammock be the same as in the original e-hammock $\mathcal{H}$. String together the *before*, *marked*, and *after* e-hammocks, in that order, to create a new (composite) e-hammock $\mathcal{O}$; do this by using the entry node of the *marked* e-hammock as the fall-through exit node of the *before* e-hammock, and using the *entry* node of the *after* e-hammock as the fall-through exit node of the *marked* e-hammock.

Step 6:

Convert the *marked* e-hammock (which is now a part of the composite e-hammock $\mathcal{O}$) into a hammock by converting all exiting jumps in it to `gotos` whose targets are the entry node of the *after* e-hammock, and by placing compensatory code in the beginning of the *after* e-hammock. The composite e-hammock $\mathcal{O}$, after this conversion, is the output of the algorithm. Finally, replace the original e-hammock $\mathcal{H}$ in the CFG with the new e-hammock $\mathcal{O}$ to obtain a resultant program that is semantically equivalent to the original, and from which the marked nodes are extractable (because they form a hammock).

Example: Consider the two CFGs in Figure 2.1, which correspond to the two fragments in Figure 1.1. Each clone is indicated using shaded nodes. The e-hammock of each clone (i.e., $\mathcal{H}$) that is identified in Step 1 of the algorithm is indicated using a dashed oval. The two corresponding output e-hammocks $\mathcal{O}$ produced by the algorithm are shown in Figure 5.2; each dashed oval here indicates the "marked" hammock, while the fragments before and after this oval are the "before" and "after" e-hammocks, respectively. □

The rest of this chapter is organized as follows. The six steps in the algorithm are described, respectively, in Sections 5.1 through 5.6. Section 5.7 summarizes the features of the algorithm, while Section 5.8 discusses its worst-case complexity.

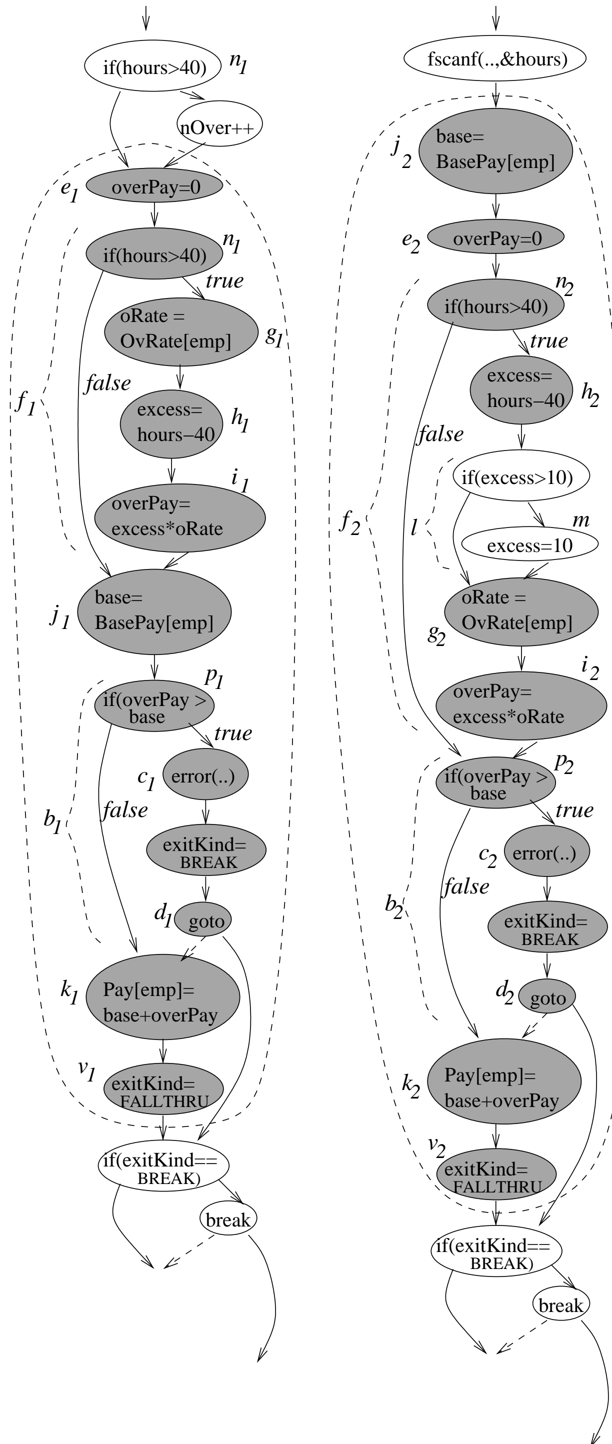


Figure 5.2 Result ($\mathcal{O}$) of applying individual-clone algorithm on each clone in Figure 2.1. Each dashed oval is the “marked” hammock; the fragments above and below the ovals are the “before” and “after” e-hammocks, respectively.

5.1 Step 1: find the smallest e-hammock containing the clone

This step identifies the smallest e-hammock $\mathcal{H}$ that contains the marked nodes, and contains no *backward* exiting jumps – exiting jumps whose targets are postdominated by the entry node of $\mathcal{H}$. The algorithm for finding this e-hammock is given in Figure 5.3 (we discuss later the reason for disallowing backward exiting jumps). The algorithm is based on the fact that every e-hammock is a block sequence. We start by assigning all marked nodes to a set `included`, and finding the most deeply nested, shortest block sequence `sequence` that completely contains `included`. If `sequence` includes no nodes (besides the entry node) that are targets of outside jumps, and contains no backward exiting jumps, then we stop (`sequence` is the e-hammock we seek). Otherwise, we add the offending outside jumps as well as the targets of the backward exiting jumps to the set `included`, and find the mostly deeply nested, shortest block sequence that contains (the newly updated set) `included`. This process continues until the block sequence in hand (`sequence`) is an e-hammock that contains no backward exiting jumps.

Example: We trace the algorithm in Figure 5.3 on the first clone in Figure 2.1. *included* initially contains all the marked nodes (the shaded nodes). The most deeply nested, shortest block sequence that contains all the *marked* nodes is $[e_1, f_1, j_1, b_1, k_1]$. In the first `for` loop (lines 4-6) the node "`nOver++`" gets added to *included*. Then `entry` gets set to e_1. Nothing gets added to *included* in the second `for` loop (lines 9-17); there are no edges coming into any of the *included* nodes from outside except to e_1, and the target of d_1 (the only exiting jump) is not postdominated by e_1. Therefore $\mathcal{H}$ is equal to the *included* nodes, as indicated by the dashed oval in Figure 2.1. □

The left column of Figure 5.4 has an example that illustrates the need to disallow backward exiting jumps. The marked nodes are indicated by the "`++`" signs. Notice that the intervening unmarked node "`n++`" cannot be moved after all the marked nodes, because there is a flow dependence from "`n++`" to the marked node "`avg = sum / n`". Therefore,

```
1: included nodes = marked nodes
2: repeat
3:    Find the most deeply nested, shortest block sequence sequence that contains all the
      included nodes.
4:    for all constituent blocks c of sequence do
5:       Add all nodes in c to included.
6:    end for
7:    entry = entry node of the first constituent block of sequence
8:    done = true
9:    for all included nodes v do
10:      if (v != entry) and (there is a CFG edge from some non-included node s to v)
         then
11:         Add s to included. Set done = false.
12:      end if
13:      if (v is a jump node) and (the true target t of v is non-included) and (entry
         postdominates t) then
14:         Add t to included. Set done = false.
15:      end if
16:   end for
17: until done
18: Smallest containing e-hammock H = included nodes.
```

Figure 5.3 Algorithm to find the smallest e-hammock that contains the marked nodes

Original Fragment $\mathcal{H}$	Output $\mathcal{O}$
`  L: k++;`	`      n++;`
`++    sum = sum + k;`	`++ L: k++;`
`++    if(k < 10)`	`++    sum = sum + k;`
`++      goto L;`	`++    if(k < 10)`
`      n++;`	`++      goto L;`
`++    avg = sum / n;`	`++    avg = sum / n;`

Figure 5.4 Example illustrating backward exiting jumps

"`n++`" can only be moved before all the marked nodes. Notice also that the first three statements in the example – "`k++`", "`sum = sum + k`", and "`if(k < 10) goto L`" – form a loop. Therefore, if we move "`n++`" to just before the first marked node, "`sum = sum + k`", we would be moving it from its original location outside the loop to inside the loop, which is an incorrect transformation. The correct transformation is to move "`n++`" to before "`k++`", therefore keeping it outside the loop. If the algorithm did not have the no-backwards-jumps requirement, the smallest e-hammock found by Figure 5.3 would only include the statements "`sum = sum + k`" through "`avg = sum / n`". "`n++`" would then be moved out to the *before* e-hammock, i.e., before "`sum = sum + k`" (which would be the first node in the *marked* hammock). In other words, "`n++`" would be moved between "`k++`" and "`sum = sum + k`". This, as we noted earlier, is an incorrect transformation.

We now illustrate how the algorithm does the correct transformation as a result of the no-backwards-exiting jumps requirement. The algorithm in Figure 5.3, when applied to this example, initially puts the statements from "`sum = sum + k`" through "`avg = sum / n`" into `sequence`. It then adds "`k++`" to `sequence` in lines 13-15 (because "`k++`" is the target of the `goto` and is postdominated by the current entry node "`sum = sum + k`"). Therefore, the e-hammock finally identified in Step 1 is the entire fragment shown in the left column of Figure 5.4.

Subsequently, Step 3 (described in Section 5.3) promotes "`k++`" (but not "`n++`"). Therefore "`k++`" becomes the first node in the *marked* hammock. Then, "`n++`" is moved to the *before* e-hammock (i.e., to before "`k++`"). The final result of the algorithm (which is semantically equivalent to the original) is shown in the right column of Figure 5.4.

5.2 Step 2: generate ordering constraints

This step is the heart of the extraction algorithm; it determines constraints among the nodes in $\mathcal{H}$ based on data dependences, control dependences and the presence of exiting jumps. The constraints generated are of three forms: "$\leq$" constraints, "$\Rightarrow$" constraints, and "$\leadsto$" constraints. Each constraint involves two nodes in $\mathcal{H}$ (the meanings of the three kinds of constraints are given in Figure 5.5). The constraints are used in Step 3 to determine which unmarked nodes must be promoted; they are also used in Step 4 to determine how to partition the remaining unmarked nodes between the *before* and *after* buckets, while preserving data and control dependences, and therefore the original semantics.

The constraints are generated in two steps. In the first step "base" constraints are generated, using the rules in Figure 5.5. In the second step extended constraints are generated from the base constraints, as described in Figure 5.6. The extended constraints are implied by base constraints, but must be made explicit in order for Step 3 (promotion) and Step 4 (partitioning of unmarked nodes) to work correctly. Each rule in Figure 5.6 specifies the pre-conditions on the left hand side of the "$\vdash$", and the corresponding extended constraint that is generated on the right hand side.

The following subsections explain the (base and extended) constraints-generation rules of Figures 5.5 and 5.6, categorized by their reason of generation (data dependences, control dependences, or presence of exiting jumps).

5.2.1 Data-dependence-based constraints

The first rule in Figure 5.5 and the first rule in Figure 5.6 both pertain to data dependences. The essential idea is that if a node n is data dependent on a node m, then no copy

1. Data-dependence constraints: For each pair of nodes m, n in $\mathcal{H}$ such that n is data (i.e., flow, anti, or output) dependent on m, and such that the data dependence is induced by a path contained in $\mathcal{H}$, generate the constraint $m \leq n$. This means that (a copy of) m must not be placed in any bucket that follows a bucket that contains (a copy of) n (recall that the order of the buckets is *before*, *marked*, *after*).

2. Control-dependence constraints: For each node n in $\mathcal{H}$, and for each predicate or jump p in $\mathcal{H}$ such that n is (directly or transitively) control dependent on p in the original CFG, generate a constraint $n \Rightarrow p$. This means that (a copy of) p must be present in each bucket that contains (a copy of) n.

3. Antecedent constraints: For each node n in $\mathcal{H}$ that is neither a predicate nor a jump and for each exiting jump j in $\mathcal{H}$ such that there is a path in $\mathcal{H}$ (ignoring non-executable edges) from n to j generate a constraint $n \rightsquigarrow j$. This means two things:

    - if n is in the *after* bucket then a copy of j must be included in the same bucket.
    - if n but not j is in the *marked* bucket then a copy of j must be included in the *after* bucket.

Figure 5.5 Rules for generating base ordering constraints

Apply the following rules repeatedly until no more extended constraints can be generated:

1. $a \leq b, b \leq c \vdash a \leq c$.
2. $p \leq b, a \Rightarrow p \vdash a \leq b$.
3. $b \leq p, a \Rightarrow p \vdash b \leq a$.
4. $n \rightsquigarrow j, j \leq m \vdash n \leq m$.

Figure 5.6 Generation of extended ordering constraints

of m should be present in a bucket that comes after a bucket that contains a copy of n. This, together with the fact that the relative ordering of nodes within any result e-hammock (*before*, *marked*, or *after*) is the same as in the original e-hammock $\mathcal{H}$ (see Section 5.5), ensures that any node n is flow/anti/output dependent on a node m in the output of the algorithm iff it is flow/anti/output on node m in the original program. This property is an important aspect of our sufficient condition to guarantee semantics preservation.

Example: Consider the second clone in Figure 2.1. One of the data-dependence constraints generated for that example is: "`fscanf(..,&hours)`" $\leq$ "`if(hours > 40)`" (due to a flow dependence). This constraint forces the `fscanf` statement to be placed in the *before* bucket in Step 4, since the `fscanf` statement is an unmarked node and the `if` predicate is marked. □

5.2.2 Control-dependence-based constraints

The second rule in Figure 5.5 generates base control-dependence constraints. These constraints, together with the fact that in the resultant CFG produced by the algorithm the relative ordering of nodes within any e-hammock (*before*, *marked*, or *after*) is the same as in the original e-hammock $\mathcal{H}$, ensure that control dependences in the original code are preserved; this too is an important aspect of semantics preservation.

Example: Consider the first clone in Figure 2.1. One of the control-dependence-based constraints generated for this clone is "`nOver++`" $\Rightarrow$ "`if(hours > 40)`", which says that a copy of the predicate "`if(hours > 40)`" must be placed in the same bucket as "`nOver++`". Since the `if` predicate is also a control-dependence parent of several marked nodes, a copy will also be placed in the *marked* bucket. This is the reason for the duplication of the `if` predicate in the algorithm output shown in Figure 5.2.

The `if` needs to be present in the same bucket as the node "`nOver++`" to ensure that this node executes only in those iterations of the loop in which "`hours > 40`" is true (otherwise this node would execute in every iteration of the loop, which is not the original semantics). □

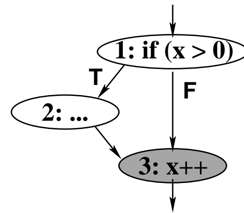


Figure 5.7 Example illustrating control-dependence-based extended constraints

Control dependence is also used to generate extended constraints. The second and third rules in Figure 5.6 are the pertinent rules, and we illustrate them using the example in Figure 5.7. Assume that node 3 is marked, nodes 1 and 2 are unmarked, and node 2 does not involve the variable `x`. The base constraints that are generated are $1 \leq 3$ (due to anti dependence), and $2 \Rightarrow 1$ (due to control dependence). The second rule in Figure 5.6 therefore applies, yielding the extended constraint $2 \leq 3$. This constraint makes intuitive sense, given the meanings of the two base constraints that were used to produce it. Because node 3 is marked, this constraint forces the algorithm (in Step 4) to place node 2 (and a copy of node 1) in the *before* bucket. This is the correct outcome.

If the extended constraint $2 \leq 3$ were not produced, then node 2 would be unconstrained. Therefore it could be placed in the *after* bucket in Step 4, which would be followed by an assignment of a copy of node 1 to the same bucket (due to $2 \Rightarrow 1$). This is a violation of the base constraint $1 \leq 3$ (node 3 is in the *marked* bucket); therefore, the algorithm would have to undo the assignments of nodes 1 and 2 to *after* (by backtracking). The extended-constraints rules allow the algorithm to avoid backtracking.

5.2.3 Exiting-jumps-based constraints

The final rules in both Figures 5.5 and 5.6 are based on the presence of exiting jumps. Before describing these rules, we present an example in Figure 5.8 that illustrates the intricacies in handling exiting jumps. The left column in the figure is a fragment of code, with the marked nodes indicated by the "`++`" signs (the surrounding loop, to which the `break` pertains, is not shown). The data- and control-dependence-based constraints generated for

Original Fragment $\mathcal{H}$	Incorrect Output	Correct Output $\mathcal{O}$

`++ x = 0;`
`   y = x;`
`   if (p)`
`     break;`
`   a = 1;`
`++ b = a;`
	`   if (p)`
	`     break;`
	`   a = 1;`
	`++ x = 0;`
	`++ if (p)`
	`++   break;`
	`++ b = a;`
	`   y = x;`
		`   if (p)`
		`     goto L1;`
		`   a = 1;`
		` L1:`
		`++ x = 0;`
		`++ if (p)`
		`++   goto L2;`
		`++ b = a;`
		` L2:`
		`   y = x;`
		`   if (p)`
		`     break;`

Figure 5.8 Example illustrating handling of exiting jumps

this example are "`x = 0`" $\leq$ "`y = x`", "`a = 1`" $\leq$ "`b = a`" (both due to flow dependences), and "`a = 1`" $\Rightarrow$ "`break`", "`a = 1`" $\Rightarrow$ "`if (p)`", "`b = a`" $\Rightarrow$ "`break`", "`b = a`" $\Rightarrow$ "`if (p)`" (all due to control dependences; recall that the `break` is a pseudo-predicate, which means that the two nodes following it are control dependent on it). The middle column shows the output of the algorithm *if* it generated no exiting-jumps-based constraints and did no other special processing of exiting jumps. Note that "`a = 1`" and "`y = x`" have been moved out to the *before* and *after* e-hammocks respectively; and copies of the `if` predicate and the `break` have been placed both in *before* and in *marked* because "`a = 1`" (in *before*) and "`b = a`" (in *marked*) are control dependent on them. This output, however, is incorrect: whenever "`p`" is *true* in the initial state, none of the assignment nodes would be reached, whereas in the original code "`x = 0`" and "`y = x`" would be reached.

Before we discuss the correct solution, we introduce a definition.

Definition 6 (Antecedent of an exiting jump) An *antecedent* of an exiting jump j in $\mathcal{H}$ is any node n such that n is not a predicate or jump and such that there is a path in $\mathcal{H}$ involving no non-executable edges from n to j.

Informally speaking, an antecedent of an exiting jump is a node that can be reached in an execution of $\mathcal{H}$ before control reaches the exiting jump.

Returning to the example in the middle column of Figure 5.8, the problem is as follows: A copy of the `break` was placed in the *before* e-hammock because "`a = 1`" (which is in *before*) is control dependent on it; however, this `break` ends up bypassing its antecedents "`x = 0`" and "`y = x`" (in the *marked* and *after* e-hammocks, respectively), although its purpose is to bypass "`a = 1`" only. Similarly, the copy of the `break` in the *marked* hammock incorrectly bypasses its antecedent "`y = x`" in the *after* e-hammock, although its purpose is to bypass "`b = a`" only. The solution we adopt is based on the following rule:

> *Rule for exiting jumps:* Let j be any exiting jump in $\mathcal{H}$, let n be any antecedent of j, and let B be the e-hammock of $\mathcal{O}$ (B is *before*, *marked* or *after*) that contains n. A copy of j is needed either in B, or in some e-hammock of $\mathcal{O}$ that

follows B. The last copy of j remains an exiting jump, but each previous copy is converted into a `goto` whose target is the fall-through exit of the e-hammock that contains that copy. This `goto` will (correctly) bypass subsequent nodes within that e-hammock that were originally control dependent on j, but will not bypass n.

The final column of Figure 5.8 illustrates the above rule. Notice that a copy of the `break` is placed in the *after* e-hammock even though this e-hammock contains no nodes that are control dependent on the `break`; the reason for this is that "`y = x`" is an antecedent of the `break`.

The exiting-jumps-based constraints (in Figures 5.5 and 5.6) can now be explained. The final rule in Figure 5.5 is a direct consequence of the "Rule for exiting jumps" defined above. The final rule in Figure 5.6 is based on the following reasoning:

1. $n \rightsquigarrow j$ implies that a copy of j is needed either in n's bucket or in some bucket that follows n's bucket.

2. $j \leq m$ implies that m should not be present in any bucket that precedes a bucket that contains j.

3. the above two points imply that m should not be present in any bucket that precedes the bucket that contains n; i.e., $n \leq m$.

5.3 Step 3: promote unmovable unmarked nodes

Figure 5.9 gives the procedure for promoting nodes. The first rule in that figure follows intuitively from the meaning of a "$\leq$" constraint. Consider the third rule. Recall that the constraint $m \Rightarrow p$ means that *a copy* of p is required in the same bucket as m (i.e., in the *marked* bucket); this constraint does *not* disallow copies of p from being present in other buckets. In spite of this it makes sense to promote p, because this promotion might lead us to discover that some other unmarked node r cannot be moved out of the way of the marked

Apply the following rules repeatedly, in any order, until no more nodes can be promoted:

1. If there exist constraints $m_1 \leq n$ and $n \leq m_2$, such that n is unmarked and m_1, m_2 are marked, promote n.
2. If there exist constraints $m_1 \leadsto j$ and $j \leq m_2$, such that j is unmarked and m_1, m_2 are marked, promote j.
3. If there exists a constraint $m \Rightarrow p$ such that m is marked and p is unmarked, promote p.

A promoted node is regarded as marked as soon as it is promoted.

Figure 5.9 Procedure for promoting nodes

nodes and needs to be promoted (e.g., there might exist constraints $m_3 \leq r$ and $r \leq p$, where m_3 is a marked node). The second rule in Figure 5.9 is based on the following reasoning:

1. m_1 in the *marked* bucket and $m_1 \rightsquigarrow j$ means that a copy of j is needed either in the *marked* bucket or in the *after* bucket.

2. m_2 in the *marked* bucket and $j \leq m$ means that j should not be placed in the *after* bucket (i.e., it must be in the *before* or *marked* bucket).

3. The above two points imply that a copy of j is needed in the *marked* bucket. Therefore, repeating our earlier argument, j needs to be promoted.

Example: Consider the second clone in Figure 2.1. Two of the data-dependence constraints in this example are "`excess = hours-40`" $\leq$ "`excess=10`" (due to output dependence), and "`excess=10`" $\leq$ "`overPay = excess*oRate`" (due to flow dependence). These two constraints cause "`excess=10`" to be promoted (by the first rule in Figure 5.9). This in turn causes the predicate "`if(excess > 10)`" to be promoted (due to the constraint "`excess=10`" $\Rightarrow$ "`if(excess > 10)`"). The remaining unmarked node "`fscanf(..&hours)`" does not get promoted. □

5.4 Step 4: partition nodes into buckets

We have so far discussed informally how a node can be forced by the constraints into a particular bucket; this notion is formalized in Figure 5.10. The procedure for partitioning nodes into buckets is given in Figure 5.11. The procedure is iterative; it assigns *forced* nodes to their buckets whenever possible, and arbitrarily selects unforced nodes and assigns them to arbitrary buckets when no forced nodes are available.

We note again that since predicates and jumps can be placed in multiple buckets, this step, strictly speaking, partitions only the set of non-predicate and non-jump nodes.

Example: Consider the first clone in Figure 2.1. All the *marked* nodes are first assigned to the *marked* bucket. No nodes are subsequently forced. Therefore the unmarked node

A node r is forced into the *before* bucket if r is not in the *marked* bucket and any of the following conditions hold:

B1: there exists a constraint $r \leq b$, where b is a node in the *before* bucket.

B2: there exists a constraint $r \leq m$, where m is a node in the *marked* bucket.

A node s is forced into the *after* bucket if any of the following conditions A1 through A4 hold. Conditions A1-A3 are applicable only if s is not in the *marked* bucket; A4 is applicable even if s is in the *marked* bucket.

A1: there exists a constraint $a \leq s$, where a is a node in the *after* bucket.

A2: there exists a constraint $m \leq s$, where m is a node in the *marked* bucket.

A3: there exists a constraint $m \leadsto s$, where m is a node in the *marked* bucket.

A4: there exists a constraint $a \leadsto s$, where a is a node in the *after* bucket.

Figure 5.10 Rules for forced assignment of nodes to buckets

Place each marked (and promoted) node in the *marked* bucket. Then, partition the unmarked nodes into *before* and *after*:

1: **repeat**
2: **if** there exists at least one node that is *forced* to be in one of the two buckets *before* or *after* (as defined in Figure 5.10), and is not already in that bucket **then**
3: Let n be an arbitrarily chosen forced node, and let B be the bucket into which n is forced. Assign n to B (make a fresh copy in case n is already in another bucket)
4: **else if** there exists at least one node that is not a "normal" predicate (i.e., not an `if`, `while`, or `do-while` predicate) and that is not in any bucket, including the *marked* bucket **then**
5: Let n be an arbitrarily chosen unmarked non-normal-predicate that is not in any bucket. Assign n to one of the two buckets *before* or *after*, chosen arbitrarily.
6: **end if**
7: If a node n was assigned in the previous `if` statement to a bucket, then place copies of predicates and jumps in $\mathcal{H}$ on which n is (directly or transitively) control dependent in the same bucket as n.
8: **until** no node was assigned to any bucket in the current iteration

Figure 5.11 Procedure for partitioning nodes into buckets

"`nOver++`" is arbitrarily placed in a bucket, say *before*. This causes a copy of its controlling predicate "`if(hours > 40)`" to be also placed in that bucket. That completes the partitioning of this clone.

Consider next the second clone in Figure 2.1. Here, after the marked nodes (including the two promoted nodes "`if(excess > 10)`" and "`excess = 10`") are assigned to the *marked* bucket, there does exist a forced node: the unmarked `fscanf` node is forced by the flow-dependence constraint "`fscanf(..,&hours)`" $\leq$ "`if(hours>40)`" (and by other constraints) into the *before* bucket. This unmarked node has no control-dependence ancestors in $\mathcal{H}$, therefore no predicate is simultaneously placed in *before*. That completes the partitioning of this clone. □

5.5 Step 5: create output e-hammock $\mathcal{O}$

The first thing in this step is to convert each of the three buckets into its corresponding e-hammock. A bucket B is converted into its corresponding e-hammock by making a copy of $\mathcal{H}$ and removing from that copy non-B nodes (nodes that are not in B). A non-B node in $\mathcal{H}$ that has no incoming edges from B-nodes (nodes in B) is removed from the copy of $\mathcal{H}$ simply by deleting it and all its incident edges. Considering the other case, let t be any non-B node that *does* have incoming edges from B-nodes in $\mathcal{H}$. It can be shown that t satisfies the following two properties (see Lemma B.3 and its proof in Appendix B):

1. At most one B-node in $\mathcal{H}$ can be reached *first* (i.e., without going through other B-nodes) along paths in $\mathcal{H}$ starting at t.

2. Moreover, if a B-node h can be reached by following paths in $\mathcal{H}$ from t, then there is no path from t that leaves $\mathcal{H}$ without going through h.

If no B-node in $\mathcal{H}$ is reachable from t, then t is removed from the copy of $\mathcal{H}$ by redirecting all edges entering it to the fall-through exit of this copy. On the other hand, if a unique B-node h is reachable from t in $\mathcal{H}$, then all edges entering t in the copy are redirected to h.

The entry node of the copy (which will be an e-hammock) is the entry node e of $\mathcal{H}$, if e is a B-node, else it is the unique B-node in $\mathcal{H}$ that is first reached from e.

As a result, if in any execution of $\mathcal{H}$ control flows through some B-node m, then through some sequence of non-B nodes, then through another B-node t, then in the created e-hammock corresponding to B the immediate CFG successor of m is t. In other words, considering the nodes in B, the relative ordering of these nodes within the created e-hammock corresponding to B is the same as it is in $\mathcal{H}$. This property, together with the property that the partitioning of nodes into buckets in Step 4 satisfies all constraints, is sufficient to guarantee semantics preservation.

After the three result e-hammocks are created (as described above), they are strung together in the order *before*, *marked*, *after* to obtain the e-hammock $\mathcal{O}$.

Example: Consider the first clone in Figure 2.1. At the end of the previous step, the *before* bucket contains "`nOver++`" and n_1, the *marked* bucket contains all the shaded nodes in that figure, and the *after* bucket is empty. Creating the *before* e-hammock consists of creating a copy of $\mathcal{H}$, and removing every node from that copy except for "`nOver++`" and n_1. As a result n_1 becomes the entry node of the *before* e-hammock; the *true* edge out of n_1 goes to "`nOver++`" (which is the unique *before* node reachable from g_1 in $\mathcal{H}$); the *false* edge out of n_1 as well as the edge out of "`nOver++`" go to the fall-through exit of the *before* e-hammock (because no *before* nodes are reachable from those two edges in $\mathcal{H}$). Creating the *marked* e-hammock involves removing only the node "`nOver++`". The result e-hammock $\mathcal{O}$ (as it is at the end of the next step, Step 6) is shown within the dashed oval in Figure 5.2. □

5.6 Step 6: convert *marked* e-hammock into a hammock

Exiting jumps are now processed specially, as described below. If copies of an exiting-jump node j of $\mathcal{H}$ are present in multiple result e-hammocks (*before, marked, after*), then each copy except the last one is converted into a `goto` whose target is the fall-through exit of the e-hammock that contains that copy (see "Rule for exiting jumps" in Section 5.2.3).

Additionally, if the *marked* e-hammock contains the last copy of an exiting jump j, then the following are done: This copy is converted into a `goto` whose target is the entry of the *after* e-hammock. An assignment "`exitKind =` *enc*" is inserted just before this new `goto`, where *enc* is a literal that encodes the kind of j (`break`, `continue`, `return`, or `goto`). In case j is a `goto` or `return`, and there are multiple exiting jumps in the *marked* e-hammock of the same kind as j (`goto` or `return`), then *enc* additionally encodes which `goto`/`return` j is; this is needed because different `gotos` can have different targets, and different `returns` can have different return expressions. A new assignment "`exitKind = FALLTHRU`" is placed in the *marked* e-hammock, at its end (i.e., all edges from within the *marked* e-hammock whose targets were the fall-through exit of that e-hammock are redirected to this new assignment, and a CFG edge is added from this assignment to that fall-through exit). Finally, the following new compensatory code is placed at the entry of the *after* hammock (as the target of the newly obtained `goto`): an `if` statement of the form "`if (exitKind ==` *enc*`)` *jump*", where *jump* is a copy of the exiting jump j. All told, these activities have the following effect: every exiting jump in the *marked* e-hammock is converted into a jump to the fall-through exit of that e-hammock, thereby changing this e-hammock into a hammock. The assignments to `exitKind` and the compensatory code are added to "undo" the change in semantics caused by the jump conversion; in other words, the behavior of the converted *marked* hammock together with the compensatory code is the same as the behavior of the unconverted *marked* e-hammock.

Note that copies of a `goto` node in $\mathcal{H}$ may be present in more than one of the created e-hammocks, with each copy having a different target. In that case, unique labels will need to be supplied for each copy during conversion of $\mathcal{O}$ into actual source code (but this is straightforward).

The algorithm is now finished. When the *marked* hammock is extracted out into a separate procedure all `gotos` in this hammock whose targets are the fall-through exit of this hammock are simply converted into `returns`.

Example: Consider again the first clone in Figure 2.1. The `break` is present in only one e-hammock – the *marked* e-hammock. This being the last copy of the `break`, it is converted into a `goto` whose target is the entry of the *after* e-hammock. The assignments to `exitKind` are then introduced in the *marked* hammock, and compensatory code is introduced in the *after* e-hammock. The final result is shown in Figure 5.2. In Figure 5.2 the *marked* hammock is indicated with the dashed oval, while the fragments preceding and following this oval are the *before* and *after* e-hammocks, respectively. □

5.7 Summary

The algorithm described in this chapter combines the techniques of statement reordering, promotion, and predicate duplication to extract difficult clones. The idea is to use statement reordering to move as many unmarked nodes that intervene between marked nodes as possible into the *before* and *after* buckets; only the unmarked nodes that cannot be moved away while satisfying the ordering constraints are promoted. Predicate duplication is tied in with reordering and happens indirectly: whenever a node is placed in a bucket, all its control-dependence ancestors are placed in the same bucket (even if they are already present in other buckets).

The goal behind this strategy is to deal with as many unmarked nodes as possible using movement (i.e., reordering) and to minimize promotions; this is a good thing, because it reduces the amount of guarded non-clone code in the extracted procedure. The algorithm never fails; i.e., it always succeeds in making the marked nodes form a hammock (it promotes as many nodes as are necessary to avoid failure).

Our key contribution in the context of this algorithm is the rules for generating ordering constraints. The constraint-generation rules take into account not just data and control dependences, but also the presence of exiting jumps, which have not been handled in previously reported approaches for the same problem. The constraints generated have the following desirable properties:

- No node can be forced (in Step 4 of the algorithm) into *both* the *before* and the *after* buckets by the constraints. (As many nodes as needed are promoted, by the promotion rules, to guarantee that this never happens.)

- The constraints are "complete"; i.e., if at any point in Step 4 there is no node available that is forced by the constraints, then *any* one of the remaining unassigned nodes n can be selected and assigned to *either* of the two buckets *before* or *after*, with the guarantee that the remaining nodes can be partitioned without violating any constraints, without a need to backtrack to n to try the other choice (bucket) for it. The absence of backtracking in the algorithm allows it to have worse-case time complexity that is polynomial in the size of the e-hammock $\mathcal{H}$ (details in Section 5.8).

- Any partitioning of the nodes in the original e-hammock $\mathcal{H}$ into *before*, *marked*, and *after* that satisfies all constraints is provably semantics preserving (see proof in Appendix B).

  In general there may be many partitionings that satisfy all constraints. Step 4 of the algorithm finds one such partitioning, by making arbitrary choices when nothing is forced. We prove in Appendix A that the partitioning found by the algorithm indeed satisfies all constraints.

5.8 Complexity of the algorithm

The worst-case time complexity of the individual-clone algorithm is $O(n^2V + n^3)$, where n is the number of nodes in the smallest e-hammock that contains the marked nodes, and V is the number of variables used and/or defined in the e-hammock. We derive this result in the rest of this section by discussing each step of the algorithm. We assume that the CFGs and Abstract Syntax Trees (ASTs) of all procedures are already available; we also assume that the *use* and *def* sets and control-dependence ancestors of all nodes are pre-computed. We assume that hash table lookups take constant time. The derivation makes use of the fact that the number of edges adjacent on any node in a CFG is bounded by a constant.

Step 1: This step finds the smallest e-hammock that contains the marked nodes. Figure 5.3 gives the procedure for this step. This procedure needs $O(n^2)$ time in the worst-case, as explained below.

The outermost `repeat` loop in that figure iterates at most n times (because each iteration adds at least one node to *included*, and the final contents of *included* is the e-hammock $\mathcal{H}$, which has n nodes). Let us consider the body of the `repeat` loop. Finding the most deeply nested, innermost block sequence `sequence` that contains the *included* nodes takes $O(n)$ time (essentially, it requires a walk up the AST from the *included* nodes until their lowest common ancestor is reached; we assume that the depth of the AST is bounded by a constant). The second `forall` loop in the `repeat` loop's body also takes $O(n)$ time. Therefore the total time requirement of this step is $O(n^2)$. (We assume that postdominators have been computed initially; that has time complexity $O(n^2)$.)

Step 2: Generating base constraints, as specified in Figure 5.5, takes $O(n^2V)$ time, as explained below.

It can be shown that whenever there is a path in the e-hammock $\mathcal{H}$ from a node m to a node n such that the *def* set of one of these two nodes has a non-empty intersection with the *def* or *use* set of the other node, there exists a (direct or extended) constraint $m \leq n$. Therefore, data-dependence constraints can be computed in time $O(n^2V)$, by doing a depth-first search starting from each node in the e-hammock (intersection of two *def/use* sets takes worst-case $O(V)$ time, using hash tables).

Generating control-dependence constraints takes $O(n^2)$ time, since each node has at most $O(n)$ control-dependence ancestors.

Antecedent constraints can be generated, again using depth-first search from each node, in $O(n^2)$ time.

We now shift our attention to the generation of extended constraints, the procedure for which is given in Figure 5.6. This procedure takes $O(n^3)$ time. The fundamental step

in this procedure, which is repeated until no new constraints are generated is: when a new constraint $m \leq n$ is generated, iterate through all existing constraints that involve m or n, and generate a set of new "$\leq$" constraints using those constraints and $m \leq n$. Each execution of this fundamental step takes $O(n)$ time, and it can be executed at most n^2 times (that is the total number of possible "$\leq$" constraints).

Step 3: The procedure for promoting nodes is given in Figure 5.9. This procedure takes $O(n^2)$ time. The fundamental step in this procedure, which is repeated until no more nodes are promoted, is: for each marked node m and for each node m that gets promoted, iterate through the constraints that involve m and see if the other nodes mentioned in those constraints need to be promoted, using the rules in Figure 5.9. This fundamental step takes $O(n)$ time, and it is repeated at most n times.

Step 4: The procedure for partitioning nodes into buckets is given in Figures 5.11 and 5.10. This step takes $O(n^2)$ time, as explained below.

Whenever a node p is added to a bucket, each other node m that in turn gets forced into some bucket via "$\leq$" constraints involving p and m can be found in constant time. This can be done, basically, by maintaining a graph whose nodes are the nodes being partitioned and whose edges are the "$\leq$" constraints, and by removing nodes from this graph as soon as they are assigned to any bucket. Once m is added to its bucket, $O(n)$ time is needed to add its control-dependence ancestors to the same bucket, and to add exiting jumps of which it is an antecedent to the *after* bucket (if necessary). On the other hand, when no forced node is available, selecting an unforced unassigned node takes constant time. Therefore this entire step takes $O(n^2)$ time.

Step 5: This step involves, for each of the three buckets B, making a copy of the original e-hammock $\mathcal{H}$ and removing from that copy nodes that are not in B. The nodes can be removed by repeating the following step as long as there remain non-B nodes in the copy: select a non-B node t that has an outgoing edge to a B-node (it can have at most one outgoing edge to a B-node, as observed in Section 5.5), and remove t by

redirecting all edges coming into it to its B-successor. This entire iterative step takes $O(n)$ time.

Step 6: This step basically involves visiting each node in each of the three buckets, and doing some constant-time processing if that node is an exiting jump. Therefore, this step takes $O(n)$ time.

In practice, when we applied a partial implementation of this algorithm to a dataset of 43 difficult clones in real programs (see Chapter 8), we found that the algorithm took 14 seconds or less on all but two of the largest clones in the dataset (it took about 5 minutes for each of those two largest clones).

Chapter 6

Clone-group extraction algorithm

This chapter describes the clone-group extraction algorithm. The input to the algorithm is a group of clones, and a mapping that specifies how the nodes in one clone match the nodes in the other clones (details about this mapping are given in Section 6.2). The output from the algorithm is a transformed program such that:

- each clone is contained in a hammock (which is suitable for extraction into a separate procedure), and
- matching statements are in the same order in all clones.

The algorithm can *fail* in certain situations; this means that the matching statements will not be the in same order in all clones (although each clone will definitely be contained in a hammock). We discuss this in detail in Section 6.3.

6.1 Algorithm overview

The first step in the clone-group extraction algorithm is to apply the individual-clone algorithm to each clone in the given group. That algorithm finds the e-hammock containing the clone, moves as many of the non-clone nodes in the e-hammock as possible out of the way, and converts exiting jumps into non-exiting jumps so that the clone is contained in a *marked* hammock that is suitable for extraction. From here on, whenever we say "clone", we actually mean the *marked* hammock that contains the clone and that was produced by the individual-clone algorithm.

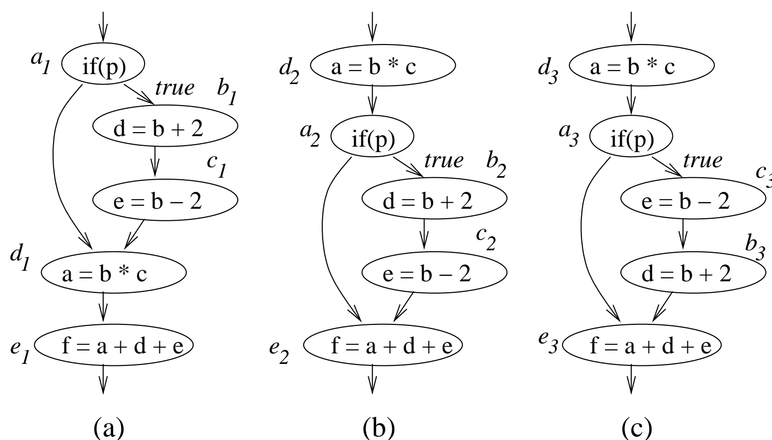


Figure 6.1 Example illustrating clone-group extraction

Recall that, as stated in Chapter 4, a *block* is a CFG subgraph that corresponds to a single (simple or compound) statement at the source-code level, whereas a *block sequence* corresponds to a sequence of statements. Recall also that every hammock is a block sequence. Each clone is a block sequence (because every hammock is a block sequence). This outermost-level block sequence of the clone is regarded, for the purposes of the clone-group algorithm, as a maximal block sequence. Clearly, this maximal block sequence can itself contain smaller blocks and maximal block sequences nested inside. The given clone group is said to be *in order* if corresponding maximal block sequences in the different clones, at all levels of nesting, are *in order* (have mapped blocks in the same order). If a maximal block sequence b in a clone and its corresponding maximal block sequences in other clones are *not* in-order, then it is not clear how extraction can be done while preserving semantics (because the single block sequence in the extracted procedure that represents b and its corresponding block sequences will have to be in one particular order). Therefore, our approach is to visit maximal block sequences in the clones, at all levels of nesting, and permute as many of them as needed to make all sets of corresponding block sequences (at all levels of nesting) be in-order.

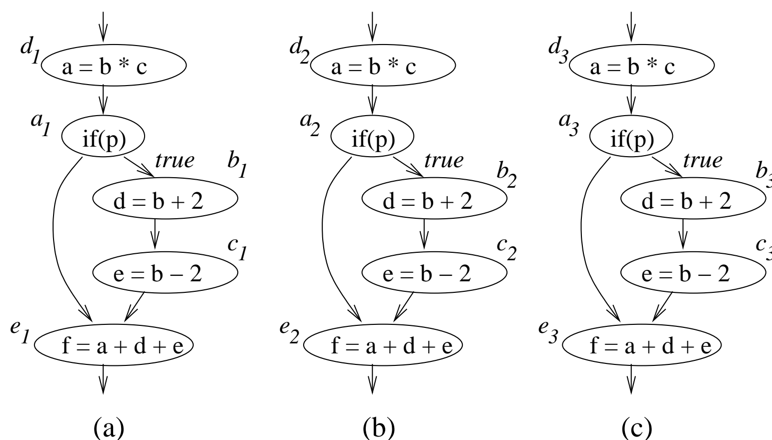


Figure 6.2 Output of clone-group extraction algorithm on clone group in Figure 6.1

6.1.1 Illustrative example

Figure 6.1 shows an (artificial) example group with three clones. The node-mapping is the obvious one: the a_is are mapped, the b_is are mapped, and so on. The clones are shown after the individual-clone algorithm has been applied to them. Each clone is, at the outermost level, a maximal block sequence that consists of an `if` block and two assignment blocks. Each `if` block in turn contains a nested maximal block sequence (its "then" part). The three outermost-level maximal block sequences correspond, as do the three maximal block sequences nested inside the three `if` blocks. Notice that neither of these two sets of corresponding maximal block sequences is in-order. Figure 6.2 contains the output of the algorithm for this example. Notice that the algorithm has permuted the outermost-level maximal block sequence in clone (a), as well as the inner maximal block sequence in clone (c). As a result, both sets of corresponding maximal block sequences are in-order, which means the group is in-order (and easy to extract).

Our approach for permuting a set of corresponding block sequences is defined later, in Figure 6.7 and in Section 6.3. However, it is notable that the approach uses control- and data-dependence-based sufficient conditions to conservatively estimate whether semantics-preserving permutations are possible; if the sufficient conditions allow, then it makes the set in-order, otherwise it fails (i.e., does no transformation).

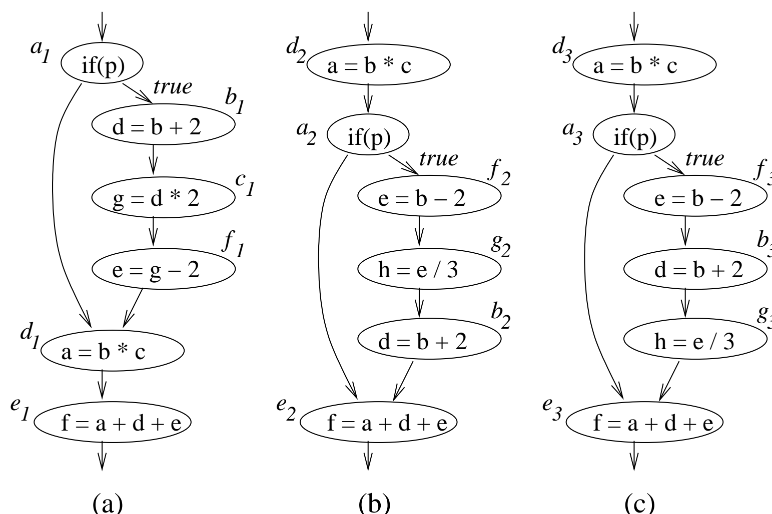


Figure 6.3 A clone-group with partial matches

6.1.2 Handling partial matches

Consider now a different example clone group, shown in Figure 6.3. As in the previous example, each node labeled x_i is mapped to nodes labeled x_j. Notice that the node c_1 in the first clone is unmapped (has no matching node in the other clones); notice also that g_2 and g_3, although mapped to each other, are not mapped to any node in the first clone. These partial matches do come up in practice, and the algorithm handles them. Partial matches introduce two complications, the first of which is that the notion of "corresponding" maximal block sequences becomes less obvious. In the example in Figure 6.3, it is intuitively clear that the "then" parts of the three `if` blocks correspond; this is because the three `if` blocks are mapped, which means they will be represented by a single `if` statement in the extracted procedure, and that `if` statement will have only one "then" part. However, not every node in each of these three maximal block sequences is mapped to some node in some other block sequence. In fact, even if the "then" part of some clone in this example had *no* mapped nodes, it would still correspond to the other "then" parts (because its `if` predicate is mapped to the other `if` predicates). We therefore define the correspondence between block sequences, as well as the mapping between blocks, recursively, as follows:

- The outermost-level maximal block sequences (i.e., the entire clones) correspond, by definition.
- Two blocks are mapped to each other if the two maximal block sequences of which they are constituents correspond, and the two blocks contain nodes that are mapped to each other.
- Two inner-level maximal block sequences correspond if both are C-nesting children of blocks that are mapped to each other, for some boolean value C.

In other words we start with the outermost-level maximal block sequences (which correspond by definition), and extend the correspondence to inner levels by determining which blocks are mapped to each other. The algorithm makes certain assumptions on the given node-mapping (specified in Section 6.2). Those assumptions have the following implications:

Uniqueness: A block in a clone is mapped to at most one block in any other clone. Similarly, a maximal block sequence is a clone corresponds to at most one maximal block sequence in any other clone.

Transitivity: The mapping between blocks is a transitive relationship, and so is the correspondence between maximal block sequences.

Kind-Preservation: Mapped blocks are of the same kind (e.g., `while` blocks are mapped to `while` blocks, and `if` blocks are mapped to `if` blocks).

Example: Consider the clones in Figure 6.3. Blocks b_1, b_2, and b_3 are mapped to each other; so are g_2 and g_3, and so are the three `if` blocks. There are two sets of corresponding maximal block sequences: the first set is the three outermost-level maximal block sequences (the three entire clones), while the second set is the "then" parts of the three `if` blocks. □

A second complication introduced by partial matches is that a set of corresponding maximal block sequences cannot simply be defined to be in-order iff mapped blocks are in the same order in *all* the block sequences in the set; this is because a constituent block of a block

sequence in the set can be mapped to blocks in some, but not all other block sequences in the set (e.g., blocks g_1 and g_2 in Figure 6.3), or can be altogether unmapped (e.g., block c_1 in the same figure). Our solution to this problem is based on partitioning the constituent blocks of the given set of maximal block sequences into equivalence classes; two blocks belong to the same equivalence class iff they are mapped to each other (recall that the blocks-mapping is one-to-one and transitive). The set of block sequences is defined to be in-order iff all of the block sequences in the set are consistent with some total order on the equivalence classes.

Our algorithm is outlined in Figure 6.4. The idea, basically, is to visit each set of corresponding maximal block sequences and check if it is already in-order; if it is not in-order, permute one or more block sequences in the set so that all the sequences become consistent with some total order on the equivalence classes. This permutation takes polynomial time, except in the situation (which is unusual in practice) where there are `gotos` from one block in a maximal block sequence to another block in the same sequence (Section 6.3.2 addresses this situation). As stated earlier, the algorithm bases its permutations on conditions that are sufficient to guarantee semantics preservation, and fails if there is no way to make the set in-order while respecting these conditions.

Example: Consider the three inner-level maximal block sequences in Figure 6.3 (i.e., the "then" parts of the `if` statements). There are four equivalence classes of blocks for this set of block sequences, $b = \{b_1, b_2, b_3\}$, $c = \{c_1\}$, $f = \{f_1, f_2, f_3\}$, and $g = \{g_2, g_3\}$. This set of maximal block sequences is not currently in-order: the block sequence in clone (a) is inconsistent with any total order in which f comes before b, and the corresponding block sequences in the other two clones are inconsistent with any total order in which b comes before f. Figure 6.5 shows the output of the algorithm for this example. Notice that the inner-level block sequences in clones (b) and (c) have been permuted, so that all three inner-level block sequences satisfy the total order b, c, f, g (due to data flows, this is the only total order that preserves semantics).

Given: A group of clones that satisfy the assumptions stated in Section 6.2.

Step 1: Apply the individual-clone algorithm individually to each clone in the given group of clones.

Step 2:

for all sets S of corresponding maximal block sequences in the clones (at all levels of nesting) **do**

  if S is not in-order **then**

    Use the procedure in Figure 6.7 to make S in-order. If that procedure fails, then fail.

  end if

end for

(At this point, the group of clones is in order or the algorithm has failed.)

Figure 6.4 Clone-group extraction algorithm.

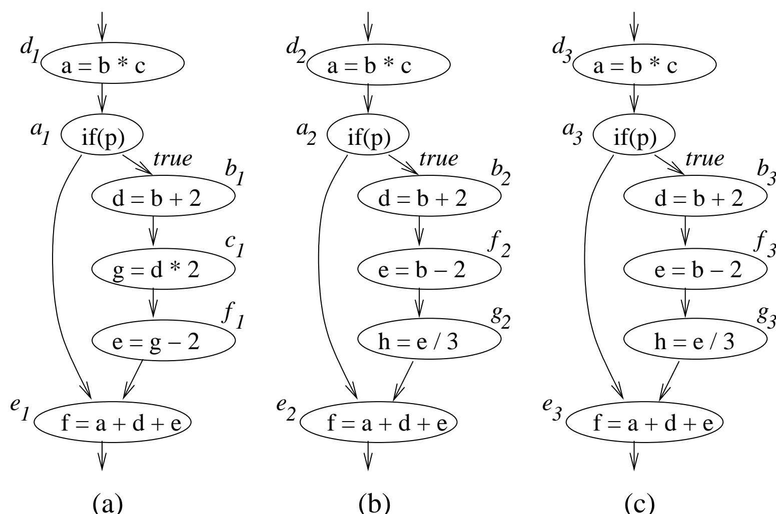


Figure 6.5 Output of algorithm on example in Figure 6.3

Notice also that the outermost-level block sequence in clone (a) has been permuted, so that the set of all three outermost-level block sequences satisfies the total order: d, `if`-block, e. □

Once the algorithm completes, and all sets of corresponding maximal block sequences are in-order (assuming the algorithm did not fail), it is easy to construct a single procedure that can replace all the clones. Each set of corresponding maximal block sequences S in the clones is represented by one maximal block sequence b in the extracted procedure. Each equivalence class of blocks of S is represented by a single constituent block of b; this block is guarded by a boolean flag parameter if its class does not contain a block from every block sequence in S. The order of blocks in b is the same as the total order with which the block sequences in S are consistent.

The rest of this chapter is organized as follows. Section 6.2 formally specifies the input to the algorithm. Section 6.3 presents the approach to making a set of corresponding maximal block sequences in-order. Finally, Section 6.4 discusses the complexity of the algorithm.

6.2 Input to the algorithm

In this section we formally specify the input to the algorithm, and the assumptions made by the algorithm regarding the input. The input is a group of clones (a mapping that defines how the nodes in one clone match the nodes in the other clones), and the CFGs of the procedures that contain the clones. Each individual clone is a set of nodes that is contained within a single procedure; however, different clones in the group can be in different procedures.

The algorithm handles a wide variety of clone groups with difficult characteristics:

- Individual clones can be non-contiguous, and can involve exiting jumps.
- Mapped nodes can be in different orders in the different clones.
- When the group consists of more than two clones, a node in one clone can be mapped to nodes in *some* but not *all* other clones (as illustrated in Figure 6.3). In fact, different clones in the group can consist of different numbers of nodes.

We call the tightest e-hammock that contains a clone and that has no backward exiting jumps (defined in Section 5.1) the "e-hammock of that clone". This e-hammock, like any e-hammock, is a block sequence. Because a clone can be non-contiguous, its e-hammock can contain nodes that are not part of the clone. The e-hammock can also contain exiting jumps. The given mapping between nodes in the clones is assumed to satisfy the following properties:

- The mapping is transitive; i.e., if m, n and t are nodes in three different clones, m is mapped to n and n is mapped to t, then m is mapped to t.
- A node in a clone is mapped to at most one node in each other clone (this allows for a node in a clone to be mapped to nodes in *some* but not all other clones).
- The clones are "non-overlapping"; i.e., the e-hammocks of the different clones are disjoint.

- The mapping preserves nesting relationships; i.e., if a node n is mapped to a node m, then one of the following must be true: neither node is a nesting child of a predicate that is inside the e-hammock of that node's clone, or both nodes are C-nesting children of predicates within their respective e-hammocks such that the two predicates are mapped, for some boolean value C (in other words, the two predicates belong to the clones, too).

- Mapped nodes are of the same "kind"; e.g., an assignment node is mapped only to other assignment nodes, a `while` predicate is mapped only to other `while` predicate nodes, and so on.

- Exiting `gotos` are mapped only to exiting `gotos`, and non-exiting `gotos` are mapped only to non-exiting `gotos`. Moreover, mapped non-exiting `gotos` have mapped targets. (For other kinds of jumps this property is implied by the assumption that the mapping preserves nesting relationships.)

Clone groups reported by our clone-detection tool often, but not always, satisfy the assumptions mentioned above. The examples in Figures 1.1 and 3.1 are ones that satisfy the above assumptions; the example in Figure 8.4 is one that does *not* satisfy the nesting-preservation requirement (the predicate "`if(filename != 0)`" in the first clone is inside the e-hammock of that clone, is a nesting parent of several cloned nodes, but is not mapped to any predicate in the second clone).

A clone group reported by the detection tool that does not satisfy these assumptions will need to be adjusted manually by the programmer so that it satisfies the assumptions before it is supplied to the extraction algorithm. (Section 3.4.4 discussed another reason why programmers might need to adjust reported clone groups, namely that they can be variants of ideal clone groups.)

6.3 Making a set of corresponding maximal block sequences in-order

Step 2 of the clone-group extraction algorithm (Figure 6.4) involves visiting each set of corresponding maximal block sequences that is not in-order, and making it in-order. The procedure to make a set of corresponding maximal block sequences in-order, which is the focus of this section, is given in Figure 6.7. The basic idea behind this procedure is to compute ordering constraints for each block sequence in the set based on control and data dependences (the procedures for computing the constraints are given in Figure 6.6), and then to permute one or more block sequences in the set while respecting the constraints so that all the block sequences in the set become consistent with some total order on the equivalence classes of blocks; respecting the constraints guarantees that the permutation is semantics preserving. The procedure in Figure 6.7 fails (without permuting any block sequence) if such a constraints-respecting permutation does not exist. The following two subsections describe, respectively, the two key steps in this procedure: generating the ordering constraints, and permuting the block sequences.

6.3.1 Constraints generation

The procedure in Figure 6.6(a), which is invoked from Step 1 in Figure 6.7, generates control-dependence-based constraints. Constraints are needed to preserve control dependences while permuting a block sequence if it has any of the following properties: there are jumps outside the sequence whose targets are inside, there are jumps inside the sequence whose targets are outside, or there are jumps from one constituent block of the sequence to another. If none of these conditions hold for a block sequence, then any permutation preserves all control dependences, and therefore no control-dependence-based constraints are needed.

Figure 6.8 contains an (artificial) illustrative example. Assume every node in the example (except the predicates and jumps) is an assignment. Nodes b_1, c_1, f_1, and e_1 are mapped to b_2, c_2, f_2, and e_2, respectively. Also, the two "if(p)" nodes are mapped to each other, as

Input: A set of corresponding maximal block sequences S. A constituent block of a block sequence in S is mapped to at most one constituent block of any other block sequence in S, and the mapping is transitive.
Output: Control-dependence- and data-dependence-based constraints.

(a) Procedure for generating control-dependence-based constraints:

1: **for all** block sequences b in S **do**
2: **for all** constituent blocks B_j of b **do**
3: **if** B_j contains a jump whose target is outside b or contains a node that is the target of a jump outside b **then**
4: **for all** other constituent blocks B_i of b **do**
5: **if** B_i precedes B_j in b **then** generate constraint $B_i < B_j$ **else** generate constraint $B_j < B_i$. ($B_i < B_j$ means that B_i must precede B_j after the permutation.)
6: **end for**
7: **end if**
8: **for all** constituent blocks B_m of b such that B_m follows B_j and there is a jump in either of these two blocks whose target is in the other block **do**
9: generate a constraint $B_j < B_m$.
10: **for all** constituent blocks B_l of b, $B_l \neq B_j$ and $B_l \neq B_m$ **do**
11: **if** B_l is between B_j and B_m **then**
12: generate constraints $B_j < B_l$ and $B_l < B_m$.
13: **else**
14: generate constraint $B_l \notin [B_j, B_m]$ (this means that B_l must not be in between B_j and B_m after the permutation).
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: **end for**

(b) Procedure for generating control-dependence-based constraints:

1: **for all** block sequences b in S **do**
2: **for all** pairs of constituent blocks B_i, B_j of b such that B_i precedes B_j **do**
3: **if** some node in B_j is data-dependent on some node in B_i then generate constraint $B_i < B_j$.
4: **end for**
5: **end for**

Figure 6.6 Procedures for generating control-dependence- and data-dependence-based constraints

Input: A set of corresponding maximal block sequences S. A constituent block of a block sequence in S is mapped to at most one constituent block of any other block sequence in S, and the mapping is transitive.

Output: Either one or more of the block sequences in S are permuted such that S becomes in-order, *or* the algorithm fails (no transformation is done).

Step 1. Generate control-dependence-based constraints as specified in Figure 6.6(a).

Step 2. Generate data-dependence constraints as specified in Figure 6.6(b).

Step 3. Create a *constraints graph* C. For each equivalence class M of constituent blocks of S create a vertex in C to represent M. (A constituent block of some block sequence in S that is mapped to no other block forms an equivalence class by itself.) For each constituent block B of S, let $v(B)$ denote the vertex in C that represents the class to which B belongs.

Step 4. For each constraint $B_i < B_j$, add an edge in C from $v(B_i)$ to $v(B_j)$.

Step 5. Find a topological ordering of the vertices in C that also respects each "$\notin$" constraint. A "$\notin$" constraint $B_n \notin [B_j, B_m]$ is respected iff $v(B_n)$ is *not* between $v(B_j)$ and $v(B_m)$ in the topological ordering.

Step 6.

1: **if** no such topological ordering exists **then**
2: fail (S cannot be made in-order while satisfying all constraints)
3: **else**
4: **for all** block sequences b in S **do**
5: permute b according to the topological ordering; i.e., a constituent block B_i of b precedes a constituent block B_j of b in the permutation iff $v(B_i)$ precedes $v(B_j)$ in the topological ordering.
6: **end for**
7: **end if**

Figure 6.7 Procedure for making a set of corresponding block sequences in-order

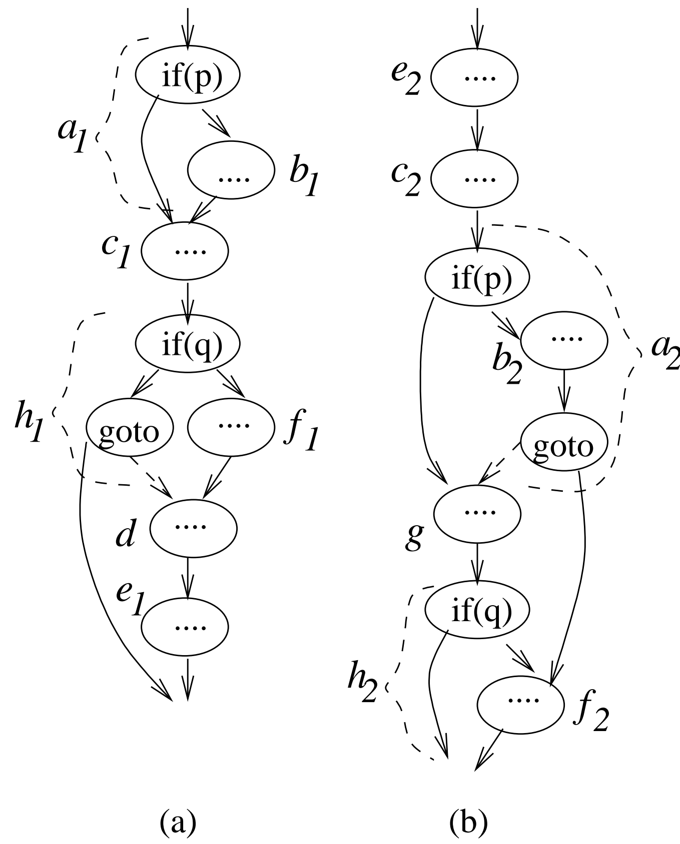


Figure 6.8 Example illustrating control dependence constraints

are the two "if(q)" nodes. The assignment nodes d and g are unmapped, as are the two `gotos`. Our algorithm allows such unmapped jumps, and these make extraction challenging (unmapped jumps are plausible in real clones; e.g., a computation could occur at two places, but one of them could have an intervening statement that checks some boundary condition and jumps out based on that).

In the procedure in Figure 6.6(a), control-dependence-based constraints are generated at two places, the first of which is the statement "`if` B_i precedes B_j .." (line 5). The block sequence (a_1, c_1, h_1, d, e_1) in Figure 6.8(a) illustrates the need for these constraints. Notice that h_1 contains a jump whose target is outside the sequence. Therefore, any permutation of this sequence needs to preserve the property that d and e_1 come after h_1 (otherwise d and e_1 would execute whether or not the `goto` in h_1 executes, which is incorrect). Similarly, a_1 and c_1 need to come before h_1 (else the `goto` could incorrectly bypass a_1 and c_1). The constraints generated in line 5 cover these situations; note that the meaning of the constraint $B_i < B_j$ is that B_i must precede B_j in the permutation.

The other place in Figure 6.6(a) where control-dependence-based constraints are generated is the statement "`if` B_l is between B_j and B_m .." (lines 11-15). The block sequence (e_2, c_2, a_2, g, h_2) in Figure 6.8(b) illustrates the need for these constraints. Notice that there is a jump in block a_2 whose target is in h_2. The constraint needed here is more complex: g must remain between blocks a_2 and h_2, whereas e_2 and c_2 can be anywhere *except* between these two blocks. Note that the meaning of the constraint $B_l \notin [B_j, B_m]$ is that B_l *cannot* be between B_j and B_m after the permutation.

Data-dependence-based constraints ensure preservation of data dependences. They are generated in a straightforward manner, using the procedure in Figure 6.6(b) (which is invoked from Step 2 of Figure 6.7).

6.3.2 Permuting the block sequences

After all the constraints are generated, each block sequence in the given set S is permuted in a constraints-respecting manner to make the set in-order. Our approach for this is to first

find a total order on the equivalence classes of constituent blocks of S that respects all constraints, and then, for each block sequence in S, determine its permutation by essentially projecting out of the total order the classes that include no block from this sequence. This ensures that each block sequence's permutation satisfies all constraints, and is at the same time consistent with the total order.

Recall that the constraints generated (in Figure 6.6) are in terms of individual constituent blocks of S, not equivalence classes. A total order on the equivalence classes satisfies a constraint c iff the total order satisfies the constraint that is obtained by replacing every block mentioned in c with the class that that block belongs to.

A total order of the equivalence classes that respects all constraints does not always exist. If in fact no such order exists, the algorithm fails for the given set S (i.e., does not permute the block sequences in S to make them in-order).

Notice that in the procedure in Figure 6.7, a *constraints graph* is used to find the total order on the equivalence classes. Each vertex in this graph represents one equivalence class, while the "$<$" constraints are encoded as its edges. Step 5 finds a topological ordering of this graph that also satisfies the "$\notin$" constraints (this topological ordering is the constraints-respecting total order on the equivalence classes). Step 5 is non-trivial. However a helpful observation is that any "$\notin$" constraint of the form $a \notin [b, c]$ is logically equivalent to the constraint $(a < b) \vee (c < a)$. Therefore a straightforward way to implement this step is: for each constraint $a \notin [b, c]$ add one of the two edges $a \rightarrow b$, $c \rightarrow a$ to the constraints graph, and see if a topological ordering is possible (i.e., see if the resulting graph is acyclic); use backtracking to systematically explore the choices until an ordering is found (and fail otherwise). This approach, although simple, has worst-case time complexity exponential in the number of "$\notin$" constraints. However, this is unlikely to be a problem in practice because "$\notin$" constraints arise only under the unusual situation where there is a `goto` in one constituent block of a block sequence whose target is in another constituent block of the same sequence. In our experimental studies (Section 8), there were no instances of a

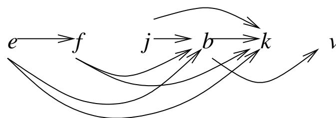


Figure 6.9 Constraints graph for the two outermost-level block sequences in the example in Figure 5.2

"$\not\in$" constraint. Still, an interesting open question is whether there is a polynomial-time algorithm for Step 5.

Example: Consider the example in Figure 5.2. Say the algorithm first visits the pair of outermost-level block sequences $(e_1, f_1, j_1, b_1, k_1, v_1)$ and $(j_2, e_2, f_2, b_2, k_2, v_2)$. The constraints graph for this pair is shown in Figure 6.9. Vertex e in this graph represents the equivalence class $\{e_1, e_2\}$, vertex f represents the equivalence class $\{f_1, f_2\}$, and so on.

Notice that b_1 and b_2 both contain jumps whose targets are outside the respective block sequences. Therefore the algorithm generates the following control-dependence-based constraints: $e < b, f < b, j < b, b < k$, and $b < v$ (these are the rewritten constraints). There are no "$\not\in$" constraints. The data-dependence constraints (again, after rewriting) are: $e < f$, $b < v$ (both due to output dependences), and $e < b$, $e < k$, $f < b$, $f < k$, $j < b$, $j < k$ (all due to flow dependence). Each of these constraints is an edge in the constraints graph (Figure 6.9).

A topological ordering of this graph is: e, f, j, b, k, v. The block sequences, when permuted in a manner consistent with this total ordering, become $(e_1, f_1, j_1, b_1, k_1, v_1)$ and $(e_2, f_2, j_2, b_2, k_2, v_2)$. Notice that this pair of block sequences is now in-order.

The other pair of corresponding maximal block sequences in this example that is not in-order is (g_1, h_1, i_1), (h_2, l, g_2, i_2) (l is unmapped). The vertices in the constraints graph for this pair are are $\{g, h, l, i\}$, and a total ordering of these vertices that respects all data-dependence-based constraints is h, l, g, i (there are no control-dependence constraints for this pair). These two block sequences, when permuted according to this ordering, become (h_1, g_1, i_1), (h_2, l, g_2, i_2). The third pair of corresponding maximal block sequences in this example (the "then" parts of the "`if(overPay > base)`" blocks) is already in-order. The

code in Figure 1.3 that is indicated with the "++"/"****" signs is the result of applying the above mentioned permutations to the clones in Figure 5.2.

A note about the `exitKind` assignments: recall that these assignments are introduced by the individual-clone algorithm, and are not part of the original clones. However, the "`exitKind = FALLTHRU`" assignments, of which there is at most one per clone (at its end), are always considered mapped. Other `exitKind` assignments are considered mapped if they correspond to exiting jumps that were originally mapped; e.g., the two assignments "`exitKind = BREAK`" in Figure 5.2 are considered mapped because the two original `break` statements to which they correspond are mapped. □

6.4 Complexity of the algorithm

Let G be the given group of clones. In this section we assume that the individual-clone algorithm has already been applied to each clone in G; therefore, that algorithm's time complexity is not included in the time complexity we present here. We use the following terminology throughout this section:

- n is the total number of nodes in all the clones in G.
- J is the total number of jump nodes in the procedures that contain the given clones.
- V is the total number of variables used/defined in G.

We assume, as we did with the clone-detection algorithm and the individual-clone algorithm, that the following structures have been built and are available to the algorithm: CFGs and ASTs of all procedures, and use/def sets of nodes. We also assume that hash table lookups take constant time, and that the depths of ASTs are bounded by a constant.

We first present the time complexity of the procedure in Figure 6.7. The input to this procedure is a set S of corresponding maximal block sequences; the procedure makes the set in-order, if possible, by permuting one or more block sequences, and fails otherwise. The worst-case time complexity of this procedure is $O(NJ + n_S V + N^2 V + 2^{MJ} M^2)$, where

- N is the total number of constituent blocks of the block sequences in S (counting outermost-level blocks only).
- M is the number of groups of *mapped* blocks. M can at most be equal to N (this worst case happens if no constituent block is mapped to any other constituent block).
- n_S is the total number of nodes in S ($n_S \leq n$).

The above result is explained below, after we present the worst-case time complexity of the entire clone-group algorithm.

The procedure in Figure 6.7 is invoked as many times as the number of sets of corresponding maximal block sequences in G (outermost-level and nested). This number is at most n. Also, N, n_S, and M can each be at most equal to n, for any set of corresponding block sequences S, and J can be at most equal to n. Therefore, the worst-case time complexity of the entire clone-group algorithm, after some simplification, is $O(n^3V + 2^{nJ}n^3)$. In the case where there are no "$\not\in$" constraints, which is what we expect for the vast majority of inputs, this time complexity reduces to $O(n^3V)$ (as explained below).

We now provide a (brief) derivation of the time-complexity result for the procedure in Figure 6.7:

- Control-dependence-based constraints can be generated in time $O(NJ)$, essentially by visiting each jump node once and generating the $O(N)$ constraints it is involved in. (The approach specified in Figure 6.6(a) for generating control-dependence-based constraints is a naive one compared to the approach just mentioned; that naive approach takes $O(N^2J)$ time.)
- The set of variables used/defined by nodes in the block, for all blocks, can be computed in time $O(n_SV)$.

  Then, data-dependence-based constraints can be generated (see Figure 6.6(b)) in time $O(N^2V)$.

- There are at most JM "$\not\in$" constraints. Recall that, as described in Section 6.3.2, each of these constraints is equivalent to the logical *or* of two "$<$" constraints, and that the approach to finding a topological order that satisfies all constraints is to systematically explore each of the two choices for each "$\not\in$" constraint. In other words the approach can try $O(2^{JM})$ choices in the worst case. For each choice, the approach checks for cycles in the constraints graph and does a topological sort of that graph if there are no cycles (Step 5 in Figure 6.7); this takes $O(M^2)$ time (there are at most M^2 "$<$" constraints). Note that if there are no "$\not\in$" constraints, then there is only choice to explore, and therefore the time complexity of finding the final topological order is simply $O(M^2)$.

Finally, Step 6 of the algorithm takes $O(N)$ time.

Chapter 7

Experimental results for the clone-detection algorithm

In this chapter, we present experimental results from the implementation of our clone-detection algorithm (Chapter 3). The tool finds clones in C programs. It uses CodeSurfer [Csu] to process the source code and build the PDGs. CodeSurfer also provides a GUI to display the clone groups identified by the tool using highlighting.

Recall that the clone-detection algorithm has three main steps: finding clone pairs, removing subsumed clone pairs, and combining the remaining clone pairs into groups. The first step, finding clone pairs, is implemented using Scheme, because CodeSurfer provides a Scheme API to the PDGs. The other two steps of the algorithm are implemented using C++.

For our first study our goals were to run the tool on some real programs to see what its time requirement was, and to see how many groups of clones it found and how large those clones were. We show in Figure 7.1:

- The sizes (in lines of source code and in number of PDG nodes) of three Unix utilities, *make*, *bison*, and *sort*, on which we ran the tool, and
- The running times for the three steps of the algorithm.

We used a 1 GHz Pentium III machine with 512 MB of memory for the experiments.

Figure 7.2 presents the results of running the tool on those three programs. For each of eight clone size ranges, three sets of numbers are reported: the number of clone groups identified that contain clones of that size, and the maximum and mean numbers of clones

Program	Program Size		Running Times (elapsed time)		
	# of lines of source	# of PDG nodes	find clone pairs (Scheme)	eliminate subsumed clone pairs (C++)	combine pairs into groups(C++)
make	30,499	288,572	400 min.	20 sec.	15 sec.
bison	11,540	31,197	18 min.	9 sec.	12 sec.
sort	1,961	6,712	4 min.	2 sec.	2 sec.

Figure 7.1 Example program sizes and running times

in those groups. Our experience indicates that clones with fewer than five PDG nodes are unlikely to be interesting to the programmer or good candidates for extraction; therefore, they are not reported by the tool.

When run on the Unix utilities, the tool found a number of interesting clones, many of which were non-contiguous and some of which involved out-of-order matches. (Some examples of the interesting clones identified by the tool were presented in Sections 3.4.1 and 3.4.2.) These results seem to validate both the hypothesis that programs often include a significant amount of "inexact" duplication, and the potential of our approach to find interesting clones.

To further evaluate the tool we performed two additional studies, described below. The goals of these studies were to understand:

- Whether the tool is likely to find (variants of) all the clone groups that a human would consider interesting;
- How many "uninteresting" clone groups the tool finds (i.e., groups that are not variants of any of the interesting groups), and how large those uninteresting clones are;
- To what extent the tool finds multiple variants of interesting clone groups rather just the "ideal" versions of those groups (the problem of variants was discussed in Section 3.4.4);

	Clone Size Ranges (# of PDG nodes)							
make	5–9	10–19	20–29	30–39	40–49	50–59	60–69	70–106
# clone groups	1417	391	64	40	39	5	3	40
max # clones in a group	43	29	6	7	2	5	7	2
mean # clones in a group	2.9	2.8	2.4	2.5	2	2.6	3.7	2
bison	5–9	10–19	20–29	30–39	40–49	50–59	60–69	70–227
# clone groups	469	149	36	11	7	10	4	27
max # clones in a group	61	26	7	3	2	2	2	2
mean # clones in a group	3.5	2.6	2.4	2.1	2	2	2	2
sort	5–9	10–19	20–29	30–39	40–48			
# clone groups	96	50	20	8	13			
max # clones in a group	9	3	2	2	2			
mean # clones in a group	2.5	2	2	2	2			

Figure 7.2 Results of running the clone-detection tool

- How often non-contiguous clones, out-of-order clone groups, and intertwined clones occur in practice;
- How many of the interesting clone groups are also good candidates for extraction into separate procedures.

For the first study, we examined one file (*lex.c*, 620 lines of code) from *bison* manually, and found four interesting clone groups. We then ran the tool on *lex.c*; it identified 41 clone groups. Nineteen of those groups were variants of the interesting manually-identified clone groups, including several variants for each of the four interesting groups. The remaining 22 clone groups identified were uninteresting in our judgment (an example of an uninteresting clone group identified by the tool, that was actually not encountered in this study, was shown in Figure 3.20). More than half of the uninteresting clone groups (12 out of 22) had clones with fewer than 7 nodes (which was the size of the smallest interesting clone); the largest uninteresting clone identified had 10 nodes.

For the second study, we examined all 95 clone groups identified by the tool for *bison* whose clones had 20 or more nodes (the largest identified clone had 227 nodes). We chose 20 as the starting size to test the hypothesis that the uninteresting clones identified by the tool tend to be quite small. Ten of the 95 clone groups identified were uninteresting; the largest of these uninteresting clones contained 28 nodes. The remaining 85 groups identified were variants of 17 interesting clone groups (in both of the studies some of the groups identified by the tool were actually variants of the union of several neighboring interesting clone groups).

In the two studies, we encountered a total of 18 interesting clone groups (three groups showed up in both studies) containing a total of 64 individual clones. Of those 64, 22 were non-contiguous. Three of the 18 clone groups involved out-of-order matches, and none involved interleaved clones. (In Chapter 8, we present similar statistics about a larger dataset of interesting clone groups that we used to evaluate the extraction algorithms.)

Fifteen of the eighteen interesting clone groups were also good candidates for extraction into separate procedures. The remaining three groups, although interesting, were not large

enough for the benefits of extraction to outweigh the costs; one of them would have required too many parameters as a separate procedure, the second one would have required several mismatching expressions whose total size was large relative to the size of the clone to be parameterized away, and the third one would have required too many predicates to be duplicated relative to the size of the clone.

7.1 Discussion

The results of our experiments indicate that our approach is capable of finding interesting clones. Many of these clones are non-contiguous, and some of the groups involve out-of-order matches. The tool is not likely to miss any clones that a human would consider interesting, and additionally is not likely to produce too many clones that a human would consider uninteresting (except small ones).

The two studies also reveal that the tool often finds multiple variants of ideal clones rather than just the ideal ones. To some extent this problem is intrinsic to the slicing-based approach (as discussed in Section 3.4.4). However, artifacts of CodeSurfer magnify the number of variants generated. For example, conservative pointer analysis sometimes causes spurious flow-dependence edges in the PDGs, which in turn cause the slicing to proceed non-ideally (CodeSurfer implements Andersen's flow-insensitive context-insensitive pointer-analysis algorithm [And94]). Another such factor is CodeSurfer's treatment of complex predicates (predicates that involve short-circuiting operators such as "`&&`", or that involve procedure calls); we discuss this factor below in detail.

Consider the example clone pair in Figure 3.11. Two variants of the ideal clone pair in that figure that were produced by the tool were shown in Section 3.4. One of them, shown in Figure 3.11, was produced by starting the slicing from the two matching statements labeled `(8)`. The other one, shown in Figure 3.19, was produced by starting the slicing from the matching predicates `(4)` and `(5)` in the first and second fragments, respectively. The production of these two variants is not unexpected, given the design of our approach. However, additional variants of the same ideal clone pair are produced because

```
    t = !type_name;
    if(!t)
      goto L;
    t = typed;
L:  if(t)
```

Figure 7.3 CodeSurfer's decomposition of the predicate "`if(!type_name && typed)`"

CodeSurfer *decomposes* complex predicates. We illustrate the problem using the predicate "`if(!type_name && typed)`", which occurs twice in each clone in that example (with labels `(4)` and `(5)`). CodeSurfer turns each of those predicates into a sequence of nodes, as shown in Figure 7.3 (`t` in that figure is a new temporary variable). As a result of this decomposition the approach finds other variant clone pairs, in addition to the two mentioned above; e.g., it finds a variant by starting the slicing from the node "`t = !type_name`" that was obtained from the predicate labeled `(5)` in the first fragment and the node "`t = !type_name`" that was obtained from the predicate labeled `(13)` in the second fragment. This variant would not have been produced if CodeSurfer did not decompose complex predicates, because the nodes `(5)` and `(13)`, when not decomposed, are syntactically non-matching.

All told, however, the tool's usefulness outweighs the problem caused by variants. As mentioned in Section 3.4.4, we were able to examine all 127 clone groups reported by the tool for *make* whose clones had 30 or more nodes, in less than four hours time. Manual clone detection is likely to take far more time, and moreover, would involve the risk of missing clones. Furthermore, it might be possible to devise simple, automatic heuristics that group together related variant clone groups produced by the tool, and pick and show one or a few promising candidates from each group to the user.

As for the running time, the tool is currently quite slow. The generation of variants is partly a cause for this. Another major cause for this is that the key step in the algorithm, finding clone pairs, is implemented in an interpreted language (Scheme). Yet another factor is that our primary concern has been improving the quality of clones found, as opposed to improving efficiency. Therefore, improvements in the tool's engineering have the potential to

speed it up significantly. Additional heuristics to reduce the production of variants can help, too. Finally, it may be possible (and profitable) to generate clone groups directly, rather than generating clone pairs and then combining them into groups (because for each clone group that contains k clones, we currently generate $(k^2 - k)/2$ clone pairs first).

Chapter 8

Experimental results for extraction algorithms

This chapter presents the results of some studies we did to evaluate the performance of both of our extraction algorithms, in comparison to an “ideal” extraction (performed by us, using our best judgment). We began the studies by identifying 50 groups of clones involving 173 individual clones in three real programs, using the clone-detection tool. Our first goal for the studies was to evaluate our individual-clone extraction algorithm. For this, we first extracted each individual clone in the dataset using our best judgment (we call this the “ideal extraction”); some techniques used during this process (reordering statements, handling exiting jumps, duplicating predicates, promotion) are also used by the algorithm, but other techniques not incorporated in the algorithm were also used as necessary (this is described later). After performing the ideal extraction, we applied the individual-clone algorithm to the clones in the dataset, and compared its results to those of the ideal extraction.

Our second goal for the studies was to evaluate the clone-group extraction algorithm. Our methodology was similar: do an “ideal extraction” on each clone group, then apply the clone-group algorithm to the groups, and then compare the two outcomes.

Chapter 9 presents the results of other studies we did, to compare our algorithm to two previously reported automatic approaches to procedure extraction, [LD98] and [DEMD00].

8.1 Dataset selection

Our studies were performed using a dataset of 50 clone groups, involving 173 individual clones, from three programs: the Unix utilities *make* and *bison* (the sizes of which were given

# clone groups:	50
# individual clones:	173
Max. clones per group:	14
Median clones per group:	2
Max. clone size (# statements and predicates):	53
Median clone size:	7
# non-contiguous clones:	30
# clones with exiting jumps:	25
# difficult clones:	43
# out-of-order groups:	10
# groups where mapping violates nesting relationships:	3
# difficult groups:	27

Figure 8.1 Dataset statistics

in Figure 7.1), and NARC[1] [WM99], a graph-drawing engine developed by IBM. We used 4 of the 70+ files in NARC for the studies; the combined sizes of these four files was 11,060 lines of code. Figure 8.1 presents some statistics about the clone groups in the dataset. Forty-three of the 173 individual clones (25%) were *individually difficult*, i.e., were non-contiguous and/or involved exiting jumps; 12 individual clones had *both* these difficult characteristics. Ten clone groups exhibited out-of-order matching. We call a clone group difficult if any of its individual clones are difficult, or if the group exhibits out-of-order matching, or if the mapping between nodes in the clones in the group does not preserve nesting relationships (see Section 6.2). Twenty-seven clone groups out of the 50 in the dataset (54%) were difficult.

The clone groups in the dataset were identified using the clone-detection tool (see Chapters 3 and 7). Because the tool often identifies multiple variants of a single ideal clone group instead of just the ideal group, we examined the output of the tool, determined what the ideal clone groups were, and chose for the dataset those ideal groups that were good candidates for extraction into separate procedures (using our best judgment). The program *make* contributed 11 clone groups to the dataset; we obtained these by examining all clone groups reported by the tool for *make* whose clones had 30 or more nodes. The program *bison* contributed 16 clone groups to the dataset (all but one of these were found in the study described in Chapter 7 where we examined all clone groups reported for *bison* whose clones had 20 or more nodes). For NARC we examined about 250 reported clone groups containing the largest clones, and from those identified 23 ideal clone groups for the dataset.

8.2 Implementation

We have partially implemented the individual-clone extraction algorithm, for C programs. We used CodeSurfer for this implementation, too; in particular the implementation uses CFGs, node *def/use* sets, and control-dependence edges computed by CodeSurfer. Because CodeSurfer supports program analysis only, and not transformations, we implemented

[1]NARC is a registered trademark of IBM.

Steps 1 through 4 of the individual-clone algorithm (see Chapter 5); i.e., the implementation finds the tightest e-hammock of a clone, computes all constraints, promotes unmovable intervening non-clone nodes, and partitions the remaining unmarked nodes into the *before* and *after* buckets. The implementation uses highlighting in conjunction with the CodeSurfer GUI both to accept the nodes in a clone as input, and to display the final contents of the three buckets *before*, *marked*, and *after*.

8.3 Performance of the individual-clone extraction algorithm

In this section, we examine the performance of the individual-clone algorithm on the clones in the dataset, in comparison to the ideal extraction. We used the partial implementation of the individual-clone algorithm for this study; its running time was less than 14 seconds (elapsed time) on all clones in the dataset except the two largest ones, on each of which it took about 5 minutes. Figure 8.2(a) summarizes the comparison of this algorithm with the ideal extraction. The 173 clones in the dataset are partitioned into four disjoint groups, one per row in Figure 8.2(a). The first row shows that 130 of the clones were *not* individually difficult (i.e., were contiguous, and did not involve exiting jumps); therefore, the individual-clone algorithm has nothing to do on these clones. The second row shows that the algorithm performed ideally on 29 out of 43 difficult clones; of these 29 clones, 16 were non-contiguous and 19 involved exiting jumps (i.e., 6 clones had both difficult characteristics). The third row pertains to 8 clones on which the the approach (as described in Chapter 5) would by itself *not* produce the ideal output, but on which the implementation, due to CodeSurfer's decomposition of complex predicates (see the example in Figure 7.3), managed to produce the ideal output. The fourth row shows that the algorithm produced non-ideal output on 6 difficult clones (more on this row and the previous row later in this section).

Category	# total	# non-contig.	# exiting jumps
Not difficult	130		
Difficult, ideal output	29	16	19
Difficult, ideal output due to CodeSurfer's representation	8	8	3
Difficult, non-ideal output	6	6	3
	173	30	25

(a) Characterization of algorithm output

Technique	Ideal output	Ideal output, due to CodeSurfer's representation	Non-ideal output	
			human	algo.
Moving without duplication	4		2	
Moving with duplication	3			5
Moving with predicate-value reuse		8	1	
Exiting jumps	19	3	3	3
Promotion	9	2	4	4

(b) Techniques used on difficult clones, with number of clones

Figure 8.2 Comparison of the individual-clone algorithm and ideal extraction

8.3.1 Techniques used to make clones extractable

Figure 8.2(b) enumerates, for each transformation technique incorporated in the algorithm, the number of difficult clones on which the technique was used in both the ideal extraction and the extraction performed by the algorithm. Each technique appears in its own row. The second column (labeled "Ideal output") pertains only to the difficult clones on which the algorithm performed ideally, and would have performed ideally even without CodeSurfer's decomposition of predicates. The third column (labeled "Ideal output, due to CodeSurfer's representation") pertains to the difficult clones on which the algorithm performed ideally, but as a result of CodeSurfer's decomposition of predicates. The last two columns pertain to the clones on which the algorithm performed non-ideally; two separate sets of numbers are required for these clones because the algorithm and the ideal extraction do not involve the exact same techniques on these clones. Note that more than one technique was applied on certain clones to make them extractable; such clones are counted separately under each technique that was applied.

Regarding the techniques: "moving without duplication" means moving an intervening unmarked node to *before* or *after* without any duplication of predicates (i.e., the unmarked node's control-dependence ancestors are not placed in any other bucket except the one that contains that node); "moving with duplication" on the other hand, occurs when an intervening unmarked node is moved out with duplication of predicates. The technique of handling exiting jumps is applicable on each clone that involves exiting jumps, and therefore the numbers in that row are the numbers in the last column of Figure 8.2(a).

The technique "moving with predicate-value reuse" is one that is *not* incorporated in the algorithm, but that is used (on 9 different clones) during the ideal extraction. This technique is essentially an extension of "moving with duplication", used in situations where moving with duplication is not possible. It is illustrated by the example clone pair shown in Figure 8.3, which is from the program *bison* and which belongs to the dataset. Each clone in this example first checks if the current character in the input stream `finput` is a digit or minus sign; if yes, it reads an integer from the stream, else it prints a warning. The warning

`++ c = getc(finput);`
`++ if (isdigit(c) || c == '-') {`
`++   ungetc (c, finput);`
`++   n = read_signed_integer(finput);`
`++   c = getc(finput);`
`++ }`
`++ else {`
`     warni("@%s is invalid",`
`            printable_version(c));`
`++   n = 1;`
`++ }`
`++ fprintf(fguard, "yylsp[%d]",`
`++          n-stack_offset);`
`++ yylsp_needed = 1;`
	`++ c = getc(finput);`
	`++ if (isdigit(c) || c == '-') {`
	`++   ungetc (c, finput);`
	`++   n = read_signed_integer(finput);`
	`++   c = getc(finput);`
	`++ }`
	`++ else {`
	`     warn("invalid`
	`          @-construct");`
	`++   n = 1;`
	`++ }`
	`++ fprintf(fguard, "yylsp[%d]",`
	`++          n-stack_offset);`
	`++ yylsp_needed = 1;`

Figure 8.3 A clone pair in *bison* extracted using predicate-value reuse

statements do not match (they invoke different functions), and are therefore not part of the clone pair. The ideal extraction, in our judgment, involves having the extracted procedure return a boolean value (the value of the predicate "`if (isdigit(c) || c == '-')`"); the two warning statements are placed after the two call sites, respectively, conditional on the value returned by the procedure. This option is preferable to promoting both the warning statements, because it avoids placing guarded code in the extracted procedure, and also because the future reusability of the extracted procedure is improved (each future call to this procedure can print its own warning message, as appropriate). Note that regular duplication, i.e., letting the warning statements be in the *after* code, conditional on a copy of the predicate "`if (isdigit(c) || c == '-')`", would not work in this example, for such a copy, if present, might use the wrong of value of `c` (because of the statement "`c = getc(finput)`" in the *marked* bucket). The solution to this problem that was adopted in the ideal extraction was to return that predicate's value, which, in essence, is the same as saving the predicate's value into a temporary variable (in the *marked* bucket), and using that variable in place of a copy of the original predicate in the *after* code. We call this technique *predicate-value reuse*.

Our individual-clone algorithm does not incorporate the predicate-value reuse technique. However, recall that CodeSurfer, as part of its decomposition of complex predicates, introduces a temporary variable `t` to hold the value of the entire predicate (see the example in Figure 7.3). Therefore, on the example in Figure 8.3, the implementation of the individual-clone algorithm *was* able to move out the warning statements to the *after* code, placing in that bucket a copy of just the final (CodeSurfer-generated) predicate "`if(t)`" (see Figure 7.3). In other words, due to this artifact of CodeSurfer, the implementation was able to produce the same output on the example in Figure 8.3 as would have been produced if predicate-value reuse was in fact incorporated in the algorithm. (The temporary variable `t`, the assignment to which stays in the *marked* bucket, can be converted into a return value of the procedure at the time of actual extraction.) The third row in Figure 8.2(a) and third column in Part (b) of that figure (both titled "Ideal output, due to CodeSurfer's representation") pertain to the 8 clones in the dataset on which the ideal extraction involved

```
++ hash = 0;
   if (filename != 0) {
     if (*filename == '\0') {
       return 1;
     }
     p = filename;
++   for (;*p != '\0'; ++p)
++     HASH (hash, *p);
++   hash %= DIRFILE_BUCKETS;
     ...
   }
```
(a)

```
++  hash = 0;
++  for (;*p != '\0'; ++p)
++    HASH (hash, *p);
++  hash %= DIRFILE_BUCKETS;
```
(b)

Figure 8.4 A difficult clone pair in the dataset from *make*

predicate-value reuse and on which the implementation produced the ideal output due to CodeSurfer's decomposition of predicates. From here on, we regard these 8 clones as clones on which the algorithm performed ideally.

8.3.2 Non-ideal behavior of individual-clone algorithm on some clones

Although the individual-clone algorithm produced the ideal output on a vast majority of the difficult clones (86%), it did produce non-ideal output on 6 clones (see Figure 8.2(a)).

On 3 of these 6 clones the algorithm performed non-ideally by over-aggressively moving intervening non-clone nodes with duplication of several predicates; these non-clone nodes were promoted in the ideal extraction for reasons of readability, although semantics-preservation did not strictly require that they be promoted.

For 1 of the 6 clones on which the algorithm performed non-ideally, the ideal extraction involved moving out an intervening non-clone node using the predicate-value reuse technique. The predicate in this clone was a simple predicate (one involving no short-circuiting operators) whose value CodeSurfer did not store in a temporary variable. Therefore, the algorithm ended up promoting the intervening non-clone node.

There are 2 remaining clones on which the individual-clone algorithm performed non-ideally, both for similar reasons. One of these 2 clones is shown in Figure 8.4(a) (Part (b) of that figure shows another clone, which happens to be an easy clone, that is in the same clone group as the clone in Part (a)). In the fragment in Part (a) the variable `hash` is not live after the `for` loop; thus, the assignment "`hash = 0`" can be moved inside the "`if(filename != 0)`" statement, to the point just before the `for` loop, so that the clone becomes contiguous (and this is what is done by the ideal extraction). However, the algorithm, to guarantee that the program's semantics is preserved, never moves nodes across the boundaries of conditionals. Therefore, on this example, it promoted all the intervening non-clone nodes (except for "`p = filename`", which was moved to the *before* bucket with duplication of predicates).

8.4 Performance of the clone-group extraction algorithm

In this section, we examine the performance of the clone-group extraction algorithm on the 50 clone groups in the dataset. As with the evaluation of the individual-clone algorithm, we first performed an "ideal extraction" on each clone group in the dataset, and then applied the clone-group extraction algorithm to those clone groups. For Step 1 of the clone-group algorithm, which is an application of the individual-clone algorithm to each clone in the group (see Figure 6.4), we used the partial implementation of the individual-clone algorithm. We did not implement Step 2 of the clone-group algorithm; we performed that step manually for this study, using information provided by CodeSurfer about def/use sets of nodes to determine the safety of a permutation, whenever that was not obvious.

Figure 8.5 presents the results of the comparison of the algorithm's output to the ideal output. The set of all clone groups in the dataset is divided into four disjoint categories, one per row. The first row shows that 23 of the 50 clone groups were "not difficult"; these groups were extractable to begin with, so there was nothing for the algorithm to do. The second row shows that the algorithm produced exactly the ideal output on 19 of the 27 difficult groups (70%). These 19 groups contain 62 individual clones, 29 of which were individually difficult.

Category	Groups: # total	Groups: # out-of order	Individual clones: # total	Individual clones: # difficult	Individual clones: # non-contig.	Individual clones: # exiting jumps
Not difficult	23		88			
Difficult, ideal	19	6	62	29	16	16
Difficult, non-ideal	4	1	8	7	7	4
Difficult, algorithm fails	4	3	15	7	7	5
	50	10	173	43	30	25

Figure 8.5 Performance of clone-group algorithm in comparison to ideal extraction

The third row indicates that the algorithm produced non-ideal results on 4 clone groups, and the fourth row indicates that it failed on 4 other clone groups (we address these issues in the following sections). Ten of the 27 difficult groups involved out-of-order matches; the algorithm was able to reorder clones in 7 out of these 10 groups (although it still produced non-ideal output on one of these seven groups, as discussed later).

The three clone groups shown in Figures 3.1, 3.5, and 3.11 are examples of clone groups that were in the dataset and on which the algorithm produced ideal results. The algorithm moved out the intervening non-clone statements in the example in Figure 3.1, and re-ordered the out-of-order group shown in Figure 3.5. In the case of the example in Figure 3.11, the input given to the algorithm (i.e., the ideal clones) consisted of the two "`case '$'`" blocks; the two `case` labels themselves were left out of the clones, and so were the two statements labeled `(3)`, and the statement labeled `(2)` (in the second clone); the two statements labeled `(1)` were mapped to each other. The algorithm promoted the statement labeled `(2)` and the two statements labeled `(3)` (they cannot be moved out without affecting data dependences, and in our judgment it is appropriate for them to be in the extracted procedure in guarded form).

We now discuss the groups on which the algorithm failed, either because of out-of-order issues or because the node-mapping did not preserve nesting relationships, and the groups on which it performed non-ideally.

8.4.1 Failure due to out-of-order issues

The algorithm failed on one out-of-order group that it could not make in-order. The two clones in this group have the following form:

Clone 1	Clone 2
`if` (p) `v |= K;`	`if` (q) `v |= L;`
`if` (q) `v |= L;`	`if` (p) `v |= K;`

The first `if` statement in the first clone is mapped to the second `if` statement in the second clone, and the second `if` statement in the first clone is mapped to the first `if` statement in the second clone. Therefore, the two `if` statements have to be permuted in one of the clones for the group to become in-order. However, the clone-group algorithm performs neither permutation, because of the output dependence (on the variable `v`) from the first `if` statement to the second `if` statement (in both clones). In other words, it fails. However, one of the two clones *was* permuted in the ideal extraction, by recognizing that two bitwise-or assignments done to the same variable (`v`) can be permuted without any change in semantics.

8.4.2 Failure due to mappings not preserving nesting structure

The clone-group algorithm failed on 3 clone groups whose node mappings did not preserve nesting relationships. One of these 3 groups is the group shown in Figure 8.4. Note that the algorithm would have actually succeeded on this example had the individual-clone algorithm (which is applied in Step 1 of the clone-group algorithm – see Figure 6.4) done the ideal transformation: move "`hash = 0`" to the point just before the `for` loop, rather than promoting the intervening non-clone nodes.

The two other clone groups on which the algorithm failed are groups that inherently violate the nesting-preservation requirement (i.e., Step 1 of the algorithm had no role in the failure). One of these clone groups is the group shown in Figure 3.13.

8.4.3 Non-ideal performance of the algorithm

The clone-group algorithm performed *non-ideally* on 4 clone groups. We consider the output of the clone-group algorithm to be non-ideal if it does not exactly match the ideal output. This can occur either because Step 1 of the algorithm (the individual-clone algorithm) produces non-ideal output, or it can occur because of a shortcoming in Step 2. As discussed in Section 8.3.2, the individual-clone algorithm produced non-ideal output on 6 clones. One of those clones is the clone in the left column of Figure 8.4; recall that the clone-group extraction algorithm failed on the clone group shown in that figure as a result. The other five clones on which the individual-clone algorithm performed non-ideally were members of 3 groups, and thus Step 1 accounted for 3 of the 4 clone groups for which the clone-group algorithm produced non-ideal output. There was one additional group on which the clone-group algorithm performed non-ideally, even though Step 1 produced the correct output and even though the group was not out-of-order. Both clones in this group contained promoted fragments (after the individual-clone algorithm was applied to the clones). The ideal extraction involved recognizing that the promoted fragments in the two clones, although not matching each other syntactically, had the same semantics; therefore the promoted fragment in one of the clones was rewritten to make it identical to the other promoted fragment (so that both fragments could be extracted without the use of any guarding in the extracted procedure). The clone-group algorithm does not incorporate any transformations besides movement of non-clone nodes (in Step 1), and reordering of cloned nodes (in Step 2). Therefore, on this example, the final output contained both the mismatching fragments, as promoted code (i.e., both fragments would have to be placed in the extracted procedure in guarded form).

8.5 Program size reduction achieved via procedure extraction

We performed another study whose goal was to determine the space-savings that can be achieved via extraction of clone groups. We performed this study on the 4 files in the

	# of lines of source	# of PDG nodes	# of clone groups extracted	total # of clones extracted	file size reduction	avg. func. size reduction
file 1	1677	2235	3	6	1.9%	5.0%
file 2	2621	4006	12	24	4.7%	12.4%
file 3	3343	6761	3	7	2.1%	4.4%
file 4	3419	4845	12	40	4.9%	10.3%

Figure 8.6 Space savings achieved in NARC via procedure extraction

NARC program, by manually extracting 30 groups of clones (77 individual clones) in the best manner according to our judgment (i.e, we performed "ideal" extractions). We extracted the clone groups into macro definitions, rather than procedures, to avoid changing the running time of the program. Extracting clones groups into macros also allowed us to extract some groups that were not used in the dataset in the previous study because the differences among the clones in those groups made procedure extraction infeasible or undesirable (in that study we only considered clone groups that were good candidates for extraction into procedures).

The results of this study are summarized in Figure 8.6, which gives, for each of the four files:

- the file's size (in lines of source code and in number of PDG nodes);
- the number of clone groups that were extracted;
- the total number of extracted clones;
- the reduction in size of the file (in terms of lines of code);
- the average reduction in size for functions that included at least one extracted clone (in terms of lines of code).

8.6 Summary

Difficult-to-extract clone groups occur frequently in practice. In the dataset, which contains all large clones that were good candidates for extraction that were found in three real programs by the clone-detection tool, 25% of the clones were *individually difficult* – were non-contiguous and/or involved exiting jumps. In particular, 17% of the individual clones were non-contiguous, and 14% of them involved exiting jumps (6% of the clones involved *both* these characteristics). Twenty percent of the clone groups in the dataset involved out-of-order matches; 54% of the clone groups were difficult – were out-of-order, or had one or more clones that were individually difficult, or had matchings that did not preserve nesting structure. The clone-group extraction algorithm produced the same output as an "ideal extraction" (done by us, using our best judgment) on 70% of the difficult groups (groups that are not difficult are extractable to begin with, so there is nothing for the algorithm to do). Considering that no automatic algorithm is likely to be able to employ the full range of transformation techniques used by a human, (and thus 100% ideal performance by an automatic algorithm is probably not feasible) we regard the results of the studies as very encouraging, indicating that the algorithm is likely to be useful in practice. In Chapter 9 we present the results of experiments in which we applied two previously reported approaches to procedure extraction to the clone groups in the dataset; our findings in those experiments was that our approach outperformed theirs on a vast majority of the clone groups in the dataset.

Chapter 9

Related Work

In the first part of this chapter, we discuss previously reported work that is related to our work on clone detection. The second part of the chapter focuses on previous work that is related to our work on procedure extraction.

9.1 Related work on clone detection

The key advance of our work over previous work in clone detection is the use of a slicing-based approach that is able to identify interesting clones even in the presence of difficult characteristics such as non-contiguity and out-of-order matching. Previous approaches to clone detection [Mar80, FMW84, LDK95, Vah95, DBF+95, Zas95, MLM96, KDM+96, Bak97, BYM+98, ACCP98, BG98, CM99, KL99, Run00, CSCM00, DEMD00], with a single exception [BG98], either work on the source text of the program, or on the abstract syntax tree (AST) representation, or on the control-flow graph (CFG) representation. The problem with these representations, which are closely tied to the source code, is that they encode the often arbitrary choice of statement-ordering that the programmer made while writing the program. Therefore these approaches, unlike ours, are in general unable to find clone groups with difficult characteristics that are nevertheless good candidates for extraction into separate procedures.

We discuss previously reported approaches to clone detection in Sections 9.1.1 through 9.1.5. Then, in Sections 9.1.6 and 9.1.7, we discuss previous work in areas that are closely related to clone detection.

9.1.1 A text-based approach

Baker [Bak95, Bak97] describes a text-based approach that finds all pairs of matching "parameterized" code fragments in source code. A code fragment matches another (with parameterization) if both fragments are contiguous sequences of source lines, and some global substitution of variable names and literal values applied to one of the fragments makes the two fragments identical line by line. Comments are ignored, as is whitespace within lines. Because this approach is text-based and line-based, it is sensitive to lexical aspects like the presence or absence of new lines, and the ordering of matching lines in a clone pair. Our approach does not have these shortcomings. Baker's approach does not find intertwined clones. It also does not (directly) find non-contiguous clones. A postpass can be used to group sets of matching fragments that occur close to each other in the source, but there is no guarantee that such sets belong together logically.

Other text-based approaches to clone detection are [Mar80, FMW84, LDK95, Vah95, Zas95, CM99, CSCM00]. Of these approaches, the approach of [Vah95] works on source code, and uses the Agrep [WM92] tool to find inexact matches. The other approaches work on intermediate-level code, assembly code or Java bytecode, and find exact matches only (modulo renamed variables, for some of them).

9.1.2 AST-based approaches

Yang [Yan91] proposes an approach to find the syntactic differences between two procedures; this problem, although not the same as clone detection, is related to it. Yang's approach is to represent each procedure using its AST, and to use dynamic programming to find the largest possible one-to-one matching between nodes in the two ASTs such that the matching preserves nesting relationships (see Section 6.2) as well as the order between sibling nodes. The two procedures are then displayed next to each other, with the non-matching nodes highlighted. This approach is purely syntactic, and is therefore sensitive to the ordering of statements and names of variables.

Baxter et al. [BYM+98] propose an AST-based approach to finding clones in source code. They find exact clones by finding identical AST subtrees, and inexact clones by finding subtrees that are identical when variable names and literal values are ignored. Non-contiguous and out-of-order matches are not found. Their approach completely ignores variable names when asked to find inexact matches; this is a problem because ignoring variable names results in ignoring all data flows, which could result in matches that are not meaningful computations worthy of extraction.

Araújo et al. [ACCP98] propose an AST-based approach to detecting clones in assembly code; the clones they find are expressions that do not span basic-block boundaries.

9.1.3 Metrics-based approaches

Kontogiannis et al. [KDM+96] define an approach that uses dynamic programming, in a manner similar to Yang's approach, to compute a similarity metric between all pairs of `begin-end` blocks in the program. This approach does not find clones in the sense of the other clone-detection approaches discussed so far. It only gives similarity measures, leaving it to the user to go through block pairs with high reported similarity to determine whether or not they are clones. Also, since it works only at the block level it can miss clone fragments that are smaller than a block, and it does not effectively deal with variable renamings or with non-contiguous or out-of-order matches. Other approaches to computing similarity metrics between fragments of code (at various levels of granularity) are [DBF+95, MLM96].

9.1.4 CFG-based approaches

Several CFG-based approaches [KL99, Run00, DEMD00] have been proposed for clone detection and elimination. All these approaches work on intermediate-level code or assembly code. The approach of Kunchithapadam et al. [KL99] finds isomorphic single-entry single-exit subgraphs in basic-block-level CFGs such that corresponding basic blocks have identical sequences of instructions. The approach of Runeson [Run00] is similar to the approach of Kunchithapadam et al., but they allow matching instructions in corresponding basic blocks

to be out-of-order if the instructions can be made in-order via semantics-preserving permutations. They also allow variables names to differ across clones in a group (although they require a one-to-one map between the names). They do not find non-contiguous clones, nor do they find clones that are non-isomorphic in the basic-block level CFGs (i.e., they restrict out-of-order sections to be contained within basic blocks).

We postpone discussion of the approach of Debray et al. [DEMD00] to Section 9.2.4.

9.1.5 Dependence-based approaches

The primary goal of the work described by Bowdidge and Griswold in [BG98] is to help convert procedural code to object-oriented code by identifying methods. As part of this process, they do a limited form of clone detection. We do not discuss the details of their approach here; however, their idea essentially is to find clones by finding matching groups of paths in the PDG such that corresponding nodes in the paths match syntactically. Our approach, in contrast, is to matching *partial slices*; our observation is that most clones that are interesting and worthy of extraction are not simply paths in the PDG. However, their use of paths allows them to present results in a factored form: when different groups of matching paths have common prefixes, they display the common prefixes only once. It is less clear if clones that are partial slices can be presented in factored form.

We reported our slicing-based approach to clone detection in [KH01]. Subsequently, another slicing-based approach [Kri01] for clone detection was proposed by Jens Krinke. The major difference between the two approaches is that while we use backward slicing primarily, and forward slicing along control-dependence edges, they use forward slicing *only*. We have experimented with full forward slicing; our experience was that it often resulted in meaningless clones (several unrelated computations can often be reached by slicing forward from a node that is a dependence predecessor of all of them). Furthermore, forward slicing only can miss interesting computations. Many interesting computations have just one output; such computations can be identified directly by slicing backward from any node that uses

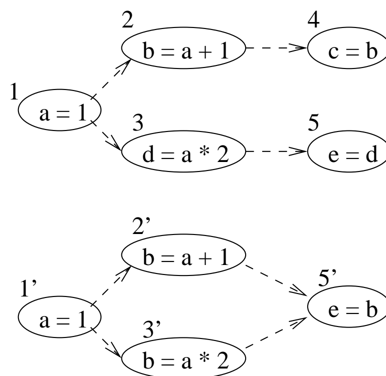


Figure 9.1 PDG subgraphs illustrating Krinke's approach

the output of that computation. Therefore backward slicing is intuitively better suited for finding interesting computations.

There is one other difference between our approach and that of Krinke. Consider the two PDG subgraphs in Figure 9.1 (the dashed edges are flow-dependence edges). Starting from the two matching nodes 1 and $1'$, their approach would map 2 to $2'$ and 3 to $3'$. They then match $5'$ to *both* 4 and to 5 (because $(4, 5')$ is reachable from the matching nodes $(2, 2')$ and $(5, 5')$ is reachable from the matching nodes $(3, 3')$). In fact, their output is simply a pair of subgraphs, that possibly have different numbers of nodes, with no mapping provided between the nodes in the two subgraphs. Our approach, on the other hand, always maps a node in a clone to one specific node in the other clone, making arbitrary choices when multiple possibilities exist. It is our view that the usability of the tool is improved by sticking to the intuitive notion that a clone pair is a one-to-one mapping between two sets of nodes.

Reps and Yang [RY89] showed, for a restricted language, that two program components (statements or predicates) have identical execution behavior (produce identical sequences of "outputs") if the (complete) backward slices taken from those two components are isomorphic. This result holds even if the two components are from different programs. They use this notion not for clone detection, but for *program integration* (the details of which are not important here). In our approach we find matching *partial* slices, as opposed to matching complete slices. We do this because, in the context of clone detection, requiring mapped

nodes to have identical execution behaviors is too strict. For example, in Figure 3.1, no two mapped components have identical execution behaviors (because the `while` loop predicates that control them are non-matching).

9.1.6 Searching for specified patterns

Various approaches have been proposed in the past that can be broadly classified as approaches to searching for specified patterns in a program. This, clearly, is different from but related to clone detection. Paul and Prakash [PP94] propose an approach for letting users search for program fragments using patterns in a pattern language that is obtained by extending the source language. Rich and Wills [RW90] propose a scheme to identify *clichés* in programs. Clichés are frequently used programming idioms; e.g., linear searches, successive-approximation loops. Their approach is based on representing a database of clichés as well as input programs using *plans*, which are hierarchical graphical representations of programs. They then provide techniques for searching the input programs for occurrences of clichés, and also to "parse" the program and build a "derivation tree" whose nodes represent clichés (successively from lower-level clichés to high-level clichés). Griswold and Notkin [GN93] define a tool-box of program analysis and transformation operations. One of the operations they provide is `scope-substitute-call`, which, when given a function, finds other fragments of code in the program that match this function and can potentially be replaced by calls to it; they find the matching fragments by searching for PDG subgraphs that match the PDG of the given function.

9.1.7 Subgraph isomorphism

A number of people have studied the problem of identifying maximal isomorphic subgraphs [BB76, McG82, BSJL92, WZ97]. Since this in general is a computationally hard problem, these approaches typically employ heuristics that seem to help especially when the graphs being analyzed are representations of molecules. In our approach we identify isomorphic partial slices, not general isomorphic subgraphs. We do this not only to reduce

the computational complexity, but also because clones found this way are more likely to be meaningful computations that are desirable as separate procedures.

9.2 Previous work related to procedure extraction

Previous work that is related to our work on procedure extraction falls into three broad categories:

1. Work on eliminating `gotos` in source code,
2. Work on compiler-based loop transformations, and
3. Work on procedure extraction itself.

The problems solved in the first two categories are different from ours. However, our technique of using the variable `exitKind` to handle exiting jumps bears some resemblance to the techniques used by work in the first category (eliminating `gotos`), e.g., [AM72, Oul82]. Similarly, our operation of splitting the tightest e-hammock $\mathcal{H}$ that contains a clone into a sequence of three e-hammocks bears resemblance to the *loop distribution* transformation [Ban93] used by work in the second category above. In this transformation, a single loop is split into several loops with duplication of the loop predicate, and of other predicates in the loop's body, if necessary. However, it is not clear that previously proposed loop-distribution approaches handle exiting jumps. Doing code motion in the presence of exiting jumps is a non-trivial problem that we address. In their context, since the e-hammock they split is always a complete loop, handling exiting jumps is perhaps not as important, because there can be no exiting `breaks` or `continues` in that situation. On the other hand, since in our context an e-hammock can be just a part of a loop body, we can have exiting jumps that are `breaks` or `continues` (`gotos` and `returns` can be exiting jumps in both contexts).

It is worth noting that in our approach we never split loops; i.e., we duplicate only `if` predicates and jumps, and assign a loop in $\mathcal{H}$ entirely to a single resultant e-hammock (we do not enforce this directly – the constraints generated in the presence of a loop indirectly

enforce this). We take this approach because duplicating loops would require duplicating the assignment statements that update the variables used in the loop predicate; this (duplication of assignments) is something we do not do, because, in general, it can require transformations such as renaming variables or copying and restoring entire data structures. Such transformations can reduce code readability and adversely affect the program's efficiency.

A large amount of work has been done in the past regarding procedure extraction; e.g., [Mar80, FMW84, GN93, LDK95, Zas95, LD98, BMD+99, KL99, CM99, CSCM00, Run00, DEMD00]. Of these, the approach of [LD98] is for extracting a single (possibly non-contiguous) fragment of code, whereas the others address extraction of a group of clones. Some of the clone-group extraction approaches work at the source-code level [GN93, BMD+99], whereas the others [Mar80, FMW84, LDK95, Zas95, KL99, CM99, CSCM00, Run00, DEMD00] target assembly code with the aim of compacting it. Our approach is an advance over these previous approaches in two respects:

- It is the first approach, to our knowledge, to address extraction of fragments that contain exiting jumps.

- It employs a range of techniques (code motion, predicate duplication, handling exiting jumps, promotion) to make clone groups that exhibit a variety of difficult characteristics suitable for extraction. Our work is an advance over previous approaches in that we not only employ a wide range of transformations, but also identify appropriate conditions under which to apply each transformation so that results are usually close to ideal. In particular, our approach addresses the non-trivial problem of doing code motion in the presence of exiting jumps.

We now discuss how previous work compares to our work in terms of the two aspects mentioned above.

9.2.1 Exiting jumps

The work of [GN93] is for Scheme programs, and thus does not address programs that contain jumps, whether they are exiting jumps or not. Most previous approaches to procedure extraction handle jumps, but not exiting jumps. For each of the example clones in Figure 1.1, the smallest exiting-jump-free region that contains the marked code is the entire outer `while` loop. The previous approaches would be able to extract this *entire* region, but not just the marked code shown in the figure. While in this particular example it is arguable whether being able to extract the marked code only is a major advantage, in general, the loop could contain a lot of non-matching code in addition to the matching code; if that is the case, extracting the entire loops means that all that non-matching code would have to be placed in the extracted code in guarded form. Worse still, if there were a `return` in that example in place of the `break`, then the region of the clone would be not just the loop, but everything else that follows until the end of the procedure. In our studies we noted that exiting `returns` occur quite frequently in practice (usually to handle error/exceptional conditions).

The approach of Marks [Mar80], which works on assembly code, extracts fragments that contain exiting jumps. However, it is notable that the machine they assume uses a single branch-and-link register to store the return address, which implies that call graphs have maximum depth two. This allows them to leave exiting jumps in original fragments unchanged in extracted procedures, but their solution does not apply to procedure extraction in the source code of a language such as C.

Some of the previous approaches [FMW84, CM99] allow clones to contain exiting jumps in restricted situations where corresponding exiting jumps in the clones in the group have the same target and the last instruction in each clone is an exiting jump. In this case they do not extract the clones into a separate procedure; instead they retain one of the clones in the group and replace all other clones by jumps (rather than calls) to the retained clone. They call this technique *tail merging*. Chen et al. [CLG03] have recently proposed an extended kind of tail merging that handles clones in which corresponding exiting jumps do not have

the same target. If an exiting jump in one clone does not have the same target as the corresponding exiting jump in the other clones, then in the retained clone they replace that exiting jump by conditional jumps to the appropriate targets. This is similar to our `exitKind` transformation, although they incorporate the additional optimization of not introducing a new variable (i.e., `exitKind`) if there is one already available in the program whose value indicates the location from which control entered the retained clone.

While these approaches do handle exiting jumps, tail merging is not a suitable technique for application to source code that is maintained by programmers, because it reduces understandability. Furthermore, tail merging is inapplicable to the source code of a language like C when the clones in a group are in different procedures (because jumps cannot cross procedure boundaries).

One of the previous assembly-code approaches [LDK95] allows extraction of clones that contain multiple entry points (not multiple outside exits). Although this could potentially be simulated in source code by having extra parameters in the extracted procedure and having `gotos` in the beginning of the procedure that transfer control to the appropriate point based on the parameter values, we do not incorporate this technique; it is not clear how useful this technique would be in source code, and moreover it has the disadvantage of producing code that is poorly structured.

9.2.2 Using a range of transformations

Previous approaches to automatic clone-group extraction either employ only a narrow range of techniques, or employ restrictive versions of these techniques, thereby making them unsuitable for extraction of various kinds of difficult clone groups that come up in practice. The approaches proposed in [Mar80, FMW84, LDK95, Zas95, KL99, CM99, CSCM00] do not extract non-contiguous clones or out-of-order groups at all. The approach of Griswold and Notkin [GN93] is capable of extracting out-of-order groups. They provide limited support for extracting non-contiguous clones, via a set of semantics-preserving primitives that the programmer can use to move individual non-clone statements; however they provide no

automatic assistance in determining which statements need to be promoted, and in which direction the others can be moved – i.e., before or after the cloned code. The approach of Runeson [Run00] also is capable of extracting out-of-order groups (but not non-contiguous clones), provided the clones in a group are isomorphic in the basic-block level CFGs (see our earlier discussion of this approach in Section 9.1.4). The approaches of [DEMD00, CLG03] allow non-contiguous clones, although in a more restrictive manner than ours; moreover, they deal with non-contiguous clones simply by promoting all intervening non-clone nodes (they do no code motion). We discuss the approach of Debray et al. [DEMD00] in greater detail in Section 9.2.4. The approach of Balazinska et al. [BMD$^+$99], while incorporating object-oriented techniques for clone extraction in source code, is conceptually similar to the approach of Debray et al.

To be fair, however, it is notable that there is a difference between the motivations of previous clone-group extraction approaches that work on the assembly-code level and that of our approach: whereas the goal of our algorithm is to extract a *given* group of clones that represent a meaningful computation, their goal is to *find and extract* groups of clones that yield space savings. Because of this difference, it might be reasonable for those algorithms to find and extract small, easy subsets of larger, more meaningful clones. However, their techniques are insufficient when dealing with extraction of programmer-specified clone groups in source code (as indicated by some studies of ours, discussed below).

The approach of Lakhotia and Deprez [LD98], which handles single-fragment extraction only, uses a range of transformations, although in a more restrictive fashion than ours. We discuss their approach in detail in Section 9.2.3.

Prior to reporting our current individual-clone extraction algorithm in [KH03], we reported a less powerful approach for the same problem in [KH00]. Our previous algorithm employed code motion to handle non-contiguous clones; however it did not employ the techniques of promotion or duplication of predicates, and it did not handle exiting jumps. As a result, that algorithm is likely to fail on many difficult clones that come up in practice (our current algorithm never fails).

9.2.3 Comparison of our algorithm with Lakhotia's algorithm

The approach of Lakhotia et al. [LD98] is the one that is closest to ours in spirit. They address extraction of a single fragment of code (i.e., their algorithm is comparable to our individual-clone algorithm).

The approach of Lakhotia et al. is to find the tightest (normal) hammock H containing the marked nodes, and to create a *marked* and an *after* hammock from the nodes in H (they do not use a *before* hammock). The key differences between our approach and theirs are:

- They promote all nodes in H that are in the backward slice from the marked nodes. We do not do this, because we can move such code to the *before* bucket (which they do not use).

- We allow dataflow from the *marked* hammock to the *after* hammock. They disallow this, and instead place in the *after* hammock all nodes in H that are in the backward slice from unmarked/unpromoted nodes. This can cause duplication of the marked code in the *after* bucket, thereby defeating the purpose of extraction.

- Our use of code motion is better than theirs, and so is our use of promotion. As a result we always succeed in transforming the marked code to make it extractable; on the other hand they fail whenever their *marked* and *after* buckets have a common output variable.

- They do not handle exiting jumps, and therefore have to start from the tightest hammock containing the marked code. The tightest hammock is usually larger (and never smaller) than the tightest e-hammock, which means they have more unmarked nodes to deal with, which exacerbates all the problems mentioned earlier.

- They do allow duplication of assignments, and saving and restoring variable values (although they do not address the difficult issues that come up in this context when arrays and pointers are present). Our approach duplicates only predicates and does not save and restore values. Although these features of their approach can potentially

Category	total # clones	# non contig.	# exiting jumps
Both outputs non-ideal	3	3	1
Their output non-ideal, ours ideal	15	5	11
They fail, we succeed non-ideally	3	3	2
They fail, our output ideal	22	19	11
	43		

Figure 9.2 Comparison of our algorithm and Lakhotia's algorithm

make it better than ours in some cases, it can also increase duplication of marked code. In practice, their other drawbacks outweigh these features, as indicated by our studies of the comparative performance of their algorithm with ours (discussed below).

To illustrate some of the advantages of our approach over that of Lakhotia et al., consider the clone group from *bison* shown in Figure 3.1. When applied to the clone in Fragment 1, their algorithm promotes the intervening non-clone `if` statement (because the definition of `c` in that statement reaches a subsequent use of `c` in a cloned node). When applied to the clone in Fragment 2, their algorithm fails (they place the assignment "`c = getc(finput)`" in both the *marked* and the *after* buckets, and `c` is an output variable in both buckets, because of data flow in the original code from the assignment "`c = getc(finput)`" to the use the of `c` in the `while` predicate). On the other hand our algorithm does the ideal thing on both these clones: move the intervening non-clone nodes out of the way of the cloned nodes.

Similarly, on the example in Figure 1.1, because of the exiting jumps, they can only extract the two entire loops.

Figure 9.2 provides data comparing the performance of our individual-clone algorithm and Lakhotia's algorithm, on the clones in our dataset of Chapter 8. We performed the comparison by (partially) implementing their algorithm, using the same CodeSurfer-based framework that we used for the implementation of our algorithm. In Figure 9.2 and in the

following discussion we talk about difficult clones only, because no transformation is required by either algorithm to make the non-difficult clones extractable. The 43 difficult clones are divided into four disjoint categories (based on the performance of the two algorithms on the clones), with one category per row. The first row is for clones on which both algorithms succeeded but produced non-ideal output; the second row is for clones on which our algorithm produced ideal output whereas theirs produced non-ideal output; the third row is for clones on which their algorithm failed and our algorithm succeeded but produced non-ideal output, and the fourth row is for clones on which they failed while we produced ideal output. Their algorithm did not produce the ideal output on even one clone in the dataset; and on all but 3 clones (those in the first row) their algorithm performed worse than ours. An important reason for this is that they do not handle exiting jumps: They failed on 8 clones (on which our algorithm succeeded) and performed non-ideally on 7 clones (on which our algorithm performed ideally) solely because of exiting jumps; i.e., if the exiting jumps were removed they would succeed on the 8 clones, and perform ideally on the 7.

However, handling exiting jumps is not the only advantage of our algorithm over theirs; our notion of when unmarked nodes can be moved away is less restrictive than theirs, and our rules for promotion are better (e.g., recall the performance of their algorithm on the two non-contiguous clones in Figure 3.1). There are 20 clones (other than the 15 mentioned in the previous paragraph) on which they perform unsatisfactorily due to reasons other than exiting jumps (i.e., these clones either have no exiting jumps, or the problem persists even if all exiting jumps are removed). In particular, they failed on 16 clones in this category, and performed non-ideally on the remaining four. In contrast, our algorithm performed ideally on 17 of these 20 groups (and produced non-ideal output on the remaining 3).

9.2.4 Comparison of our algorithm with Debray's algorithm

We selected the assembly-code compaction approach of Debray et al [DEMD00], from among the various previously reported clone-group extraction approaches, for a more detailed comparison with our approach. The reason we selected their approach is that it employs more

techniques than other previous assembly-code compaction approaches, is conceptually similar to the source-code based approach of [BMD+99], and is likely to perform better than the other source-code based approach [GN93] because groups involving non-contiguous clones, whose extraction is addressed by [DEMD00] but not by [GN93], occur more frequently in our dataset than out-of-order groups that are handled better by [GN93]. (The approach of Debray et al. does not, strictly speaking, employ more techniques than that of Runeson [Run00]; Runeson's approach allows out-of-order matches, but as with the approach of [GN93], it disallows non-contiguous clones.)

The basic approach of Debray et al. works as follows: For each procedure, they build a CFG in which the nodes represent basic blocks. They then find groups of isomorphic single-entry single-exit subgraphs in the CFGs such that corresponding basic blocks have identical instruction sequences (modulo register renamings), and then extract each group into a new procedure.

In an extension to their basic approach, they allow corresponding basic blocks to have non-identical instruction sequences; in that case they walk down the two sequences in lock-step and "promote" every mismatching instruction (i.e., it is included in the extracted procedure with a guard). Thus, although inexact matches are allowed, every mismatch is handled by using the guarding mechanism: every intervening non-matching statement and every copy of an out-of-order matching statement is placed in the extracted procedure with guarding. (Although they propose this extension, they disable it in their experiments because it hurt performance).

In addition to the fact that they use guarding to handle all mismatches, their approach has two other weaknesses compared with ours:

1. The requirement that the CFG subgraphs be isomorphic prevents many reasonable clone groups from being extracted; e.g., they would fail to extract the clone group in Figure 3.1 because the intervening non-matching statement "`if (c == '-') c = '_'`" makes the four basic-block-level CFG subgraphs that contain the four clones non-isomorphic.

	Category	# clone-groups
1.	They fail (ours ideal)	9
2.	Their output non-ideal (ours ideal)	6
3.	They fail (our output non-ideal)	2
4.	Both outputs non-ideal	1
5.	Both fail	3
6.	Both outputs ideal	4
7.	Their output ideal (ours non-ideal)	1
8.	We fail (their output non-ideal)	1
		27

Figure 9.3 Comparison of our algorithm to that of Debray et al.

2. Because they are restricted to extracting single-entry single-exit structures, they cannot handle exiting jumps. For instance, in the example of Figure 1.1, due to the presence of the `breaks`, the smallest single-entry single-exit structure enclosing each clone is the entire surrounding loop. Therefore they could extract the entire loop, but not just the desired clones.

Figure 9.3 provides data comparing the performance of our algorithm and that of Debray et al. on the 27 difficult clone groups in our dataset of Chapter 8. We did not implement their algorithm for this comparison; however, because they do no code motion and instead promote all intervening non-clone nodes, we could manually apply their algorithm in a straightforward manner.

The 27 difficult clone groups are divided into 8 disjoint categories, with one per row in Figure 9.3. As shown in rows 1 through 5 and 8, their algorithm either fails or performs non-ideally on a vast majority of the clone groups, 22 out of 27, while our algorithm fails on none and produces non-ideal output on only 8 of those 27 groups. The main reason for the better performance of our algorithm is that, as discussed earlier, it employs a variety of transformations to tackle difficult aspects, while their algorithm uses promotion only.

As shown in the last two rows of Figure 9.3, their algorithm performs better than ours on 2 clone groups. On one of these groups we perform non-ideally by over-aggressively moving intervening non-clone nodes out of the way using duplication of predicates, whereas guarding, which is their solution, is the ideal outcome. The other group is the out-of-order group mentioned in Section 8.4.1 on which our algorithm fails; they succeed (non-ideally) on this group by using guarding.

Since we reported our individual-clone extraction algorithm in [KH03], De Sutter et al. [SBB02] have proposed an approach to clone detection and elimination; this approach is quite similar to that of Debray et al., except that they also find exactly matching sequences of instructions that are parts of basic blocks (the approach of Debray et al. treats entire basic blocks as the units of clone detection).

Chapter 10

Conclusions

Code duplication is a widespread problem in real programs. Duplication is usually caused by copy-and-paste: a new feature that resembles an existing feature is implemented by copying and pasting code fragments, perhaps followed by some modifications. Duplication degrades program structure. Detecting clones (instances of duplicated code) and eliminating them via procedure extraction gives several benefits: program size is reduced, maintenance becomes easier (bug fixes and updates done on a fragment do not have to be propagated to its copies), and understandability is improved (only one copy has to be read and understood).

In this thesis, we focused on the detection and extraction of *inexactly* matching groups of clones, i.e., groups whose individual clones are non-contiguous, groups in which matching statements are in different orders in different clones, and groups where variable names are not identical in all clones. Our first contribution was a novel program-dependence-based approach that identifies duplication by finding matching "partial" slices in PDGs. This approach is an advance over previous approaches to clone detection that work on source text, CFGs, or ASTs, in its ability to find inexact matches. The approach also has the benefit of being likely to identify clones that are good candidates for extraction into separate procedures (are meaningful computations, and are extractable).

Non-contiguous clones, and groups where matching statements are out-of-order, are non-trivial to extract. *Exiting jumps* – jumps from within the code region that contains a clone to outside that region – also complicate extraction, because after extraction control flows out of the procedure-call to a single statement, the statement that follows the call.

Semantics-preserving transformations are required when difficult characteristics such as non-contiguity, out-of-order matches, and exiting jumps are present, so that each clone becomes a contiguous well-structured block that is suitable for extraction, and so that matching statements are in-order across the group. The second contribution of this thesis was a pair of algorithms, one for making an individual fragment extractable, and one for making a group of clones extractable. These algorithms are an advance over previous work on procedure extraction in two ways: they are the first to handle exiting jumps, and they employ a range of semantics-preserving transformations to make clone groups that exhibit a variety of difficult characteristics extractable. We provide proofs of semantics preservation for both our extractability algorithms.

We have implemented our clone-detection algorithm, and the heart of our single-fragment extractability algorithm. We have experimented with the clone-detection algorithm on several real programs, and have found that it is likely to identify most clones that a programmer would consider interesting, and only a few clones that a programmer would consider uninteresting (mainly at small sizes). The main drawback of the approach is that it often identifies multiple variants of an "ideal" clone group, instead of just the ideal groups. Therefore, the programmer needs to examine the output of the tool and determine the ideal clone groups that the reported clone groups correspond to. However, in our experience, this is not an overwhelming burden.

We have also experimented with our extractability algorithms, using a dataset of clone groups identified by our clone-detection tool. We found that the algorithms produced the "ideal output" – the best output according to our judgment – over 70% of the time. Considering that no automatic algorithm is likely to incorporate the full sophistication of a programmer, we regard these results as very encouraging. Furthermore, when compared to two previously reported approaches to procedure extraction, we found that our algorithm outperformed theirs on a vast majority of the inputs.

Future work on the clone-detection approach could include devising heuristics beyond those currently proposed that reduce the "variants" problem. Engineering efforts can be

made to speed up the tool, and an extension to the approach could be developed to directly identify groups of clones, rather than identifying clone pairs first and then grouping them, as we do now. From the extraction perspective, future work could include experimental studies involving programmers (other than the author) to further evaluate the usefulness of the approach. Future work could also include devising and evaluating a scheme to determine the parameters that are needed by a procedure that replaces a group of clones. Finally, in addition to making these follow-on improvements, it would be interesting to think about higher-level applications for the techniques proposed in this thesis. In particular, it might be worthwhile to investigate (semi) automated approaches that make use of clone detection, procedure extraction and perhaps other transformations as underlying tools with the goal of improving the overall structure and maintainability of a program (in all aspects).

DISCARD THIS PAGE

Appendix A: The partitioning in the individual-clone algorithm satisfies all constraints

Theorem A.1 The partitioning of nodes (into buckets *before*, *marked* and *after*) done in Step 4 of the individual-clone algorithm (Figure 5.11, Chapter 5) satisfies all constraints generated in Step 2 of that algorithm (Figures 5.5 and 5.6).

We first recall a few details of the individual-clone algorithm. Step 4 partitions the nodes in $\mathcal{H}$ into the three buckets *before*, *marked*, and *after*. Copies of a predicate node in $\mathcal{H}$ may be present in multiple buckets, although a single bucket has at most one copy of a node. The ordering of the three buckets is *before*, then *marked*, then *after*. There are three kinds of constraints. A constraint $p \leq q$ is satisfied iff no copy of p is in a bucket that follows H, where H is the earliest bucket that contains a copy of q. A constraint $p \Rightarrow q$ is satisfied iff every bucket that has a copy of p also has a copy of q. A constraint $p \leadsto q$ is satisfied iff a copy of q is present either in H or in some bucket that follows H, where H is the last bucket that contains (a copy of) p.

While any constraint can be satisfied (or not satisfied) only at the end of Step 4 (when partitioning is complete), "$\leq$" constraints can be *violated* at intermediate points in the execution of Step 4. A constraint "$p \leq q$" is said to be violated at an intermediate point if a copy of p is present in some bucket that follows the first bucket to have a copy of q. Note that a "$\leq$" constraint cannot be violated if one or both nodes involved in the constraint are not present in any bucket yet; also, a constraint that is violated at some intermediate point remains violated from then on, despite any other assignments done later in the step. "$\Rightarrow$" and "$\leadsto$" constraints can never be violated at intermediate points in Step 4; intuitively, the reason for this is that while "$\leq$" constraints rule out certain assignments, "$\Rightarrow$" and "$\leadsto$" constraints rule nothing out (they only *require* certain properties).

The first sub-step in Step 4 is to assign the marked (and promoted) nodes to the *marked* bucket (see Figure 5.11). No constraints are violated at the end of this sub-step, because

"$\Rightarrow$" and "$\leadsto$" constraints can never be violated, and because "$\leq$" constraints cannot be violated when only the *marked* bucket is non-empty.

The next sub-step in Step 4 is the `repeat` loop. The node n assigned in the "`if..else if..`" statement in that loop is called the *initiator* of that iteration of the loop. Lemmas A.2 and A.3 concern this loop.

Lemma A.2 Consider any iteration of the `repeat` loop in Step 4 (Figure 5.11), such that:

- an initiator node n is assigned to a bucket B in the "`if..else if..`" statement in that loop (B is *before* or *after*).
- the nodes assigned in previous iterations did not cause any constraints to be violated.

The assignment of n to B does not cause any constraints to be violated.

PROOF OF LEMMA A.2. For contradiction, assume that the assignment of n to B violates a constraint c. Since only "$\leq$" constraints can be violated, and since no constraint of the form $p \leq q$ can be violated when p is in *before* or q is in *after*, c has to have one of the following forms: $n \leq b, n \leq m, a \leq n, m \leq n$, where a is a node that is already in *after*, b is a node that is already in *before*, and m is a node that is already in *marked*. We consider each of these cases below, and in each case show that a contradiction results.

Case (c is of the form $n \leq b$, where b is a node in *before*, or of the form $n \leq m$, where m is a node in *marked*): In this case B = *after* (otherwise c is not violated). Note that according to Rules B1 and B2 in Figure 5.10, c forces n into *before*. The only possible reason why n was placed in *after* in spite of this is that some other constraint c_2 forces n into *after*. Here are the possibilities for the form of c_2 (basically, these are obtained from rules A1-A4 in Figure 5.10, respectively):

Case (c_2 is of the form $a \leq n$, where a is a node in *after*): c is of the form $n \leq b$ or $n \leq m$. Therefore, by the first rule for generating extended constraints (see Figure 5.6), one of the two extended constraints $a \leq b, a \leq m$ exists. Each of

these extended constraints is violated even before n is assigned (because a, b, and m are respectively in *after*, *before*, and *marked*). This is a contradiction of the statement of this Lemma.

Case (c_2 is of the form $m_2 \leq n$, where m_2 is a node in *marked*): If c is of the form $n \leq b$, the extended constraint $m_2 \leq b$ exists. This constraint was violated even before n was assigned, which, as we noted earlier, is a contradiction.

On the other hand, if c is of the form $n \leq m$, then the first promotion rule in Figure 5.9 would have caused n to have been promoted; i.e., a copy of n is already in the *marked* bucket. But in this case $m_2 \leq n$ could not have forced n into *after* (Rule A2 applies only when n is not already present in the *marked* bucket). This is a contradiction of our earlier claim that c_2 forces n into *after*.

Case (c_2 is of the form $m_2 \rightsquigarrow n$, where m_2 is a node in the *marked* bucket): This constraint exists because m_2 is an antecedent of n. If c is of the form $n \leq b$, the fourth rule in Figure 5.6 applies, which means that the extended constraint $m_2 \leq b$ exists. This constraint was violated even before the assignment of n, which is a contradiction.

On the other hand, if c is of the form $n \leq m$, then the second promotion rule in Figure 5.9 would have caused n to have been promoted; i.e., a copy of n is already in the *marked* bucket. But in this case the constraint $m_2 \rightsquigarrow n$ does not force n into the *after* bucket, which contradicts our earlier claim that c_2 forced n into *after*.

Case (c_2 is of the form $a \rightsquigarrow n$, where a is a node that is already in the *after* bucket): That is, a is an antecedent of n. If c is of the form $n \leq b$ ($n \leq m$), then the fourth rule in Figure 5.6 applies, giving rise to the extended constraint $a \leq b$ ($a \leq m$). Both these extended constraints were violated even before n was assigned, which is a contradiction.

Case (c is of the form $a \leq n$, where a is a node in *after*, or of the form $m \leq n$, where m is a node in *marked*): In this case $B =$ *before* (otherwise c is not violated). Note that according to Rules A1 and A2 in Figure 5.10, c forces n into *after*. The only possible reason why n was placed in *before* in spite of this is that some other constraint c_2 forces n into *before*. Here are the possibilities for the form of c_2 (basically, these are obtained from Rules B1 and B2 in Figure 5.10, respectively):

Case (c_2 is of the form $n \leq b$, where b is a node in *before*): c is of the form $a \leq n$ or $m \leq n$. Therefore, by the first rule for generating extended constraints (Figure 5.6), one of the two extended constraints $a \leq b, m \leq b$ exists. Each of these extended constraints is violated even before n is assigned (because a, b, and m are respectively in *after*, *before*, and *marked*). This is a contradiction of the statement of this Lemma.

Case (c_2 is of the form $n \leq m_2$, where m_2 is a node in *marked*): If c is of the form $a \leq n$, the extended constraint $a \leq m_2$ exists. This constraint was violated even before n was assigned, which is a contradiction.

On the other hand, if c is of the form $m \leq n$, then the first promotion rule in Figure 5.9 would have caused n to have been promoted; i.e., a copy of n is already in the *marked* bucket. But in this case $n \leq m_2$ could not have forced n into *before* (Rule B2 applies only when n is not already present in the *marked* bucket). That is a contradiction of our earlier claim that c_2 forced n into *before*.

□

Lemma A.3 No constraints are violated at any intermediate point in the execution of the `repeat` loop in Step 4 (see Figure 5.11).

Proof of Lemma A.3. The proof is by induction on the number of nodes assigned in the loop so far to buckets. Each iteration of the loop consists of the assignment of the initiator node n of that iteration to a bucket (in one of the two branches of the `if` statement),

followed by the assignment of the non-initiator nodes (control-dependence ancestors of the initiator).

Base case

We need to prove that the first node n assigned in the loop causes no constraints to be violated. No constraint can be violated when just the *marked* bucket is non-empty. In other words, no constraints were violated just before assignment to n to *before*/*after*. Therefore Lemma A.2 applies, which implies that the assignment of n resulted in no violations of any constraints.

Inductive case

The inductive hypothesis is that a certain number of nodes have already been assigned to *before* and *after*, and that these assignments cause no constraints to be violated. There are two sub-cases under the inductive case: the node currently being assigned is an initiator, and the node currently being assigned is not an initiator.

We first consider the case where the current node n is an initiator, and is assigned to a bucket B. Because no constraints were violated prior to the assignment of n to B (inductive hypothesis), Lemma A.2 applies, and implies that the assignment of n to B results in no violations of any constraints.

We now consider the second case case, where the currently assigned node p is a non-initiator, is assigned to a bucket B, and is a control-dependence ancestor of an initiator node n that was assigned to B at the beginning of the current iteration of the loop. For contradiction, say the assignment of p violates a constraint c; c has to be of the form $p \leq q$ ($q \leq p$), where q is a node was earlier assigned to some bucket. Since p is a control-dependence ancestor of n, there exists a sequence of control-dependence constraints $n \Rightarrow q_1 \Rightarrow q_2 \Rightarrow \cdots \Rightarrow p$; therefore, there exists an extended constraint $n \leq q$ ($q \leq n$). Since n and p are both in B, $p \leq q$ ($q \leq p$) is violated implies that $n \leq q$ ($q \leq n$) is also violated. Since n and q both were assigned before p, $n \leq q$ ($q \leq n$) was violated before the assignment of p. This contradicts the inductive hypothesis, and therefore we are done. □

Proof of Theorem A.1.

We now show that every constraint generated in Step 2 of the algorithm (Figures 5.5 and 5.6) is satisfied at the end of Step 4 (Figure 5.11).

"$\leq$" constraints: Lemma A.3 showed that none of these constraints are violated at the end of Step 4. Therefore, all these constraints are satisfied at the end of this step (some of the "normal" predicates in $\mathcal{H}$ have possibly not been assigned to any bucket at the end of the step; "$\leq$" constraints that mention such unassigned nodes are *trivially satisfied*).

"$\Rightarrow$" constraints: Each of these constraints is satisfied, because whenever a node is assigned to a bucket, its control-dependence ancestors are also assigned to the same bucket.

"$\leadsto$" constraints: These are satisfied at the end of Step 4, because Rules A3 and A4 (Figure 5.10) ensure that the appropriate nodes are forced into buckets as long as unsatisfied "$\leadsto$" constraints remain. Recall that the `repeat` loop in this step does not terminate until no forced nodes remain.

□

Appendix B: The individual-clone algorithm is semantics-preserving

Recall that the individual-clone algorithm (Chapter 5) identifies the e-hammock $\mathcal{H}$ that contains the marked nodes, and transforms this e-hammock into the output e-hammock $\mathcal{O}$. In this section we prove that this transformation is semantics-preserving; i.e., we show that the original e-hammock $\mathcal{H}$ and the resultant e-hammock $\mathcal{O}$ produced by the algorithm are semantically equivalent. Recall that Step 6 of the algorithm has two substeps: In the first substep each copy of exiting jump except for its final copy is converted into a `goto`; in the second substep exiting jumps in the *marked* bucket are converted into `gotos`, and compensatory code is added to the *after* bucket. In this section, we assume that $\mathcal{O}$ is the resultant e-hammock as it is at the end of the first substep of Step 6 of the algorithm; our assumption is justified because the second substep of Step 6 is obviously semantics-preserving.

Recall that there are three e-hammocks in $\mathcal{O}$: *before*, *marked* and *after*. In this section, whenever we refer to an e-hammock H *in* $\mathcal{O}$, we mean that H is either the *before*, or *marked*, or *after* e-hammock.

Theorem B.1 A *program state* is a tuple of values, with one value for each variable in the program. Let $E_{\mathcal{H}}$ be an execution of the e-hammock $\mathcal{H}$ (treating $\mathcal{H}$ as if it were a complete program) starting from some program state s. Similarly, let $E_{\mathcal{O}}$ be the execution of the e-hammock $\mathcal{O}$ from the same starting state s. Each of the two executions is a sequence of (dynamic) *instances* of the nodes in $\mathcal{H}$. The two executions satisfy the following properties:

- the program states at the conclusions of the two executions are identical, and
- control flows out of $\mathcal{H}$ and $\mathcal{O}$, at the conclusions of the two executions respectively, to the same node outside $\mathcal{H}$ (in the containing CFG).

In other words, $\mathcal{H}$ and $\mathcal{O}$ have identical semantics.

In the rest of this section we state and prove three key properties, *ConstraintsSat*, *IdentExecs*, and *DefsReached*. We finally use these three properties to prove Theorem B.1.

Definition 7 (Actual definitions) Recall that, as stated in Chapter 2, the *def* (*use*) set of a node n in $\mathcal{H}$ is a statically computed over-approximation of the set of variables that may be defined (used) at that node. An instance of n within one of the two executions $E_\mathcal{H}, E_\mathcal{O}$ *actually defines* a variable if that variable is actually assigned to by that instance. Therefore, the *actually defines* set of an instance of n is a subset of the *def* set of n; also, the *actually defines* set contains more than one variable only if the expression inside n includes procedure calls.

We now introduce terminology that we use throughout the proof:

- A node n is said to use (define) a variable v iff v is in n's statically computed *use* (*def*) set.
- An instance of a node n is said to use a variable v iff n uses v (i.e., by definition, *use* sets of instances are identical to the use sets of the corresponding nodes).
- An instance of a node n is said to define a variable v iff that instance *actually defines* v (i.e., as far as instances are concerned, we use "define" as shorthand for "actually define").
- An (actual) definition of a variable v in an instance i_m in an execution ($E_\mathcal{H}$ or $E_\mathcal{O}$) is said to *reach* some instance i_n that occurs somewhere after i_m in that same execution iff no other instance between i_m and i_n in that execution (actually) defines v.
- If a node n uses a variable v, then the value of v *consumed* by an instance of n is the value of variable v when that instance begins execution. Two instances of a node n are said to consume *identical values* if, for every variable v in the *use* set of n, the values of v consumed by the two instances are identical.

We now state and prove Property *ConstraintsSat*.

Property *ConstraintsSat*. Let i_m and i_n be any two instances in the execution $E_{\mathcal{H}}$ such that i_m comes before i_n, and such that both instances define some variable v or one defines v and the other uses v. Let m and n be the two nodes in $\mathcal{H}$ of which i_m and i_n are instances, respectively. No copy of of m is present in an e-hammock in $\mathcal{O}$ that follows any e-hammock in $\mathcal{O}$ that contains a copy of n. (Recall that the ordering of the three e-hammocks in $\mathcal{O}$ is *before*, *marked*, and *after*; also note that at least one of the two nodes m, n defines a variable, and is therefore present in only e-hammock in $\mathcal{O}$.)

PROOF OF PROPERTY *ConstraintsSat*. By considering the various cases for i_m and i_n, we show that the constraint $m \leq n$ is generated in Step 2 of the algorithm. Property *ConstraintsSat* then follows automatically, because the partitioning in Step 4 of the algorithm (as described in Figure 5.11) satisfies all constraints (Theorem A.1).

Case (i_m and i_n both define v): Because i_m precedes i_n in $E_{\mathcal{H}}$, n is *output dependent* on m, with the dependence induced by a path contained in $\mathcal{H}$. Therefore the constraint $m \leq n$ is generated (see the first rule in Figure 5.5).

Case (i_m uses v and i_n defines v): Because i_m precedes i_n in $E_{\mathcal{H}}$, n is *anti dependent* on m, with the dependence induced by a path contained in $\mathcal{H}$. Therefore the constraint $m \leq n$ is generated (first rule in Figure 5.5).

Case (i_m defines v and i_n uses v):

Case (an instance i_s of some node $s \in \mathcal{H}$ is in between i_m and i_n in $E_{\mathcal{H}}$ such that v belongs to the *def* set of s): Let i_p be the last instance in $E_{\mathcal{H}}$ in between i_m and i_n such that v belongs to the *def* set of p, where p is the node of which i_p is an instance. p is output dependent on m, and n is flow dependent on p. Therefore the constraints $m \leq p$ and $p \leq n$ are generated. These two constraints result in the generation of the extended constraint $m \leq n$ (first rule in Figure 5.6).

Otherwise: n is flow dependent on m. Therefore the constraint $m \leq n$ is generated.□

Our goal now is to prove Property *IdentExecs*. We work towards that by stating three lemmas (and proving the second and third of these).

Lemma B.2 Let q be any node that is neither the *entry* node nor the *exit* node of a CFG. Let
$S = [(p_1 = entry) \rightarrow p_2 \rightarrow \cdots \rightarrow p_m \rightarrow q]$ be a path in the CFG (m could be equal to 1 in which case the path is simply $[(p_1 = entry) \rightarrow q]$). There exists an integer $k, 1 \leq k \leq m$ such that:

1. p_k is a predicate node in S (recall that the *entry* node is a predicate, too), and

2. the edge from p_k to its successor in S is labeled C, where C is either "true" or "false", and

3. q postdominates only the C-edge of p_k; i.e., q is C-control dependent on p_k, and

4. for all $l, k < l \leq m$: q postdominates p_l

Lemma B.3 Let H be any e-hammock in $\mathcal{O}$ (H is *before*, *marked*, or *after*). A node p in $\mathcal{H}$ is called an H-node if (a copy of) p is present in H (a copy of p could also be present in some other e-hammock L in $\mathcal{O}$, in which case p is also an L-node). Let t be any non-H node in $\mathcal{H}$.

1. At most one H-node in $\mathcal{H}$ can be reached *first* (i.e., without going through other H-nodes) along paths in $\mathcal{H}$ starting at t.

2. Moreover, if an H-node h can be reached by following paths in $\mathcal{H}$ from t, then there is no path from t that leaves $\mathcal{H}$ without going through h.

Proof of Lemma B.3.

We first prove the first statement in the lemma. The proof is by contradiction. That is, assume there exist two H-nodes p and q in $\mathcal{H}$ such that there exist paths
$P_1 = [(p_1 = t) \rightarrow p_2 \rightarrow \cdots p_m \rightarrow p]$ and $Q_1 = [(q_1 = t) \rightarrow q_2 \rightarrow \cdots q_n \rightarrow q]$, such that both

paths are contained within $\mathcal{H}$ and such that each of the p_is and q_is is a non-H node. Since every node is assumed to be reachable from the *entry* node of the CFG, clearly there is a path P from *entry* to t. Applying Lemma B.2 on the path $P + P_1$, we infer that either p postdominates t, or p is control dependent on some node p_i in P_1 that precedes p. This second case actually cannot hold, for the following reason: Whenever a node is placed in a bucket the algorithm places copies of all its control ancestors in that bucket; this contradicts our starting assumption that p is a H-node whereas every other node on P_1 is a non-H node. Therefore we have shown that p postdominates t. Applying a similar argument on the path $P + Q_1$ we can show that q also postdominates t. Because each of the two nodes p, q can be reached from t without going through the other (via paths P_1, Q_1 respectively), the previous postdomination result implies that both p and q postdominate each other. However, that is possible only if $p = q$. Therefore we have proved the first statement in the lemma by contradiction.

We now prove the second statement in the lemma, again by contradiction. Say an H-node h is first reached from t via a path that is contained in $\mathcal{H}$. Repeating the earlier argument, we can show that h postdominates t. Consider the path from t that leaves $\mathcal{H}$ without going through h. Since h postdominates t, h postdominates the edge e in this path that actually leaves $\mathcal{H}$. But once control leaves $\mathcal{H}$, the only way to reach h is by re-entering $\mathcal{H}$ through its entry node. Therefore the entry node of $\mathcal{H}$ postdominates e. This actually is impossible, by the following reasoning: e is either the executable edge out of an exiting jump, or e is a fall-through edge to the fall-through exit node of $\mathcal{H}$. The first case cannot be true because $\mathcal{H}$ has no backward exiting jumps, while the second case cannot be true in any CFG (because of the presence of the non-executable edges, the entry node of a block or block-sequence can never postdominate the fall-through exit node of that block/block-sequence). $\Box$

Definition 8 (*first_reached*) Let m be an H-node in $\mathcal{H}$, let e be a CFG edge out of m (e is labeled *true* or *false* if m is a predicate), and let t be the target of the edge e. We define the node *first_reached*(m, e, H), using four cases.

Case (t is an H-node in $\mathcal{H}$): *first_reached*$(m, e, H) = t$.

Case (t is a non-H node in $\mathcal{H}$ and h is the unique H-node in $\mathcal{H}$ that is reached first from t along paths in $\mathcal{H}$): *first_reached*$(m, e, H) = h$.

Case (t is a non-H node in $\mathcal{H}$ and no H-node in $\mathcal{H}$ is reachable from t along paths in $\mathcal{H}$): *first_reached*(m, e, H) does not exist.

Case (t is outside $\mathcal{H}$): *first_reached*(m, e, H) does not exist.

Lemma B.4 Let H be any e-hammock in $\mathcal{O}$ (H is *before*, *marked* or *after*). A *maximal non-H subgraph* is defined as a subgraph of $\mathcal{H}$ that consists only of nodes not in H, and that satisfies two properties:

- for each CFG edge $p \rightarrow q$ in $\mathcal{H}$ such that neither p nor q is in H, if one of p, q belongs to the subgraph then the other one belongs to it, too.
- Treating all edges in the CFG as undirected, the subgraph is connected.

The statement of this lemma is that each maximal non-H subgraph has exactly one of the following two properties:

- there is a unique t outside the subgraph such that every edge leaving the subgraph goes to t, t is in $\mathcal{H}$, and t is in H. Clearly, t is the unique H-node that is first reached from each node in the subgraph.
- no nodes in $\mathcal{H}$ that also belong to H are reachable from nodes in the subgraph.

Proof of Lemma B.4. Let G be any maximal non-H subgraph in $\mathcal{H}$. We define an *end point* of G as:

- either an H-node m in $\mathcal{H}$ such that there is an edge in $\mathcal{H}$ from some node in G to m, or
- some node n in G such that the target of some edge leaving n is outside $\mathcal{H}$.

G may have a number of end points; we call an end point of the first kind an H end-point, and an end point of the second kind a non-H end-point. For any end point p we define the *reachability set* of p to be the set of nodes in G from which there is a path in G to p (if p is a non-H end-point, then by definition it belongs to its own reachability set). Every end point of G clearly has at least one node in its reachability set. Furthermore, every node in G belongs to the reachability set of at least one end point of G. (For any node t in G consider any path from t to the exit node of the CFG; if an H-node is encountered on this path then the first such node is the end point of G to whose reachability set t belongs; else t belongs to the reachability set of the last node in this path that belongs to G.)

If G has no H end-points, or one H end-point and no non-H end-points, then verify that it satisfies the lemma; therefore we are done.

We now show that G cannot have more than one H end-point, or one H end-point and some non-H end-points. For contradiction, let m be an H end-point of G, and let G have at least one other end point. Let M be the reachability set of m. We have two cases for M.

The first case is when every node in G is also in M. In this case, any node r in G that is in the reachability set of some end point n other than m (there is at least one such node r) is also in the reachability set of m. We return to this case later.

The remaining case is when M is a strict subgraph of G. Recall that G is a maximal non-H subgraph of $\mathcal{H}$; therefore, by definition, when all edges are treated as undirected edges G is a connected subgraph of $\mathcal{H}$; therefore, M being a strict subgraph of G, we infer that there exists an edge $e = r \rightarrow s$ such that both r and s are in G but only one of them is in M. If s belongs to M then r would have to belong to M too (because there would be a path from r to m via s); therefore r, but not s, belongs to M. Therefore s belongs to the reachability set of some other end point n of G (recall that every node in G belongs to the reachability set of some end point). Therefore, due to the edge $r \rightarrow s$, r too belongs to the reachability set of n.

In other words, we have shown that in both cases some node r in the reachability set of m is also present in the reachability set of another end point n. That is, the H-node

m is reached first from r along paths contained in $\mathcal{H}$, and either n is another H-node that is reached first from r or there is a path in $\mathcal{H}$ from r that leaves $\mathcal{H}$ (via n) without going through m. Neither of this can be possible (according to Lemma B.3). Therefore G cannot have more than one H end-point, or one H end-point and some non-H end-points. □

Recall that each e-hammock H in $\mathcal{O}$ is created (in Step 5 of the algorithm) by making a copy of $\mathcal{H}$ and removing from that copy maximal non-H subgraphs. Recall also that a subgraph that has a single H end-point is removed by redirecting all edges coming into the subgraph to that end point, whereas a subgraph that has no H end-points is removed by redirecting all edges coming into it to the fall-through exit of H. Therefore, we have the following corollary of Lemma B.4.

Corollary B.5 Let H be an e-hammock in $\mathcal{O}$.

1. If the entry node n of $\mathcal{H}$ is an H-node then n is also the entry node of H; else the unique H-node in $\mathcal{H}$ that is first reached from n along paths in $\mathcal{H}$ is the entry node of H.

2. Let m be any H-node in $\mathcal{H}$, and e be a CFG edge out of m in $\mathcal{H}$. The target of the same edge e in H is:

   - outside H, if *first_reached*(m, e, H) does not exist
   - equal to the node *first_reached*(m, e, H) in H, if *first_reached*(m, e, H) exists

Recall the terminology we introduced earlier: s is some program state, $E_{\mathcal{H}}$ is the execution of $\mathcal{H}$ with s as the initial state, and $E_{\mathcal{O}}$ is the execution of $\mathcal{O}$ with s as the initial state. The execution $E_{\mathcal{O}}$ is clearly decomposable into three consecutive sub-executions, $E_{\mathcal{O}}^{before}, E_{\mathcal{O}}^{marked}, E_{\mathcal{O}}^{after}$, corresponding to the three consecutively stringed e-hammocks *before*, *marked*, and *after*. The execution $E_{\mathcal{H}}$ of the original e-hammock $\mathcal{H}$ is also decomposable into three sub-executions: for any B (where B is *before*, *marked* or *after*), the B sub-execution $E_{\mathcal{H}}^{B}$ of $E_{\mathcal{H}}$ is simply the projection of $E_{\mathcal{H}}$ restricted to instances of B-nodes.

The three sub-executions of $E_{\mathcal{H}}$ may therefore be interleaved, and may even overlap (because any instance in $E_{\mathcal{H}}$ of a predicate/jump that belongs to multiple e-hammocks in $\mathcal{O}$ will belong to multiple sub-executions).

Two corresponding sub-executions of $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$ (for instance, $E_{\mathcal{H}}^{before}$ and $E_{\mathcal{O}}^{before}$) are said to be *identical* if (a) the two sub-executions consist of equal number of instances, and (b) corresponding instances (position-wise) in the two sub-executions are instances of the same CFG node, and *consume* identical values. Point (b) implies that corresponding instances define the same set of variables, with the same values.

Property *IdentExecs.* An e-hammock H in $\mathcal{O}$ (H is *before*, *marked* or *after*) is said to be *reached* in the execution $E_{\mathcal{O}}$ if control reaches the entry node of this e-hammock in $E_{\mathcal{O}}$ (an e-hammock would not be reached if control reaches a copy of an exiting jump in some preceding e-hammock in $\mathcal{O}$, and that copy has not been converted into a `goto` whose target is within $\mathcal{O}$).

For any e-hammock H that is reached in the execution $E_{\mathcal{O}}$, $E_{\mathcal{O}}^{H}$ (the H sub-execution of $E_{\mathcal{O}}$) is identical to $E_{\mathcal{H}}^{H}$ (the H sub-execution of $E_{\mathcal{H}}$).

Proof of Property *IdentExecs*. The proof is by induction on the position of an e-hammock H in $\mathcal{O}$ (first, second, or third). Actually, we combine the base-case and inductive arguments into a single argument; letting H be any e-hammock in $\mathcal{O}$ that is reached in $E_{\mathcal{O}}$, we prove that $E_{\mathcal{H}}^{H}$ is identical to $E_{\mathcal{O}}^{H}$. The inductive hypothesis is that for each e-hammock B in $\mathcal{O}$ that precedes H, the two sub-executions $E_{\mathcal{H}}^{B}$ and $E_{\mathcal{O}}^{B}$ are identical (B is definitely reached in $E_{\mathcal{O}}$ because H is reached). We call this inductive hypothesis the "outer" inductive hypothesis (to distinguish it from the "inner" induction, introduced below). Within the proof, we make distinctions (wherever needed) between the case where there is no e-hammock in $\mathcal{O}$ that precedes H and the case where such e-hammocks do exist.

We use an "inner" induction to show that $E_{\mathcal{H}}^{H}$ is identical to $E_{\mathcal{O}}^{H}$; this induction is on the length of $E_{\mathcal{H}}^{H}$ (i.e., on the number of instances in this sub-execution). The base-case argument and inductive argument for the inner induction follow.

Base case: Let n be the entry node of $\mathcal{H}$, if that node is an H-node, else let n be the unique H-node in $\mathcal{H}$ that is first reached from the entry node along paths in $\mathcal{H}$. Clearly then, the first instance in $E^H_{\mathcal{H}}$ (i.e., the first instance of an H-node in $E_{\mathcal{H}}$) is an instance of n. By Corollary B.5, n is the entry node of H. Therefore the first instance in $E^H_{\mathcal{O}}$ is also an instance of n.

We now show that both these first instances, which we call $n_{\mathcal{H}}$ and $n_{\mathcal{O}}$ respectively, consume identical values for v, where v is any variable used by n; in other words $n_{\mathcal{H}}$ and $n_{\mathcal{O}}$ consume identical values. Let $d_{\mathcal{H}}$ be the closest instance that precedes $n_{\mathcal{H}}$ in $E_{\mathcal{H}}$ and that defines v (for now we assume that no variable is used in $\mathcal{H}$ without being defined first; this assumption is unrealistic, but we re-address it later in the proof). In other words, the definition in $d_{\mathcal{H}}$ reaches the use in $n_{\mathcal{H}}$. Let d be the node in $\mathcal{H}$ of which $d_{\mathcal{H}}$ is an instance. Since $n_{\mathcal{H}}$ is the first instance in $E_{\mathcal{H}}$ of an H-node, d is not present in H. Applying Property *ConstraintsSat* on $d_{\mathcal{H}}$ and $n_{\mathcal{H}}$, we determine that d is present in some e-hammock D in $\mathcal{O}$ that precedes H (for instance, if H is *after*, D could be *before* or *marked*). By the outer inductive hypothesis:

1. $E^D_{\mathcal{H}}$ is identical to $E^D_{\mathcal{O}}$.

2. let $d_{\mathcal{O}}$ be the instance in $E^D_{\mathcal{O}}$ that corresponds position-wise to $d_{\mathcal{H}}$ in $E^D_{\mathcal{H}}$; $d_{\mathcal{O}}$ is an instance of d.

3. $d_{\mathcal{H}}$ and $d_{\mathcal{O}}$ consume identical values, and therefore assign the same value to variable v.

Our goal now is to show that no instance that defines v intervenes between the instances $d_{\mathcal{O}}$ and $n_{\mathcal{O}}$ in $E_{\mathcal{O}}$; that would imply that $n_{\mathcal{H}}$ and $n_{\mathcal{O}}$ consume the same value of v, and therefore we would be done. This goal is achieved by proving two properties:

1. $d_{\mathcal{O}}$ is the last instance in $E^D_{\mathcal{O}}$ to define v.

2. for each e-hammock B in $\mathcal{O}$ that is in between D and H, $E^B_{\mathcal{O}}$ contains no instance that defines v.

It is sufficient to show these properties, because $n_{\mathcal{O}}$ is the first instance in $E^H_{\mathcal{O}}$. We first consider the first property. For contradiction, assume there is some instance $e_{\mathcal{O}}$ in $E^D_{\mathcal{O}}$ that follows $d_{\mathcal{O}}$ and that defines v. Let e be the node of which $e_{\mathcal{O}}$ is an instance. We already know that $E^D_{\mathcal{H}}$ is identical to $E^D_{\mathcal{O}}$; therefore there is an instance $e_{\mathcal{H}}$ in $E^D_{\mathcal{H}}$ that corresponds position-wise to $e_{\mathcal{O}}$, that comes after $d_{\mathcal{H}}$, that defines v, and that is an instance of e. Because $d_{\mathcal{H}}$ is the closest definition of v to precede $n_{\mathcal{H}}$ in $E_{\mathcal{H}}$, $e_{\mathcal{H}}$ comes after $n_{\mathcal{H}}$ in $E_{\mathcal{H}}$. Applying Property *ConstraintsSat* on $e_{\mathcal{H}}$ and $n_{\mathcal{H}}$, the node e should be present either in H or in some e-hammock in $\mathcal{O}$ that comes after H (because n is in H). However D, which contains e, is present before H in $\mathcal{O}$. Therefore we are done showing the first property.

The second property can be proved in a similar manner. For contradiction assume that for some e-hammock B in $\mathcal{O}$ that is in between D and H, $E^B_{\mathcal{O}}$ contains an instance $f_{\mathcal{O}}$ that defines v; let f be the node of which $f_{\mathcal{O}}$ is an instance. Applying the outer inductive hypothesis on $E^B_{\mathcal{O}}$ and $E^B_{\mathcal{H}}$, we know that there is an instance $f_{\mathcal{H}}$ in $E_{\mathcal{H}}$ that defines v, and that is an instance of the same node f. Again, because $d_{\mathcal{H}}$ is the closest definition of v to precede $n_{\mathcal{H}}$ in $E_{\mathcal{H}}$, either $f_{\mathcal{H}}$ precedes $d_{\mathcal{H}}$ or comes after $n_{\mathcal{H}}$ in $E_{\mathcal{H}}$. Applying Lemma *ConstraintsSat* we infer that f should either be in the e-hammock D, or in some e-hammock that precedes D in $\mathcal{O}$, or in the e-hammock H, or in some e-hammock that comes after H in $\mathcal{O}$. However B, which contains f, is in between D and H. Therefore we are done showing the second property, and hence the entire base case.

Inductive case: The inductive hypothesis of the "inner" induction is that the first i instances in $E^H_{\mathcal{H}}$ are identical to the first i instances in $E^H_{\mathcal{O}}$, where i is some integer that is ≥ 1. In particular, the ith instance in $E^H_{\mathcal{H}}$ and the ith instance in $E^H_{\mathcal{O}}$ are both instances of the same node m, have consumed identical values, and have therefore produced the same result. Therefore control leaves both ith instances out of the same edge e (e is the *true* or *false* edge out of m if m is a predicate, and is the sole edge out of m otherwise). We now have two cases:

Case (*first_reached*(m, e, H) does not exist): In this case clearly $E^H_{\mathcal{H}}$ contains only i instances. By Corollary B.5, edge e out of m in H has its target outside H. Therefore,

$E^H_{\mathcal{O}}$ also has only i instances. The inductive hypothesis is that the first i instances are identical in the two sub-executions; therefore we are done proving Property *IdentExecs*.

Case (*first_reached*(m, e, H) exists): Let $t =$ *first_reached*(m, e, H). Clearly, the $(i+1)st$ instance $t_{\mathcal{H}}$ in $E^H_{\mathcal{H}}$ is an instance of t. By Corollary B.5, the target of the edge e in H is also t. Therefore, the $(i+1)st$ instance $t_{\mathcal{O}}$ in $E^H_{\mathcal{O}}$ is also an instance of t.

It remains to be shown that in the second case above $t_{\mathcal{H}}$ and $t_{\mathcal{O}}$ consume identical values. We prove this for any single variable v that is used by t, using two cases.

Case (none of the first i instances in $E^H_{\mathcal{H}}$ define v): Let $d_{\mathcal{H}}$ be the closest instance that precedes $t_{\mathcal{H}}$ in $E_{\mathcal{H}}$ and that defines v. In other words, the definition in $d_{\mathcal{H}}$ reaches the use in $n_{\mathcal{H}}$. Let d be the node of which $d_{\mathcal{H}}$ is an instance, and let d belong to an e-hammock D in $\mathcal{O}$. The case we are in currently implies $D \neq H$. Applying Property *ConstraintsSat* on the instances $d_{\mathcal{H}}$ and $t_{\mathcal{H}}$, we infer that D precedes H in $\mathcal{O}$. By the outer inductive hypothesis:

1. $E^D_{\mathcal{H}}$ is identical to $E^D_{\mathcal{O}}$.
2. let $d_{\mathcal{O}}$ be the instance in $E^D_{\mathcal{O}}$ that corresponds position-wise to $d_{\mathcal{H}}$ in $E^D_{\mathcal{H}}$; $d_{\mathcal{O}}$ is an instance of d.
3. $d_{\mathcal{H}}$ and $d_{\mathcal{O}}$ consume the same values, and therefore assign the same value to variable v.

Our goal now is to show that no instance that defines v intervenes between the instances $d_{\mathcal{O}}$ and $t_{\mathcal{O}}$ in $E_{\mathcal{O}}$; that would imply that $t_{\mathcal{H}}$ and $t_{\mathcal{O}}$ consume the same value of v, and therefore we would be done. This goal is achieved by proving three properties:

1. $d_{\mathcal{O}}$ is the last instance in $E^D_{\mathcal{O}}$ to define v.
2. for each e-hammock B in $\mathcal{O}$ that is in between D and H, $E^B_{\mathcal{O}}$ contains no instance that defines v.

3. none of the first i instances $E_{\mathcal{O}}^{H}$ define v.

The first two properties are proved exactly as in the base case. The third property follows from the inner inductive hypothesis.

Case (v is defined in the first i instances in $E_{\mathcal{H}}^{H}$): Let $d_{\mathcal{H}}$ be the last instance among the first i instances in $E_{\mathcal{H}}^{H}$ to define v. Let d be the node of which $d_{\mathcal{H}}$ is an instance; clearly d belongs to H. We first show that there are no instances that intervene between $d_{\mathcal{H}}$ and $t_{\mathcal{H}}$ in $E_{\mathcal{H}}$ that define v. For contradiction, assume there was such an instance $e_{\mathcal{H}}$; this instance cannot be an instance of an H-node because in that case it would be among the first i instances in $E_{\mathcal{H}}^{H}$, and $d_{\mathcal{H}}$ would not be the last instance among the the first i instances in $E_{\mathcal{H}}^{H}$ to define v. In other words, the node e of which $e_{\mathcal{H}}$ is an instance does not belong to H. However, applying Property *ConstraintsSat* on the instances $d_{\mathcal{H}}$ and $e_{\mathcal{H}}$ in $E_{\mathcal{H}}$, we infer that e belongs to some e-hammock in $\mathcal{O}$ that follows H (H contains d). Applying Property *ConstraintsSat* on the instances $e_{\mathcal{H}}$ and $t_{\mathcal{H}}$ in $E_{\mathcal{H}}$, we infer that d is in some e-hammock that precedes H (H contains t). We thus have a contradiction. Therefore there is no instance in $E_{\mathcal{H}}$ that intervenes between $d_{\mathcal{H}}$ and $t_{\mathcal{H}}$ and that defines v. In other words, the value of v defined in $d_{\mathcal{H}}$ reaches $t_{\mathcal{H}}$.

By the inner inductive hypothesis, there is an instance $d_{\mathcal{O}}$ in $E_{\mathcal{O}}^{H}$ that corresponds to $d_{\mathcal{H}}$ in $E_{\mathcal{H}}^{H}$, such that $d_{\mathcal{O}}$ defines the same value of v as $d_{\mathcal{H}}$. Also, since $d_{\mathcal{H}}$ is the last among the i instances in $E_{\mathcal{H}}^{H}$ to define v, by the same inductive hypothesis no instance that follows $d_{\mathcal{O}}$ and that precedes $t_{\mathcal{O}}$ in $E_{\mathcal{O}}^{H}$ defines v. In other words the value of v defined in $d_{\mathcal{O}}$ reaches $t_{\mathcal{O}}$. Therefore we have shown that $t_{\mathcal{H}}$ and $t_{\mathcal{O}}$ consume the same value of v.

With that we have finished arguing the inductive case. A note about the assumption in the proof that no variable is used in $\mathcal{H}$ without being defined first: We can enforce this assumption, just for the sake of the proof, by constructing a hammock I that is simply a sequence of assignments $v = v$, where v is any variable that could be used in $\mathcal{H}$ without being defined first. We can then prepend I to both $\mathcal{H}$ and $\mathcal{O}$, thus letting it be the first

hammock in both. The base-case of the outer induction would then be to show that $E^I_{\mathcal{H}}$ is identical to $E^I_{\mathcal{O}}$; this is clearly true because $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$ start from the same initial state s. The rest of the proof remains the same as above. □

Now that we have proved Properties *ConstraintsSat* and *IdentExecs*, we move on to proving Property *DefsReached*.

Property *DefsReached*. Let $d_{\mathcal{H}}$ be an instance in the execution $E_{\mathcal{H}}$ such that $d_{\mathcal{H}}$ defines some variable. Let d be the node in $\mathcal{H}$ of which $d_{\mathcal{H}}$ is an instance, and let D be the e-hammock in $\mathcal{O}$ that contains d (because d defines variables, it belongs to a unique e-hammock in $\mathcal{O}$). D is reached in the execution $E_{\mathcal{O}}$ (i.e., control enters D during the execution $E_{\mathcal{O}}$).

Proof of Property *DefsReached*. The only reason D would not be reached in $E_{\mathcal{O}}$ is that some e-hammock H that precedes D in $\mathcal{O}$ is reached in $E_{\mathcal{O}}$, and within $E^H_{\mathcal{O}}$ an exiting jump j that has not been converted into a `goto` (in Step 6 of the algorithm) is reached. Applying Property *IdentExecs* on H, an instance of j is present in $E^H_{\mathcal{H}}$ (i.e., is present in $E_{\mathcal{H}}$). Since j is an exiting jump, this instance of j is the last instance in $E_{\mathcal{H}}$; in other words, the instance $d_{\mathcal{H}}$ comes before the instance of j in $E_{\mathcal{H}}$. This implies that there is a path in $\mathcal{H}$ from d to j, which in turn implies that there is a constraint $d \rightsquigarrow j$ (see the third rule in Figure 5.5). Recall the partitioning of nodes in $\mathcal{H}$ into the e-hammocks in $\mathcal{O}$ satisfies all constraints (Theorem A.1); therefore, because $d \rightsquigarrow j$ is satisfied, we infer that a copy of j is present either in D or in some e-hammock in $\mathcal{O}$ that follows D. In other words, the copy of j in H is *not* the last copy of j in $\mathcal{O}$; therefore, the copy of j in H would have been converted into a `goto` (whose target is the fall-through exit of H) in Step 6; but this contradicts our starting claim that j in H has not been converted into a `goto`. □

We finally use Properties *ConstraintsSat*, *IdentExecs*, and *DefsReached* to prove Theorem B.1.

Proof of Theorem B.1.

Our goals are to prove that:

- the program states at the conclusions of the two executions $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$ are identical, and

- control flows out of $\mathcal{H}$ and $\mathcal{O}$, at the conclusions of the two executions respectively, to the same node outside $\mathcal{H}$ (in the containing CFG).

We first prove the first statement in the theorem – the final states at the end of executions $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$ are identical. We actually prove that for any single variable v, the value of v is identical at the end of the two executions. There are two broad cases: either v is not defined by any instance in $E_{\mathcal{H}}$, or v is defined in $E_{\mathcal{H}}$. In the first case, we can show that v is not defined in $E_{\mathcal{O}}$ either (i.e., the final value of v at the end of each of the two executions is simply the value of v in the initial state s). We show by this contradiction: assume v is defined $E_{\mathcal{O}}^{H}$, where H an e-hammock in $\mathcal{O}$ (*before*, *marked*, or *after*) that is reached in $E_{\mathcal{O}}$. Applying Property *IdentExecs* $E_{\mathcal{H}}^{H}$ also defines v; however, that contradicts our claim that v is not defined in $E_{\mathcal{H}}$.

We now look at the second case – v is defined by some instance in $E_{\mathcal{H}}$. Let $d_{\mathcal{H}}$ be the last instance in $E_{\mathcal{H}}$ to define v, and let d be the node in $\mathcal{H}$ of which $d_{\mathcal{H}}$ is an instance. Let D be the e-hammock in $\mathcal{O}$ that contains d. Property *DefsReached* says that D is reached in $E_{\mathcal{O}}$, while Property *IdentExecs* says that some instance in $E_{\mathcal{O}}^{D}$ defines v. Property *IdentExecs* also says that the last instance $d_{\mathcal{O}}$ in $E_{\mathcal{O}}^{D}$ to define v is identical to $d_{\mathcal{H}}$; i.e., $d_{\mathcal{O}}$ is an instance of d, it consumes identical values as $d_{\mathcal{H}}$, and it therefore assigns the same value to v as $d_{\mathcal{H}}$. We now complete the proof by showing that no instance in $E_{\mathcal{O}}^{H}$ defines v, where H is any e-hammock in $\mathcal{O}$ that comes after D and that is reached in $E_{\mathcal{O}}$. For contradiction, assume there is such an e-hammock H in $\mathcal{O}$, and assume $e_{\mathcal{O}}$ is an instance in $E_{\mathcal{O}}^{H}$ that defines v. Let e be the node in $\mathcal{H}$ of which $e_{\mathcal{O}}$ is an instance. By Property *IdentExecs* there is an instance $e_{\mathcal{H}}$ in $E_{\mathcal{H}}^{H}$ that is identical to $e_{\mathcal{O}}$. Since $d_{\mathcal{H}}$ is the last instance in $E_{\mathcal{H}}$ to define v, $e_{\mathcal{H}}$ precedes $d_{\mathcal{H}}$ in $E_{\mathcal{H}}$. Since both these instances define v, Property *ConstraintsSat* applies, and states that e is not present in any e-hammock in $\mathcal{O}$ that follows the e-hammock that contains d. However H contains e, and our claim was that it comes after D. We have therefore proved by contradiction that $d_{\mathcal{O}}$ is the last instance in $E_{\mathcal{O}}$ to define v. Therefore the final value of v at the end of the two executions $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$ is equal to the value assigned to v by $d_{\mathcal{H}}$ (and $d_{\mathcal{O}}$).

We now prove the second statement in the theorem – control flows out of $\mathcal{H}$ and $\mathcal{O}$, at the conclusions of the two executions $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$, respectively, to the same outside node (in the containing CFG). There are two broad cases to consider: control flows out of $\mathcal{H}$ to its fall-through exit in $E_{\mathcal{H}}$, or control flows out of $\mathcal{H}$ through an exiting jump j. In the first case, because control does not reach any exiting jump in $E_{\mathcal{H}}$, we infer (using Property *IdentExecs*) that control reaches no exiting jump in $E_{\mathcal{O}}^{H}$, where H is any e-hammock that is reached in $E_{\mathcal{O}}$. In other words, for each e-hammock H in $\mathcal{O}$, control flows out of H to its fall-through exit in $E_{\mathcal{O}}$. That is, control flows out of $\mathcal{O}$ in $E_{\mathcal{O}}$ to the fall-through exit of $\mathcal{O}$. In other words control flows out of $\mathcal{H}$ and $\mathcal{O}$, in the two executions, respectively, to the fall-through exits of the two respective e-hammocks. However, these two fall-through exit nodes are actually the same node in the containing procedure (because the transformation done by the algorithm is to simply replace $\mathcal{H}$ with $\mathcal{O}$ in the containing CFG).

We now go to the other case – control leaves $\mathcal{H}$ in $E_{\mathcal{H}}$ through an exiting jump j (i.e., an instance of j is the last instance in $E_{\mathcal{H}}$, and no other instance in $E_{\mathcal{H}}$ but the last one is an instance of any exiting jump). Let L be the last e-hammock in $\mathcal{O}$ to have a copy of j. We first show that L is reached in the execution $E_{\mathcal{O}}$. If L is *before*, or if L is *marked* and *before* is empty, this is trivially true.

Say L is *marked* and *before* is non-empty. We have two cases: either *before* has a copy of j, or it does not. Consider the first case. Since an instance of j is the last instance in $E_{\mathcal{H}}$, Property *IdentExecs* tells us that an instance of j is the last instance in $E_{\mathcal{O}}^{before}$. However, because *before* does not contain the last copy of j (L does), the copy of j in *before* is a `goto` whose target is the fall-through exit node of *before* (see Step 6 of the algorithm). Therefore control flows out of *before* in $E_{\mathcal{O}}^{before}$ to the fall-through exit of *before*. The other case is that *before* does not have a copy of j. Since the last instance in $E_{\mathcal{H}}$ is the only instance of an exiting jump in $E_{\mathcal{H}}$, the last instance in $E_{\mathcal{H}}^{before}$ is not an instance of any exiting jump; by Property *IdentExecs*, the same is true for $E_{\mathcal{O}}^{before}$. Therefore control flows out of *before* in $E_{\mathcal{O}}^{before}$ to the fall-through exit of *before*. Therefore, in either case, control reaches the

fall-through exit of *before* (i.e., the entry of *marked*) in $E_{\mathcal{O}}$; i.e., L is reached in $E_{\mathcal{O}}$. (A similar argument can be used to show that L is reached even it is the *after* e-hammock.)

Property *IdentExecs* is now applicable, and according to it $E^L_{\mathcal{H}}$ is identical to $E^L_{\mathcal{O}}$. Since an instance of j is the last instance in $E_{\mathcal{H}}$, and since j is in L, we infer that an instance of j is the last instance in $E^L_{\mathcal{O}}$. That is, control reaches j in L in the execution $E_{\mathcal{O}}$. Since the copy of j in L is the last copy of j in $\mathcal{O}$, this copy of j has not been converted into a `goto` whose target is the fall-through exit of L; i.e., this copy remains in its original form – a `return`, `break`, `continue`, or `goto`. Therefore control leaves $\mathcal{O}$ via the copy of j in L in the execution $E_{\mathcal{O}}$. In other words, control leaves $\mathcal{H}$ and $\mathcal{O}$ via j in the two executions $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$, respectively. It can easily shown that this implies our final result: control flows out of $\mathcal{H}$ and $\mathcal{O}$, at the conclusions of the two executions $E_{\mathcal{H}}$ and $E_{\mathcal{O}}$, respectively, to the same in the containing CFG (the argument proceeds by considering each possibility for the kind of j: `break`, `continue`, `return`, or `goto`). □

Appendix C: The clone-group algorithm is semantics-preserving

Recall that the first step in the clone-group algorithm is to apply the individual-clone algorithm on each individual clone. We have already proven this step to be semantics-preserving. The individual-clone algorithm produces a *marked* hammock corresponding to each clone, and this *marked* hammock is a block sequence. Recall that the approach of the clone-group algorithm (Figure 6.4) is to visit each set of corresponding maximal block sequences (the outermost set, and inner sets at all levels of nesting) individually, and to make that set in-order by permuting one or more of its block sequences (this permutation is done by the procedure in Figure 6.7). Let M be any one of the *marked* hammocks. From the perspective of M, the clone-group algorithm visits all maximal block sequences in M, including M itself and including inner maximal block sequences nested inside M, and permutes these block sequences. Let b_m be M itself, or any one of the maximal block sequences nested somewhere inside M (at any depth). In this section we prove that the permutation done by the algorithm to b_m is semantics-preserving. Clearly it follows from this that the transformation to M, and hence the transformation to each *marked* hammock in the group, is semantics-preserving. That would complete the proof that the clone-group algorithm is semantics preserving.

The result we prove is stated formally as Theorem C.1.

Theorem C.1 Let M be any one of the *marked* hammocks (block sequences) produced by the individual-clone algorithm, when it is invoked as a subroutine by the clone-group algorithm on one of the given clones. Let b_m be any one of the maximal block sequences in M (b_m could be M itself, or could be a maximal block sequence nested inside M at any depth). Note that b_m is not necessarily a hammock (although M is a hammock); i.e., there may be jumps outside b_m whose targets are in b_m, and there may be jumps in b_m whose targets are outside.

Let b'_m be the transformed result produced by the algorithm; i.e., b'_m is a permutation of b_m. The statement of this theorem is that b_m and b'_m are semantically equivalent. That is, considering executions of b_m and b'_m from identical initial program states s, and from the same starting node e (a starting node e is either the entry node of b_m/b'_m, or a node in b_m/b'_m that is the target of an outside jump), we have the following properties:

- the executions of b_m and b'_m terminate with the same final states.
- either control flows out of b_m to its fall-through exit and control flows out of b'_m to its fall-through exit, OR control leaves b_m through a jump j whose target is outside b_m and control leaves b'_m through the same jump j. In other words, both executions terminate with control reaching the same outside node.

Throughout this section we use b_m to refer to a *marked* hammock, or a maximal block sequence nested somewhere inside a *marked* hammock. b'_m is the permutation of b_m produced by the algorithm.

Definition 9 (constituent chains of hammocks) Let $B_1, B_2, \ldots, B_m$ be the sequence of blocks that constitute b_m; i.e., $b_m = [B_1, B_2, \ldots, B_m]$. A sub-sequence $H = [B_j, \ldots, B_k]$ of b_m is said to be a *constituent hammock* of b_m iff there are no jump nodes in H whose targets are outside H and no jump nodes outside H whose targets are in H. (Note that this is stricter than simply saying that H is a hammock, because here we disallow jumps from outside H to come even into the entry node of H, and we disallow jumps in H to go even to the fall-through exit of H.)

If $A_1, A_2, \ldots, A_k$ are consecutive sub-sequences of b_m such that each A_i is a constituent hammock of b_m, then $[A_1, A_2, \ldots, A_k]$ is said to be a *constituent chain of hammocks* of b_m. (Note that a constituent chain of hammocks is itself a constituent hammock.)

A *permutation* of a constituent chain of hammocks $[A_1, A_2, \ldots, A_k]$ has intuitive meaning: it is a permutation of the constituent blocks of the chain, treating each A_i as an atomic unit whose contiguity and internal ordering is preserved.

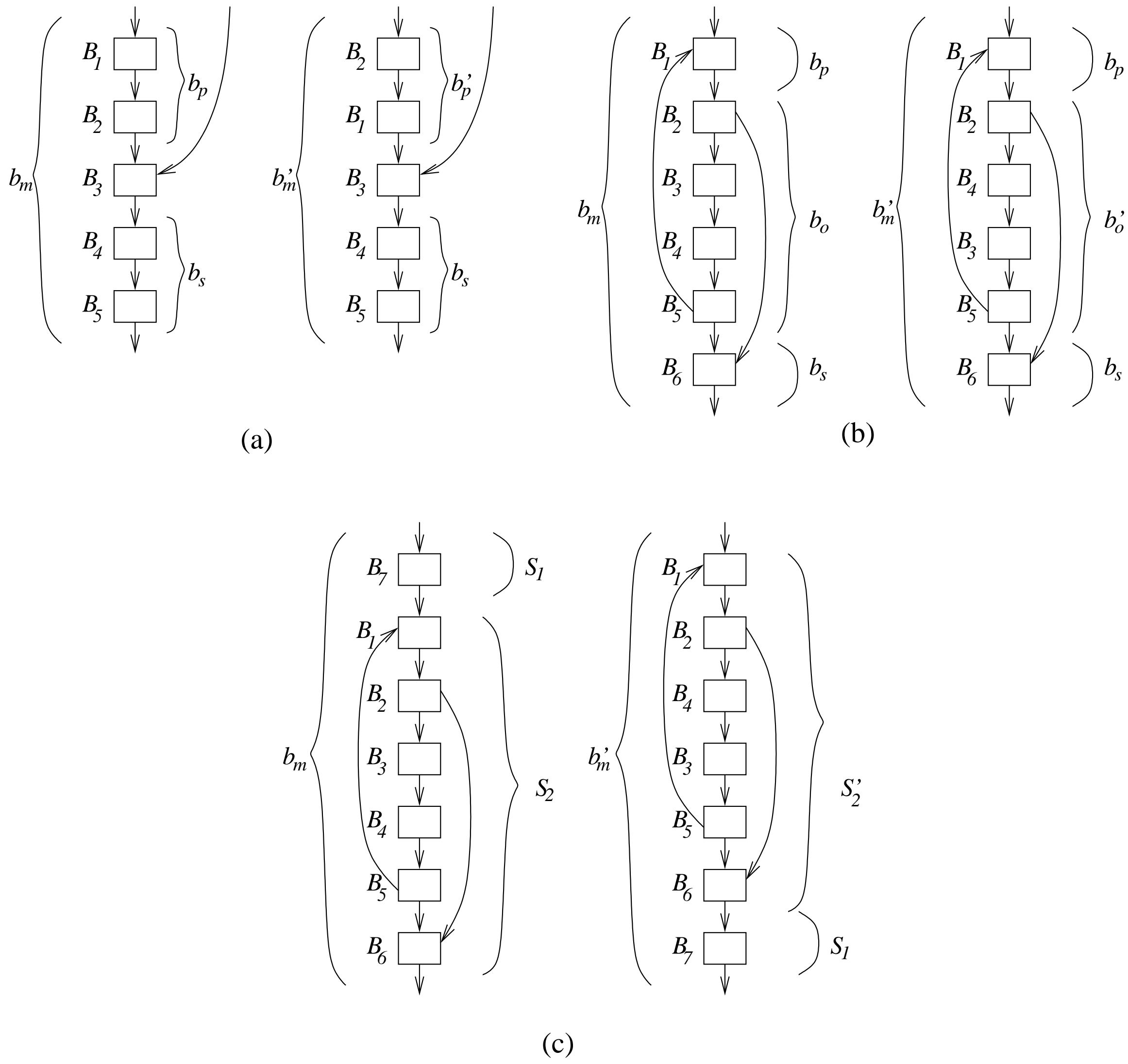


Figure C.1 Chains of hammocks

Example: Figure C.1 contains illustrative examples that we will use throughout this section. Consider the block sequence b_m on the left side of Figure C.1(c). B_3 and B_4 are constituent hammocks of b_m; therefore $[B_3, B_4]$ is a constituent chain of hammocks of b_m. The sub-sequences S_1 and S_2 are two other constituent hammocks of b_m; therefore $[S_1, S_2]$ is another constituent chain of hammocks of b_m.

In Figure C.1(a) $[B_1, B_2]$ is a constituent chain of hammocks of b_m; so is $[B_4, B_5]$. □

Let B be any constituent block of b_m. Due to the presence of jumps that *affect* B, B may *execute* (i.e., control may enter B) zero, one, or more times during any execution of b_m. Intuitively, a jump in b_m affects B if its source is before B and its target is either after B or outside b_m, or if its source is after B and target is before B. A jump outside b_m affects B if its target is in b_m but after B. In general, an arbitrary permutation of b_m changes the set of jumps that affect B; if that happens B may execute different number of times before and after the permutation, which implies that semantics may not be preserved. However, any permutation of any constituent chain of hammocks $C = [A_1, A_2, \ldots, A_k]$ of b_m preserves the effects of all jumps on all constituent blocks of b_m. The reason for this is that whenever control enters C during an execution, control enters each hammock A_i in C exactly once before leaving C; this is true originally, and is true after any arbitrary permutation the A_is.

In the rest of this section we introduce a series of definitions and lemmas, which will finally be used to prove Theorem C.1. First, we state Lemma C.2 (we provide no proof, for the result is fairly obvious).

Lemma C.2 Let S_1 and S_2 be any two sub-sequences of b_m such that the two overlap, and neither is completely contained in the other. Let the first constituent block of S_2 come after the first constituent block of S_1 in b_m. Clearly there are three smaller sub-sequences in b_m: a sub-sequence S_a that is a prefix of S_1 and that does not overlap S_2, a sub-sequence S_b that is the region where S_1 and S_2 overlap, and a sub-sequence S_c that is a suffix of S_2 and that does not overlap S_1. If the constituent blocks of S_1 occur contiguously in b'_m, and if the constituent blocks of S_2 occur contiguously in b'_m, then the constituent blocks of each

smaller sub-sequence (S_a, S_b, and S_c) occur contiguously in b'_m; i.e., b'_m consists of three sub-sequences S'_a, S'_b, and S'_c, in that order, such that S'_a, S'_b, and S'_c are permutations of S_a, S_b, and S_c, respectively.

Example: Consider b_m and b'_m in Figure C.1(c). Sub-sequence $[B_1, \ldots, B_5]$ of b_m overlaps sub-sequence $[B_2, \ldots, B_6]$ of b_m; also the constituent blocks of each of these two sub-sequences occur contiguously in b'_m. Therefore Lemma C.2 applies; the three smaller sub-sequences of b_m are $[B_1]$, $[B_2, \ldots, B_5]$, and $[B_6]$. Notice that these three sub-sequences occur in the same order in b'_m, and that the second sub-sequence is permuted. □

We now provide some definitions. A *forward jump* in b_m is a jump node in a constituent block of b_m whose target is in another constituent block of b_m that follows. Similarly, a *backward jump* in b_m is a jump node in a constituent block of b_m whose target is in some preceding constituent block of b_m. A *jump interval* of b_m is a sub-sequence $[B_i, \ldots, B_j]$ of b_m such that there is either a forward jump from B_i to B_j or a backward jump from B_j to B_i. B_i and B_j are respectively called the *head* and *tail* of the jump interval. Two jump intervals of b_m are said to be *overlapping* if the two intervals share at least one constituent block in common. A set S of jump intervals of b_m is said to be *connected* if every interval in S is related to every other interval in S via the transitive closure of the overlap relationship (note that the overlap relation itself is not transitive). The head of S is the earliest constituent block of b_m that belongs to any interval in S, while the tail of S is the last constituent block of b_m to belong to any interval of S.

Example: In Figure C.1(c) S_2 is a connected set of two overlapping jump intervals, $[B_1, \ldots, B_5]$ and $[B_2, \ldots, B_6]$. □

Lemma C.3 If $[B_j, \ldots, B_m]$ is a jump interval of b_m, then:

- B_j comes before B_m in b'_m, and
- the blocks sub-sequence $[B_j, \ldots, B_m]$ occur contiguously in b'_m.

PROOF OF LEMMA C.3. From the definition of jump interval it follows that there is a forward/backward jump connecting B_j and B_m. The loop "`forall` constituent blocks B_l of $b \ldots$" in Figure 6.6(a) (lines 10-16) therefore applies; the constraints generated there ensure that the properties stated above hold in b'_m. □

Lemma C.4 Let S be any connected set of jump intervals of b_m, and let B_h and B_t respectively be the head and tail of S.

1. Every constituent block of b_m that is in between B_h and B_t belongs to some interval in S; i.e., S is a sub-sequence of b_m.
2. The constituent blocks of S occur contiguously in b'_m; i.e., some permutation of S is a sub-sequence of b'_m.

PROOF OF LEMMA C.4. The first property follows in a straightforward manner from the definition of a connected set of jump intervals. We prove the second property using induction on the number of jump intervals in S.

The base case is when S consists of a single jump interval. In this case Lemma C.3 states that the constituent blocks of this interval occur contiguously in b'_m.

The inductive case is that S has n jump intervals, and that S is connected. Let B_l be the last constituent block of b_m to satisfy the property that it is the head of some interval in S. Let L be any one of the intervals in S whose head is B_l. Because S is connected, and because the head of no interval in S comes after B_l, it follows that $S' = (S - \{L\})$ is a connected set of jump intervals. Applying the inductive hypothesis, we infer that S' is a sub-sequence of b_m, and that the constituent blocks of S' occur contiguously in b'_m. If the sub-sequence L is completely contained within the sub-sequence S', then the sub-sequences S and S' are equal, and we are done proving the lemma. If the sub-sequence S' is completely contained within the sub-sequence L, then the sub-sequence S is equal to the sub-sequence L; Lemma C.3 states that L is contiguous in b'_m, and there we are done. On the other hand, if the none of the above two conditions are true, then Lemma C.2 applies (with S' and L

here substituting for S_1 and S_2 in that lemma's statement). That lemma states that the constituent blocks of b_m that are in present in the union of S' and L occur contiguously in b'_m; in other words the constituent blocks of S occur contiguously in b'_m. □

Let b be any sub-sequence of b_m. We define an *entering jump* of b to be any jump node outside b whose target is in b. Similarly, a *leaving jump* of b is a jump node in b whose target is outside b. A jump interval of b_m / constituent hammock of b_m / constituent chain of hammocks of b_m, when completely contained in b, is said to be a jump interval of b / constituent hammock of b / constituent chain of hammocks of b.

Lemma C.5 Let b be any sub-sequence of b_m such that the constituent blocks of b occur contiguously in b'_m; i.e., some permutation b' of b is a sub-sequence of b'_m (b_m and b'_m were introduced earlier in this section). There exists a set S of constituent chains of hammocks of b such that the only differences between b and b' are that chains in S are permuted in b'.

Proof of Lemma C.5. The proof is by induction on the length of the permuted block sequence b. The base case is when b consists of one block only; the Lemma holds trivially in this case because there there are no permutations of a sequence of length one.

In the inductive case, let b consist of n blocks, where $n > 1$. The inductive hypothesis is that for any proper sub-sequence c of b, if some permutation c' of c is a sub-sequence of b', then c and c' differ only in that some set of constituent chains of hammocks of c are permuted in c'. Our argument is based on the structure of b, and we have three cases.

Case 1 (b has an entering jump or a leaving jump): Let B_j be any constituent block of b that is the target of an entering jump of b, or that contains a leaving jump of b. Call this jump j. Let b_p be the prefix of b up to but not including B_j, and let b_s be the suffix of b starting at the constituent block that immediately follows B_j.

We now prove that all constituent blocks of b_p precede B_j in b', and that all constituent blocks of b_s come after B_j in b'. If j is also an entering/leaving jump of the full sequence b_m, then the constraints generated in the first `if` statement (lines 3-7) in Figure 6.6(a) clearly ensure this. On the other hand, say j is not an entering/leaving jump of b_m. In that case

j is a forward/backward jump of b_m such that B_j is the head or tail of the jump interval caused by this jump, and such that the other end point of this interval is outside b. Say B_j is the head of this interval (the argument for the case when B_j is the tail is similar). If B_j is the first constituent block of b, then according to Lemma C.3 every subsequent block of b follows B_j in b'. On the other hand, if this is not the case, then note that b and the jump interval caused by j are two overlapping sub-sequences of b_m that each remain contiguous in b'_m (this is true for b according to this lemma's statement, and is true for the jump interval according to Lemma C.3). Therefore Lemma C.2 applies (with b_p here substituting for S_a there, and $[B_j] + b_s$ substituting for S_b there). This lemma, together with Lemma C.3, tells us that the constituent blocks of b_p precede B_j in b', and that the constituent blocks of b_s come after B_j in b'.

We have shown so far that b' is equal to some permutation b'_p of b_p, followed by B_j, followed by some permutation b'_s of b_s. One of b_p or b_s may be empty, but clearly both are of length less than n. Therefore the inductive hypothesis applies, which means b_p differs from b'_p only in that some constituent chains of hammocks of b_p are permuted in b'_p (the same is also true about b_s and b'_s). Therefore we can infer that b' differs from b only in that some set of constituent chains of hammocks of b are permuted in b' (with each such chain being completely inside b_p or b_s).

Example: Figure C.1(a) illustrates this case. Assume b is equal to the entire sequence b_m. B_3 in this example corresponds to B_j in the argument above, $[B_1, B_2]$ corresponds to b_p, and $[B_4, B_5]$ corresponds to b_s. Notice that b_m and b'_m differ in that the constituent chain of hammocks $[B_1, B_2]$ is permuted in b'. □

Case 2 (every constituent block of b belongs to some jump interval of b, and the set of all jump intervals of b is connected): Let S be some minimal connected set of jump intervals of b such that the head of S is the first constituent block of b and the tail of S is the last constituent block of b (the case we are in guarantees that such an S exists). Let B_l be the last constituent block of b to satisfy the property that it is the head of some interval in S. Let L be any one of the intervals in S whose head is B_l. Because S is connected, and because

the head of no interval in S comes after B_l, it follows that $S_1 = (S - \{L\})$ is a connected set of jump intervals. Clearly, because the sub-sequence S is equal to b (see the definition above), the sub-sequence S_1 is a sub-sequence of b. Therefore, applying Lemma C.4, we infer that the constituent blocks of S_1 occur contiguously in b'. We call this inference I1.

The constituent blocks of L are contained in b and occur contiguously in b (because L is a jump interval of b). Lemma C.3 therefore tells us that constituent blocks of L occur contiguously in b'. We call this inference I2.

We now make our third inference, I3: (a) the sub-sequences S_1 and L of b overlap, (b) the head of S_1 is the first constituent block of b, (c) the tail of S_1 is *not* the last constituent block of b, (d) the head of L is *not* the first constituent block of b, and (e) the tail of L is the last constituent block of b. Recall that the set of blocks S is the union of the set of blocks S_1 and the set of blocks L. Property (a) holds because otherwise the set S would not be connected. Property (b) holds because otherwise the first constituent block of b would not belong to S (recall that the head of L does not come before the head of any other jump interval in S). If the tail of S_1 is the last constituent block of b, then together with (b), we would infer that the sub-sequence S_1 contains the sub-sequence L, which would make S non-minimal. Therefore, Property (c) holds. This in turn implies Property (e) (because the last constituent block of b definitely belongs to S). This in turn implies Property (d) (otherwise the sub-sequence L would contain the sub-sequence S_1, which would make S non-maximal). Therefore we are done showing I3.

From I3 it follows that b can be partitioned into three consecutive smaller sub-sequences: b_p, which is the prefix of the sub-sequence S_1 that does not overlap with L, then b_o, which is the overlapping portion of the two sub-sequences S_1 and L, and finally b_s, which is the suffix of the sub-sequence L that does not overlap with S_1. Each of these three smaller sub-sequences is non-empty, and therefore each one is of length less than n.

Inferences I1-I3 allow us to apply Lemma C.2; S_1 here corresponds to S_1 in that lemma's statement, and L here corresponds to S_2 in that lemma' statement. That lemma tells us that b' is equal to some permutation of b_p followed by some permutation of b_o followed by

some permutation of b_s. Because b_p is of length less than n, the inductive hypothesis applies, which means b_p differs from b'_p only in that some constituent chains of hammocks of b_p are permuted in b'_p (the same is also true about b_o and b'_o, and b_s and b'_s). Therefore we infer that b' differs from b only in that some set of constituent chains of hammocks of b are permuted in b' (with each such chain being completely inside one of the three smaller sub-sequences of b).

Example: Figure C.1(b) illustrates this case. Assume b is equal to the entire sequence b_m. Notice that every constituent block of b_m belongs to one of the two jump intervals, and that the two intervals are connected. Notice also that the difference between b_m and b'_m is that the constituent chain of hammocks $[B_3, B_4]$ of b_m, which is inside b_o, is permuted in b'_m. □

Case 3 (the default case): In this case the set of all jump intervals of b is either not connected, or does not include every constituent block of b. Our strategy now is to partition b into a series of sub-sequences $S_1, S_2, \ldots, S_k$. Each sub-sequence S_i either consists of a single constituent block of b (if that block belongs to no jump interval), or consists of those blocks that belong to some maximal connected set of jump intervals of b. Each S_i is contained in b. k is greater than one, because otherwise every constituent block of b belongs to the same maximal connected set of jump intervals, which is the previous case. The constituent blocks in each S_i occur contiguously in b'; this is trivially true if S_i consists of a single block, and is true in the other case also according to Lemma C.4. In other words, b' differs from b in that each S_i in b is permuted in b', and the outer sequence $S_1, S_2, \ldots, S_k$ is permuted in b'. Since k is greater than one each S_i has less than n blocks; therefore the permutation of each S_i corresponds to the permutation of some set of constituent chains of hammocks inside S_i (by the inductive hypothesis).

We complete the proof by showing that each S_i is itself a constituent hammock of b, which implies that the permutation of the outer sequence $S_1, S_2, \ldots, S_k$ is also a permutation of a constituent chain of hammocks of b. Let us call the source and target of a jump the two *ends* of the jump. Since b has no entering or leaving jumps (by the case we are in now), we

only to have to show that there are no forward/backward jumps in b whose one end is in S_i but whose other end is outside S_i, for all i between 1 and k. For contradiction assume this is not true; i.e., assume there is a forward/backward jump whose one end is in S_i, for some i, but whose other end is in a constituent block B_j of b that is not inside S_i. In other words, there is a jump interval I involving B_j and a constituent block of S_i. Therefore, since S_i is a connected set of jump intervals, the intervals in S_i unioned with $\{I\}$ is also a connected set of jump intervals. But that implies that S_i is *not* a maximal connected set of jump intervals of b, which is a contradiction. □

Example: Figure C.1(c) illustrates this case. Assume b is the entire sequence b_m. Notice that b_m consists of two sub-sequences: S_1, which is a single block, and S_2, which is a maximal set of connected jump intervals. Notice that both S_1 and S_2 are constituent hammocks of b_m. Also notice that b_m and b'_m differ as follows: (i) the constituent chain of hammocks $[B_3, B_4]$ of b_m, which is inside S_2, is permuted in b'_m, and (ii) the constituent chain of hammocks $[S_1, S_2]$ of b_m is permuted in b'_m. □

Lemma C.6 Let C be a constituent chain of hammocks of b_m, and let C' be a constituent chain of hammocks of b'_m such that C' is a permutation of C (b_m and b'_m were introduced earlier in this section). C and C' are semantically equivalent; i.e., if C and C' are executed, respectively, with identical initial program states s, then the final program states when control leaves C and C', respectively, are identical.

(Note: Since C and C' are hammocks, each one has a unique entry node, and a unique node outside to which control flows. Therefore, semantic equivalence of C and C' can be defined just in terms of the program states.)

Before we provide the proof for this lemma, we state a sublemma.

Sublemma 1 Let H be any one of the hammocks of the chain C; therefore, H is present in C' also. H is said to have an *upwards exposed* use of a variable v if there is a path in H from its entry to a node that uses v, and there is no node on this path that defines v. Consider the execution of H within the execution of C (from starting state s). Let I be the vector of

values for variables that have upwards-exposed uses in H, at the point of time when control enters H during the execution of C. Provided control enters H during the execution of C' with the same vector of values I for variables that have upwards-exposed uses in H, we have:

- any variable v is defined inside H during the execution of C iff v is defined inside H during the execution of C' (as in Appendix B, whenever we say that a variable is defined in an execution, we mean that the variable is *actually defined* by some instance in that execution).

- the final value assigned to v inside H (which we simply call "the value assigned to v by H") during the execution of C is equal to the final value assigned to v by H during the execution of C'.

We omit a detailed proof for Sublemma 1; the result of this sublemma is intuitive because H in C is identical to H in C', and because the execution behavior of a hammock depends only on the initial values of variables that have upwards-exposed uses in it.

Proof of Lemma C.6. In the initial part of the proof we show that for each hammock H in C and for each variable v, either v is not defined in either execution of H (within the execution of C and within the execution of C'), or the same value is assigned to v by H in both executions. In the end we argue that this implies the result of Lemma C.6.

An observation we make is that since there are no jumps from one hammock of C to another (they would not be constituent hammocks of C if such jumps existed), and no jumps whose source/target (but not both) are in C (again, for the same reason), the execution of C consists of a single execution of each hammock of C, in their order of presence within C; also, the same is true for C'.

Our proof is by induction on the position of hammock H within C. Therefore, the base case is regarding H_1 – the first hammock of C. Let v be any variable that has an upwards-exposed use in H_1, and let H_k be any subsequent hammock of C that defines v. The node in H_k that defines v is clearly anti dependent on the node in H_1 that uses v, via a path that is contained in C; therefore a constraint $B_1 < B_k$ is generated in Step 2 of the algorithm

(Figure 6.7), where B_1 is the constituent block of H_1 that contains the use, and B_k is the constituent block of H_k that contains the definition. This constraint ensures that B_k comes after B_1 in b'_m (Steps 5 and 6 in the algorithm always create permutations that respect all constraints). Now, because C' is a permutation of C with each hammock treated as an individual unit, we infer that H_k comes after H_1 in C'. In other words, we have shown that no hammock that follows H_1 in C and that defines variables that have upwards-exposed uses in H_1 comes before H_1 in C'. Therefore the program states at the start of the two executions of H_1 (within the executions of C and C') are equal to each other (and to s) wrt variables that have upwards-exposed uses in H_1.

We now go onto the inductive case. Consider the nth hammock H_n of C. The inductive hypothesis is that for each of the previous hammocks H_i in C, where $i < n$, the execution of H_i within the execution of C and and the execution of H_i within the execution of C' assigned identical values to variables. Let v be any variable that has an upwards-exposed use in H_n. We now show that the value of v when the execution of H_n begins within the execution of C is identical to the value of v when the execution of H_n begins within the execution of C'. We have two cases to consider.

The first case is that a value was assigned to v in the execution of C before control reached H_n. Let H_d be the last hammock of C that assigns to v before control enters H_n. Therefore, there is a path in C from a node d in H_d to a node n in H_n such that d defines v and n uses v; therefore, either n is flow dependent on d, or n is flow dependent on some other node u that is output dependent on d. Therefore, there exists a (directly generated or transitive) constraint $B_d < B_n$ (see Step 2 in Figure 6.7) where B_d is the constituent block of H_d that contains d and B_n is the constituent block of H_n that contains n. Therefore, following the argument presented earlier, we infer that H_d comes before H_n in C'. Let the value assigned to v by H_d in the execution of C be v'. Clearly, this is the value of v when H_n begins executing within the execution of C. Because $d < n$, the inductive hypothesis applies, and tells us that the value assigned to v by H_d in the execution of C' is also v'. Our goal now is to prove that no hammock between H_d and H_n in C' assigns to v in the

execution of C'. For contradiction assume some hammock H_k that is in between H_d and H_n assigns to v during the execution of C'.

H_k cannot be after H_n in C; for if that were the case then the node in H_k that defines v would be anti-dependent on the node in H_n that uses v (i.e., n), which means that the constraint $B_n < B_k$ would have been generated, where B_n is the constituent block of H_n that contains n and B_k is the constituent block of H_k that contains the definition of v. This implies that H_k could not have come before H_n in C'. Therefore H_k precedes H_n in C; therefore the inductive hypothesis applies, which tells us that H_k assigns a value to v in the execution of C also (our claim was that it assigns a value to v in the execution of C'). Because H_d is the last hammock before H_n to assign a value to v in the execution of C, H_k has to precede H_d in C; however, in that case, the node in H_d that defines v is output dependent (in C) on the node in H_k that defines v, which means that a constraint $B_k < B_d$ would have been generated by the algorithm, where B_k is the constituent block of H_k that contains the definition of v and B_d is the constituent block of H_d that contains the definition of v. However, this constraint is not respected in C', which is not possible (the output of the algorithm respects all constraints). Therefore we have shown that H_d is the last hammock that assigns to v before control enters H_n in the execution of C'. Therefore, in both executions, the value of v when control enters H_n is v'.

The remaining case is that no hammock of C that precedes H_n assigns to v before control enters H_n in the execution of C. In this case, using an argument that is a subset of the previous case's argument, we can show that the value of v when control enters H_n both executions is equal to the value of v in the initial state s.

We are done showing that the values of all variables that have upwards-exposed uses in H_n are identical at the time execution of H_n begins, within the executions of C and C'. Therefore Sublemma 1 applies, and tells us that H_n assigns identical values to variables in both executions. The inductive proof is now complete.

We now show that for any variable v the value of v is the same at the end of the executions of C and C'. Say no hammock of C assigned to v during the execution of C; then by the

result proved above, the same is true in the execution of C'. Therefore the value of v, in both cases, is simply the initial value of v (the value of v in the state s), and we are done proving Lemma C.6.

On the other hand, say H_v is the last hammock of C to assign to v in the execution of C. By the inductive proof above, H_v assigns to v in the execution of C' also, and moreover in both cases H_v assigns the same value v' to v. We now need to show that no hammock that comes after H_v in C' assigns to v. For contradiction assume there is such a hammock H_k. By the inductive result proved above, H_k assigns to v in the execution of C also. Since H_v is the last hammock of C to assign to v, H_k comes before H_v in C. In that case the node in H_v that defines v is output dependent on the node in H_k that defines v, which means a constraint $B_k < B_v$ is generated by the algorithm, where B_k is the constituent block of H_k that contains the definition of v and B_v is the constituent block of H_v that contains the definition of v. This constraint is violated in C', which is not possible. Therefore we have shown that the final value of v after both executions is v'. □

Proof of Theorem C.1. Rather than providing a detailed proof we provide some intuition on how Lemmas C.5 and C.6 imply Theorem C.1. Recall that according to Lemma C.5 the only difference between b_m and b'_m is that some set S of constituent chains of hammocks of b_m are permuted in b'_m. Moreover, for each chain C in S, C is semantically equivalent to its permutation C' in b'_m (Lemma C.6), and both C and C' are single-entry single-outside-exit regions. In other words, assuming control enters C and C' with identical states, control leaves C and C' to reach the same outside node, with identical states. If no two chains of S overlap, this is sufficient to guarantee that executions of b_m and b'_m from identical initial program states s, and from the same starting node e, finish with identical program states and finish at the same node (which is outside b_m/b'_m).

Things are not as obvious if chains in S overlap; if two chains C_1 and C_2 in S overlap, ambiguity exists over whether b'_m is obtainable from b_m by permuting either of these chains first, or whether one of these chains needs to be permuted first. However, note that in the proof of Lemma C.5 only Case 3 allows for overlapping chains. In this case we note that one

of the permuted chains is the outer chain $S_1, S_2, \ldots, S_k$, whereas each other permuted chain is completely inside one of the hammocks S_i of the outer chain. In other words, whenever two chains in S overlap, one of the two chains is completely contained in a single hammock of the other chain. In other words, the order in which chains in b_m are permuted does not matter – any order gives the same result (i.e., b'_m). □

Example: As we noted earlier, in the example of Figure C.1(c), b'_m differs from b_m in that the following two constituent chains of hammocks of b_m are permuted in b'_m: $[B_3, B_4]$, and $[S_1, S_2]$. These two chains overlap; however the first chain is completely contained inside the one of the hammocks (S_2) of the second chain. □

LIST OF REFERENCES

[ACCP98] Guido Araújo, Paulo Centoducatte, Mario Côrtes, and Ricardo Pannain. Code compression based on operand factorization. In *Proc. Int'l Symp. on Microarchitecture*, pages 194–201, December 1998.

[AM72] E. Ashcroft and Z. Manna. The translation of 'go to' programs to 'while' programs. In *Proc. Inf. Processing '71*, pages 250–255. North-Holland Publishing Company, 1972.

[And94] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[Bak95] B. Baker. On finding duplication and near-duplication in large software systems. In *Proc. IEEE Working Conf. on Reverse Eng.*, pages 86–95, July 1995.

[Bak97] B. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Jrnl. on Computing*, 26(5):1343–1362, October 1997.

[Ban93] U. Banerjee. *Loop transformations for restructuring compilers: the foundations*. Kluwer Academic, Boston, 1993. 305 p.

[BB76] H. Barrow and R. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4(4):83–84, January 1976.

[BG98] R. Bowdidge and W. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Trans. on Software Eng. and Methodology*, 7(2):109–157, April 1998.

[BH93] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. volume 749 of *Lect. Notes in Comput. Sci.*, pages 206–222. Springer-Verlag, 1993.

[BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BMD+99] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proc. IEEE Working Conf. on Reverse Eng.*, pages 326–336, 1999.

[BSJL92] D. Bayada, R. Simpson, A. Johnson, and C. Laurenco. An algorithm for the multiple common subgraph problem. *Jrnl. of Chemical Information and Computer Sciences*, 32(6):680–685, Nov.–Dec. 1992.

[BYM+98] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Int. Conf. on Software Maintenance*, pages 368–378, 1998.

[CF94] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[CLG03] W.-K. Chen, B. Li, and R. Gupta. Code compaction of matching single-entry multiple-exit regions. In *10th Int. Symposium on Static Analysis (SAS 2003)*, volume 2694 of *LNCS*, pages 401–417. Springer-Verlag, June 2003.

[CM99] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. ACM Conf. on Prog. Lang. Design and Impl.*, pages 139–149, May 1999.

[CSCM00] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[Csu] http://www.codesurfer.com.

[DBF+95] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *Int. Jrnl. of Applied Software Technology*, 1(3-4):219–36, 1995.

[DEMD00] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[FMW84] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GN93] W. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. on Software Eng. and Methodology*, 2(3):228–269, July 1993.

[KDM+96] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Eng.*, 3(1–2):77–108, 1996.

[KH00] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proc. ACM Symp. on Principles of Prog. Langs.*, pages 155–169, January 2000.

[KH01] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. Int. Symp. on Static Analysis*, pages 40–56, July 2001.

[KH02] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Proc. Fundamental Approaches to Softw. Eng.*, volume 2306 of *Lect. Notes in Comput. Sci.*, pages 96–112. Springer-Verlag, April 2002.

[KH03] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th Int. Workshop on Program Comprehension (IWPC 2003)*, pages 33–42. IEEE Computer Society, May 2003.

[KKP+81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. ACM Symp. on Principles of Prog. Langs.*, pages 207–218, January 1981.

[KL99] Krishna Kunchithapadam and James R. Larus. Using lightweight procedures to improve instruction cache performance. Technical Report CS-TR-1999-1390, University of Wisconsin-Madison, 1999.

[Kri01] J. Krinke. Identifying similar code with program dependence graphs. In *8th Working Conference on Reverse Engineering (WCRE '01)*, pages 301–309. IEEE Computer Society, 2001.

[LD98] A. Lakhotia and J. Deprez. Restructuring programs by tucking statements into functions. *Inf. and Software Technology*, 40(11–12):677–689, November 1998.

[LDK95] S. Liao, S. Devadas, and K Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proc. Chapel Hill Conf. on Advanced Research in VLSI*, 1995.

[LPM+97] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Int. Conf. on Software Maintenance*, pages 314–321, 1997.

[LS80] B. Lientz and E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Reading, Mass., 1980.

[LS86] S. Letovski and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 198–204, May 1986.

[Mar80] B. Marks. Compilation to compact code. *IBM Journal of Research and Development*, 24(6):684–691, November 1980.

[McG82] J. McGregor. Backtrack search algorithms and maximal common subgraph problem. *Software – Practice and Experience*, 12:23–34, 1982.

[MLM96] 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 Int. Conf. on Software Maintenance*, pages 244–254, 1996.

[OO84] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 177–184, 1984.

[Oul82] G. Oulsnam. Unravelling unstructured programs. *The Computer Journal*, 25(3):379–387, August 1982.

[PP94] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[RSW96] S. Rugaber, K. Stirewalt, and L. Wills. Understanding interleaved code. *Automated Software Eng.*, 3(1–2):47–76, June 1996.

[Run00] J. Runeson. Code compression through procedural abstraction before register allocation. Master's thesis, Computing Science Department, Uppsala University, Sweden, March 2000.

[RW90] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.

[RY89] T. Reps and W. Yang. The semantics of program slicing and program integration. In *Proc. Colloquium on Current Issues in Programming Languages*, volume 352 of *LNCS*, pages 360–374. Springer-Verlag, 1989.

[SBB02] B. De Sutter, B. De Bus, and K. De Bosschere. Sifting out the mud: low level C++ code reuse. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 275–291, November 2002.

[SMC74] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Jrnl.*, 13(2):115–139, 1974.

[Vah95] F. Vahid. Procedure exlining: A transformation for improved system and behavioral synthesis. In *International Symposium on System Synthesis*, pages 84–89, September 1995.

[Wei84] M. Weiser. Program slicing. *IEEE Trans. on Software Eng.*, SE-10(4):352–357, July 1984.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10), October 1992.

[WM99] V. Waddle and A. Malhotra. An E log E line crossing algorithm for leveled graphs. volume 1731 of *Lect. Notes in Comput. Sci.*, pages 59–71. Springer-Verlag, 1999.

[WZ97] T. Wang and J. Zhou. Emcss: A new method for maximal common substructure search. *Jrnl. of Chemical Information and Computer Sciences*, 37(5):828–834, Sept.–Oct. 1997.

[Yan91] W. Yang. Identifying syntactic differences between two programs. *Software – Practice and Experience*, 21(7):739–755, July 1991.

[Zas95] M. Zastre. Compacting object code via parameterized procedural abstraction. Master's thesis, Department of Computer Science, University of Victoria, British Columbia, 1995.