Automated Bug Isolation via Program Chipping

by

CHAD DAVID STERLING

B.S. (University of California, Davis) 2003

THESIS
Submitted in partial satisfaction of the requirements for the degree of
MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Professor Ronald A. Olsson (Chair)

Professor Daniel M. Gusfield

Professor Zhendong Su

Committee in Charge

2005

To Mom,
consider this opportunity grabbed.

Contents

List of Figures

List of Tables

Acknowledgments

The research included in this thesis could not have been performed if not for the assistance, patience, love and support of many individuals.

I would like to extend my gratitude first and foremost to God. Without His constant blessings and love, I never would have made it. The goodness that you see or hear from me is truly from the Creator. I thank him because I am not the architect of that, only the recipient.

I would like to thank my advisor Ron Olsson for instructing and mentoring me over the course of my undergraduate and graduate studies. His insight lead to the original proposal to attempt to isolate bugs using what we now refer to as *chipping*. He has helped me through extremely difficult times over the course of the development of ChipperJ and its many many refinements, analysis of the results, and the writing of the thesis. His guidance has been a lamp unto my feet and a light unto my path. I sincerely thank him for his time and his confidence in me.

I would like to thank Kim Reinking for all she did for the Computer Science Department. Her counseling helped me to make the right choice to stay in Davis to complete this degree.

I would like to thank the members of the Black Graduate And Professional Association (BGPA) and of the Black Engineers' Association (BEA) for all of the encouragement and support, and for providing an outlet for me in my times of frustration. Thank you for giving me the chance to grow and develop as a leader. Thank you for sharing your lives with me. You have all inspired me.

I would like to thank the MESA Engineering Program (MEP), for providing the support, advising, and the space and opportunity to grow and develop over the years.

Despite the changes that the program has gone through over the years (both good and bad), I appreciate what I have gained via MEP and the support and praise that the staff has given me. Thanks to the students, many of whom are my best of friends, and all of whom are a constant source of motivation for me.

I would like to thank the National Consortium for Graduate Degrees for Minorities in Engineering and Science, Inc. (GEM) for the fellowship that supported me through the duration of my graduate education. Without that financial commitment I would not have been able to make the contribution that I have. Specifically, I would like to thank Forestine Blake - Manager of Fellowships & Awards for GEM and Sandy Brooks my first intern coordinator at HP who found me great teams to work with and who encouraged me to apply for GEM in the first place.

I would like to thank Garfield for being a loyal, stable, and bug free computer twenty-four hours a day, seven days a week. I was right to call him one sweet, fat cat. I hope his next owner treats him as well as I have.

I would also like to thank Trung Pham for participating in a few of the early discussions of the development of ChipperJ.

I would like to thank my family for their constant words of encouragement and praise even when they did not understand what I was doing. They have been my rock and foundation throughout my education and I love them all very much.

Thank you mom for everything that you have done. There is no way that I can ever express my gratitude for all that you have done, for all of your strength, and for all of your love. Above anyone else you are my hero, and I am so lucky to be your son. Dedicating this thesis to you is just a small token of my appreciation.

Finally I would like to extend my deepest gratitude to my ancestors. Thank you for surviving and making it possible for me to be here today to continue on the valiant struggle for true equality and true liberation via education. Thank you for watching over me and protecting me. Thank you for setting the example of excellence during your earthly existence that I now try to emulate in my earthly existence.

My committment to the community will be the rent I pay for the space I've occupied.

Chapter 1

Introduction

The process of identifying the source of a bug in a program and then fixing that bug is known as debugging. A problem or set of problems can generally be identified by bad or undesired actions from the program (such as bad or wrong output) or in more severe cases a bad program state such as a segmentation fault or a null pointer exception. We will refer to this identifiable sign of a problem as a *symptom*.

The process of debugging is very labor intensive. In most cases, the complexity of the bug in the code has no relationship to the amount of time needed to find and correct that problem. In general, having an experienced programmer on the debugging team will expedite the debugging process. However, program-specific problems can slow and even stump even the most experienced programmers, especially if they are unfamiliar with the code that is being debugged. Probably the best way to expedite this process is to have the person or persons who wrote the code and who are the most familiar with it debug it. In our experience, familiarity with code is often directly correlated with the speed of the debugging process. Unfortunately, as the code volume grows, the likelihood of a single person being the sole creator of the program decreases. Adding more people to a program project, though good for efficiency, workload balancing, meeting deadlines, etc., also tends to create more opportunities for bugs due to issues such as differences in programming style, component interaction, miscommunication about program specifications, and the abilities (or lack thereof) of the individual programmers. Also, the older the code is, the less likely that the person or persons who originally developed it will be available to help debug it, especially as the code moves into *legacy* status.

Thus, software developers spend an enormous amount of their time in debugging and testing their products. Unfortunately, even with the most extensive and rigorous search for bugs during the initial development and testing portions of the software life cycle, the release of the program to the intended user base will typically uncover more bugs that need to be fixed. When a user discovers a bug in a program, the user typically sends the entire program to the developer (or, more often, to the program maintainer), who then needs to isolate the cause of the bug. For example, a compiler or library developer might be presented with an entire user program and just told that something does not work properly. These user programs are usually quite large and the developer can spend much time trying to isolate the bug. Some of that time is wasted, in that the developer needs to understand some of the program's logic to see what the program is doing and exactly what code is being executed.

Traditional program slicing and debugging techniques and tools can assist the developer to some extent, but such tools might not be available for the source language and they can be costly to build or use. Moreover, traditional techniques and tools require some manual intervention. For example, suppose that the bug is that the source program loops infinitely. Typically, the developer would use a debugger to find where the program is looping and then slice the source program on variables involved in that loop to reduce the size of the program.

This example is typical of a common general process: Given a buggy program and some symptom, the developer often reduces the program to a smaller one that exhibits the same symptom. From that smaller program (much smaller, one hopes), the developer can more quickly determine the cause of the bug. Unfortunately, if appropriate tools are not available, this process of reducing the program is done by hand, requires some understanding of the structure of the program, and can be tedious. For example, the developer sometimes employs ad-hoc techniques to track down what parts of the program contribute to the bug. One such technique is *binary search*: repeatedly eliminate half (or some portion) of the program and see whether the program still exhibits the symptom.

Our work aims to remedy this problem by developing techniques and tools to automate this program-reduction process. The developer can specify a symptom and our tool will *automatically* try various heuristics (including binary search mentioned earlier) to reduce the program. We call our approach *program chipping* and our tools *chippers*.

To gain experience with our ideas, we have developed a program chipper for Java, called ChipperJ, and have run it on a variety of small to large programs, including a Java Compiler, looking for various symptoms. The results are promising. The reduced program is generally about 20-35% of the size of the original. ChipperJ takes less than an hour on large programs to perform this reduction; although even if it took overnight, that would be reasonable if it saved the developer time.

Program chipping is a specific application of the general notion of *data slicing* [9]. In our case, programs are the data. Program chipping is in some ways similar to delta debugging [37] (a form of data slicing), but exploits application-specific knowledge, in our case, the syntactic structure of the program. This important difference together with our chipping techniques allows us to produce automatically a reasonable number of reduced programs in a reasonable amount of time. Program chipping is also related to *program slicing* [32]. However, with program slicing, the user looks for behavior with respect to a variable or a group of variables, whereas in program chipping the user looks for behavior with respect to the overall program behavior. In this regard, program chipping treats the program as a "black box" and proceeds automatically to isolate the bug. Notably, program chipping techniques are "light-weight" in the sense they use fairly simple techniques and do not require more sophisticated program analysis techniques, such as slicing.

The rest of this thesis is organized as follows. Chapter 2 presents a general overview of our approach and introduces terminology. Chapter 3 describes ChipperJ, our program chipper for Java. Chapter 4 discusses the effectiveness of ChipperJ over a range of small and large buggy programs with various symptoms. Chapter 5 outlines some of our design decisions and some limitations of our approach. Chapter 6 presents related work and includes a brief comparison of chipping with delta debugging. Finally, Chapter 7 describes future work and concludes the thesis.

Chapter 2

Overview of Approach

2.1 Terminology

We use the following terminology:

- *symptom*: identifying characteristic in behavior of interest. It may be something in the program's output, something absent from the program's output, that the program loops infinitely, etc. It may represent a bug.
- *variant*: program derived from the original program. A variant is smaller in size than the original program.
- *good variant*: a variant that exhibits the symptom.
- *bad variant*[1]: a variant that does not exhibit the symptom.
- *best variant*: the smallest *good variant* found by the chipper.
- *minimal variant*: the smallest possible good variant. It may not be found by the chipper.
- *simplification*: process that produces a new variant from another variant (the 0^{th} variant is the original program).

[1] In the course of this work, we have also seen some ugly variants ;-)

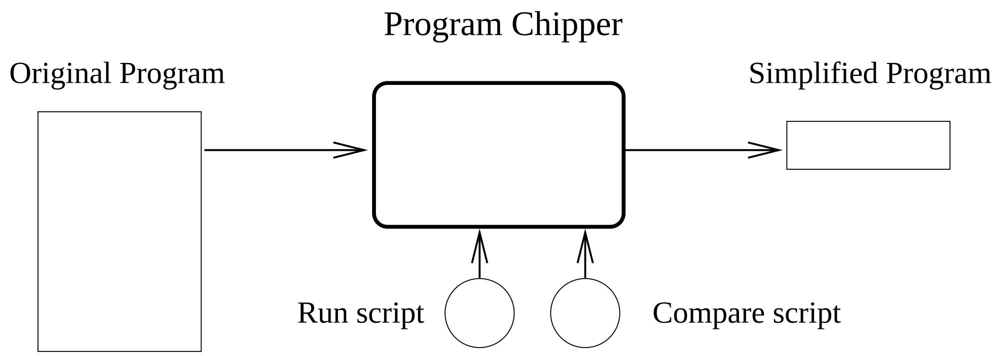


Figure 2.1: Overview of program chipping

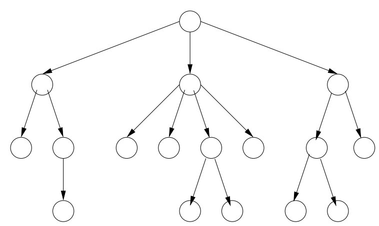
We use the terms *smallest* and *smaller* with respect to some application-specific measure, e.g., number of nodes in a parse tree or number of non-blank lines of source code

As indicated in Chapter 1, our general approach is to automatically reduce a program to a variant that exhibits the same symptom. Our approach, program chipping, is "light-weight" (even naïve) compared to other techniques such as program slicing. In short, our work uses the idea of data slicing [9], where the data is the original program. Our approach is to simplify the original program in various ways, based only on the syntactic structure of the program. In doing so, we produce variants and execute each one to see whether it is good or bad.

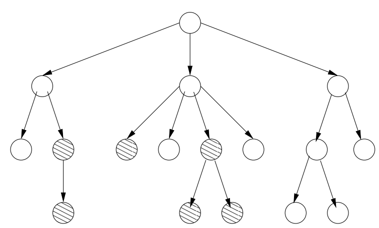
The variants are derived from the original program so they, of course, contain many of the same statements. However, in some cases the simplifications can modify some statements, so that they are not identical in the two programs. Also, the execution behaviors of the variants can differ from that of the original program. Our approach requires only that a good variant exhibit the same symptom, not that that symptom manifests itself for the same reason. Section 5.2.2 discusses this issue in detail.

Figure 2.1 shows a high-level view of a program chipper. The chipper takes a program and two scripts. The Run script specifies how to compile and run the program. The Compare script specifies the symptom; more precisely, it specifies what to look for in the results from the Run script and how to do so. The goal is for the resultant program to be considerably simpler (e.g., shorter in length) than the original, so that finding the symptom in the remaining code will be easier.

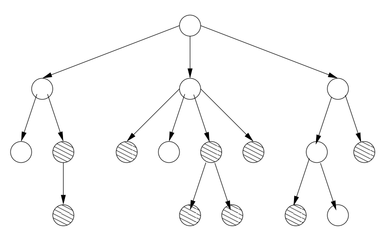
In more detail, the program chipper builds a parse tree corresponding to original program, as shown in Figure 2.2(a). From this parse tree, the chipper tool deletes or modifies one or more nodes to generate other parse trees, e.g., Figure 2.2(b). The program variant corresponding to each such parse tree is tested to see if it is "good." Specifically, the tool generates from the parse tree the corresponding source program, compiles and runs it, and determines whether it exhibits the symptom (using both the Run and Compare scripts). This process continues until the chipper identifies a



(a) Parse tree for original program



(b) Parse tree for an intermediate variant



(c) Parse tree for best variant

Figure 2.2: Parse trees during program chipping. Shading indicates that the original node has been deleted.

best variant (parse tree, or set of parse trees), e.g., Figure 2.2(c).

The prior description does not specify the exact process that the tool follows to reduce parse trees, and to determine when to terminate. One possibility is to generate and test all possible variants. Certainly, though, for any reasonably-sized program, that approach is impractical. The key, then, is to identify heuristics that will be useful in practice, as discussed in Chapter 3.

Chapter 3

ChipperJ: a program chipper for Java

We have developed a program chipper for Java, called ChipperJ. ChipperJ is built on top of Sun's Java 1.4.2 Compiler. The rest of this chapter describes the user view of ChipperJ, introduces simplifying transformations made by the chipper, introduces the v-diff tool, gives a complete description of ChipperJ's simplification algorithms, discusses how ChipperJ deals with the key pragmatic issue of infinite loops, and introduces internal chipping.

3.1 User View

The user gives ChipperJ the original program, specifies how to run the program and how to check for the symptom. ChipperJ is concerned only with program runtime errors. Thus, ChipperJ expects that the user gives a fully compilable and runnable program at the onset of the chipping process. In general, the compiler gives fairly specific guidance as to what is causing a syntactic or semantic error, thus we do not include such errors in the problem domain even though ChipperJ *could* be used to isolate such errors[1].

[1]Because some C or C++ language compilers give less descriptive output statements, it might be useful to use a C or C++ chipper on compiler errors.

The compile, run, and comparison of a variant's output to the symptom is orchestrated via the Run and Compare scripts, as shown in Figure 2.1. ChipperJ then produces the best (in the sense defined in Section 2.1) variant. The user can then examine the best variant to determine the exact cause of the given symptom. We use scripts, Run and Compare, to allow the user the flexibility to specify how to compile and run the program, as well as what constitutes the symptom.

In our experience using ChipperJ, Run and Compare are just shell scripts that use one or more invocations of *grep* and other standard UNIX tools. However, they need not be simple scripts, and we imagine for certain programs, that they could contain sophisticated logic. The chipper requires only that the user's program is runnable from the command line (and subsequently from the Run script) and the symptom can be identified by the Compare script.

3.1.1 Run Script Details

```
#!/bin/sh
filePrefix=$1
mainClass=$2
outputFile=$3

(echo compiling the Variant; javac $filePrefix.java) \
&& (echo executing; java $mainClass > $outputFile 2>&1)
```

Figure 3.1: Example of a simple Run script

A typical Run script compiles the variant program, runs it on some specified input according to the user's preference, and saves the program's output to a file. Figure 3.1 shows a simple example of a Run script that takes three arguments: the prefix of the file that should be compiled, the name of the main class, and the name of the output file where the results of the variant run should be stored. It compiles the program by invoking the *javac* command to compile the variant and then executes the variant using the *java* command. In practice, the actual Run scripts used to compile and execute the user's Java program will need to be more complex than this example in order to handle issues such as variants containing infinite loops (discussed in Section 3.6) and to allow for alternative methods of variant compilation (such as internal compilation discussed in Section 3.7).

3.1.2 Compare Script Details

```
#!/bin/sh
file=$1

grep -e "This Is The Symptom" $file.output >/dev/null 2>&1
```

Figure 3.2: Example of a simple Compare script

A typical Compare script searches the program's output for the given symptom. Typical symptoms include erroneous output, null pointer exceptions, and infinite looping. So that users can specify line numbers in the original code as part of the symptom (e.g., that a "NullPointerException" occurred on line 90 of main.java), ChipperJ (by default) preserves line numbers in the variants by replacing with a blank line any entire line of code or comments that is chipped away.

Figure 3.2 shows a simple example of a Compare script. This Compare script takes a single argument: the name of the variant's output file that the chipper wishes to check. In this case, the symptom is that the program being chipped outputs "This Is The Symptom." It checks the output for a symptom by invoking the *grep* command to search for the desired string in the output file.

3.2 Simplifying Transformations

ChipperJ follows the overall simplification approach shown in Figure 2.2. First, ChipperJ builds a parse tree of the original Java program. It then generates variant programs by removing or modifying nodes in the parse tree. The nodes in the tree represent Java syntactic structures such as `if` statements, `while` loops, variable declarations, output statements, expressions, etc. For each variant, ChipperJ checks whether the variant exhibits the same symptom as the original program. If it does, ChipperJ follows the simplification path of trying to reduce that variant further.

To limit to a reasonable number the amount of variants ChipperJ checks, it uses a left-to-right, top-to-bottom traversal of the parse tree. ChipperJ tries simplification at each node *at most once*; some nodes in the original parse tree are not considered for simplification and some are pruned before ChipperJ reaches them. ChipperJ might perform some simplification(s) on a node depending on the node's type. For example, it tries to simplify a statement block in a variety of ways (as described later), but does not try to simplify an assignment statement. For any given node type, ChipperJ will attempt a few different kinds of simplifications on the node. The number of specific simplifications ChipperJ tries at each node is bounded as determined by the type of node and its contents. Examples: for an assignment statement, the bound is zero; for an `if` statement, the bound depends on the number of arms within the `if` statement; and for a block, the bound depends on the number of statements within the block. A more detailed analysis of the complexity of the chipper's simplification algorithms can be found in Section 3.5.5.

For any simplification ChipperJ performs, it checks, using the Run and Compare scripts, whether the simplification produces a good variant. If the variant is good, then chipping continues in this new good variant. If the variant is bad, then chipping continues in the current good variant, i.e., the bad variant is not considered further. In either case, simplification continues at the same node with the next possible simplification for that particular node. The evaluation of a variant (i.e., good or bad) will determine the next simplification that the chipper will attempt, thus the simplification path for two syntactic structures of the same type and comparable size will vary significantly based on the symptom and what contribution that the structures make to the symptom. If the chipper encounters a good variant during the simplification process for some syntactic structure, it disregards the previous composition of that structure in favor of the new simplified variant of that structure. This ensures that the parse tree is always being reduced in size. If no simplifications remain to be tried, simplification continues at the next node and if no nodes remain, the chipper is finished.

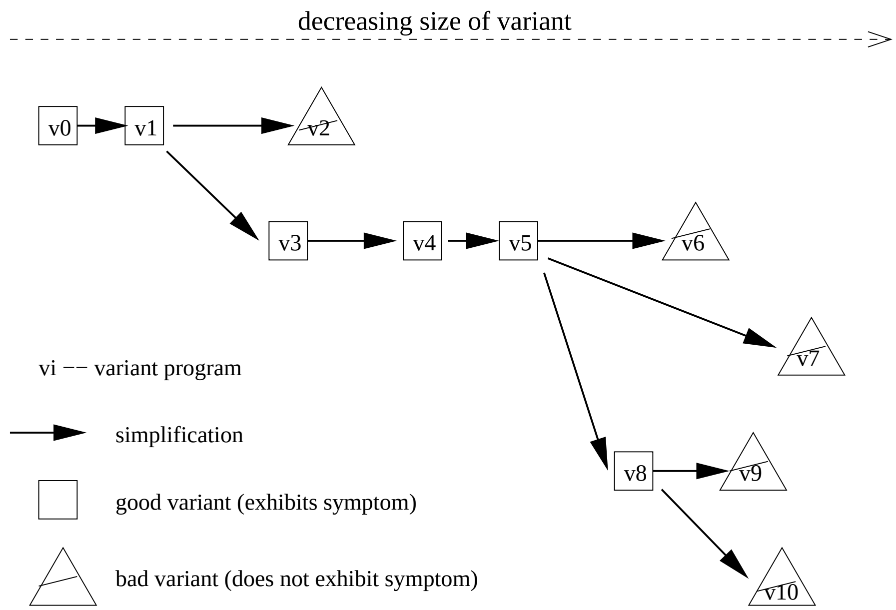


Figure 3.3: Sample ChipperJ simplification process

Note that a variant can be bad because its execution does not exhibit the symptom or because it does not compile, e.g., if some necessary variable declaration has been chipped away. Generating bad variants does not affect the overall search for the best variant, except for the time it takes to generate and compile the variant and, if it compiles successfully, to run and check the bad variant's output for the symptom.

Figure 3.3 illustrates a sample simplification process. The first simplification ChipperJ tries on the original program (variant v0) results in good variant v1. The first simplification ChipperJ tries on v1 results in bad variant v2, so ChipperJ tries another simplification on v1, which results in good variant v3. The process continues until ChipperJ has no further simplifications to try. In this example, v8 is the best variant. Note that each simplification produces a smaller variant. However, the variants produced from a given good variant can vary in size, e.g., when trying to simplify a block, as described in the next section.

3.3 *V-Diff*

We also developed a GUI based tool called *v-diff* (short for Visual Differentiator). It displays a graphical representation of the variants created by ChipperJ. Figure 3.4[2] shows the graphical interface of the *v-diff* tool. The *v-diff* tool was primarily developed to help us debug the chipper and ensure that the sequence of simplifications happened as expected. However, we found it to be a useful tool during the process of isolating the symptom as well.

V-Diff provides a feature to compare two different variants that can be selected from a list of variants or that can be selected from a graphical view of the variants that shows the order in which they were created, differentiating the good from the bad variants.

V-Diff also provides a way to view the progression of variants in order to highlight the differences between the variants. The variants are shown two at a time in a frame by frame succession; thus, this feature is appropriately named *movie mode*. Before viewing a sequence of variants, the user can opt to either see all variants in comparison with the original program or to see each variant in comparison with the variant that immediately precedes it. Instead of developing our own differentiating tool to show the differences between files, we use tkdiff, which is a graphical front end to the diff program. It provides a side-by-side view of two files with the differences highlighted [30]. The movie mode also lets the user specify the starting point and length (in number of variants to be viewed) of the sequence of variants that the user wishes to view.

Movie mode is helpful because it visually highlights code areas that are required to produce the symptom and that may be directly related to the bug that the user is trying to isolate. This is especially helpful when only looking at the sequence of good variants and not including the bad variants in-between them.

Figure 3.4 shows the *v-diff* tool in action. The right panel contains a scrollable, clickable canvas that shows the good variants (light shading) and bad variants (dark

[2]Because this thesis is printed in black and white, the colors used by *v-diff* to differentiate between good and bad variants do not appear in the figure.

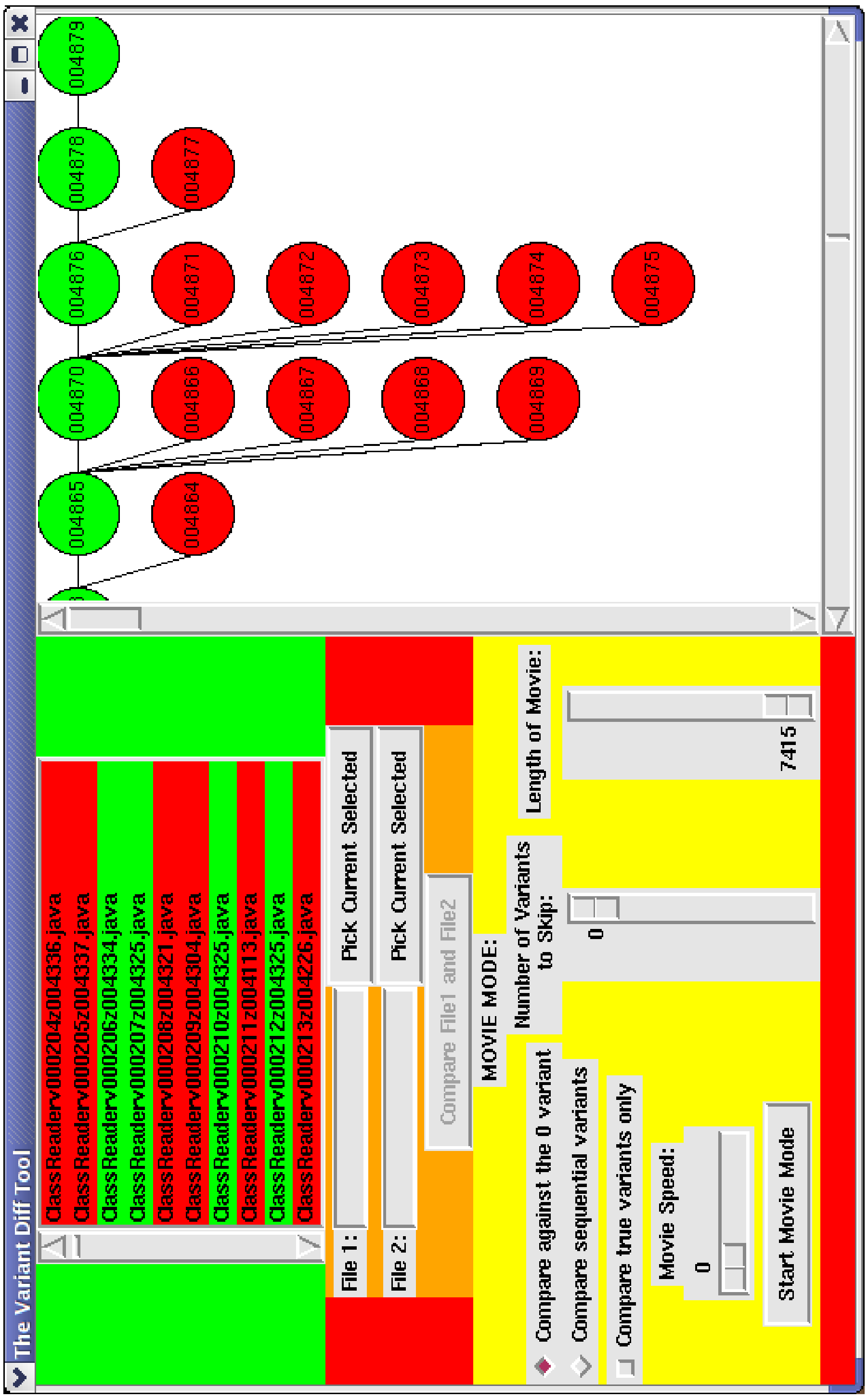


Figure 3.4: Screen shot of the *v-diff* tool

shading). The variants are identified by their variant number. The top left panel is a scrollbox with a list of all of the variants for a particular run of the chipper. ChipperJ uses the convention:

`<File Prefix>v<Variant Number>z<Variant Size>.java`

to name the variants. Below that scrollbox is a control panel with two textboxes. They can be filled with the names of variants selected either from the scrollbox or from the scrollable canvas. Doing this allows the user to directly compare two variants. On the bottom left hand side is the control panel for the movie mode option.

Sometimes, a symptom is not specific enough to isolate the proper bug (Section 5.2.2). When this happens, the chipper may create a variant that produces the same symptom that the user is trying to isolate, but for a different reason than the user's original intent. This variant would be perceived by the chipper as a good variant. This will divert the Chipper from simplifying along the path of the *true* symptom (i.e., the user's intended symptom) in favor of the *faux* symptom. By looking at the sequence of variants using *v-diff*, the user can see where the *faux* symptom is produced. This is helpful in the process of increasing the specificity of the symptom or modifying the code of the original program so that the *faux* symptom is no longer produced, thus keeping the chipper on track to isolate the originally intended symptom.

3.4 Specific Simplifications

This section outlines the simplifications that ChipperJ performs on various Java syntactic structures. Section 3.5 mirrors this section in that it goes into detail for all of the simplifications outlined here.

3.4.1 Simplifications on Blocks

The block component is the primary syntactic structure with which most Java programs are developed. ChipperJ tries, in the order given, the following simplifications on blocks.

SB-1. If the block is the body of a method, eliminate the entire block, leaving only the surrounding braces (possibly with a `return` statement, see below).

SB-2. *Binary search*, also known as *block splitting*. ChipperJ splits the statement list in block B into two halves, $B1$ and $B2$.[3] It first removes $B2$ from the parse tree. If the resulting variant is good (i.e., still exhibits the symptom), then ChipperJ attempts (recursively) the statement block simplifications on the first half starting with the binary search step. Otherwise, ChipperJ removes $B1$ from the parse tree and tries this simplification on $B2$. If simplification fails on each of $B1$ and $B2$, then ChipperJ restores B into the parse tree.

SB-3. Eliminate one statement at a time from the remaining statements in the block. If the statement removal produces a bad variant, then the statement is restored before the chipper proceeds to the next statement. Otherwise, the statement removal is permanent and the chipper continues with the next statement.

SB-4. Simplify each remaining statement. For example, if a `while` loop (the whole of which is considered a single statement) cannot be entirely eliminated by the previous step, then ChipperJ will try to simplify the `while` loop according to the `while` loop simplification scheme as outlined below.

For a particular block, the recursive nature of the *block splitting* part of the algorithm may lead the Chipper to take many steps during the block simplification.

One complication arises when, for example, ChipperJ removes the entire body of a non-void method. Without a `return` statement the variant will not compile, will automatically lead to a bad variant, and will mislead the chipper into thinking that some portion of the method body contributes to the symptom. In this case, ChipperJ would restore the method body and try the remaining block simplifications. Although restoring the method body is not harmful to the simplification process, the size of the best variant ChipperJ finds may not be as small as it could be, and even if it is

[3]ChipperJ uses the number of statements in the block to determine where to split the block and treats declarations as statements.

as small, the chipper has wasted time attempting simplifications on a block that has no relevance to the symptom.

Type-Appropriate Replacement Block	**Return Type Of Method**
{ }	void
{`return ' ';`}	char
{`return 0;`}	int or any numerical type
{`return false;`}	boolean
{`return null;`}	any kind of Object or String

Table 3.1: Type-appropriate replacement blocks ChipperJ generates

So, instead of replacing the method body with an empty block, ChipperJ restores it with a block containing a type-appropriate `return` statement (Table 3.1).

ChipperJ has the ability to insert type-appropriate `return` statements during the block splitting phase of block simplification as well (SB-2). If the block is the body of a non-void method, then it is likely that removing the second half of the block will also remove the `return` statement at the end of the method, thus leading to a bad variant. Thus, ChipperJ checks the first half of the block for a `return` statement and adds one of the type-appropriate `return` statements from Table 3.1, if necessary, before testing to see if the variant exhibits the symptom

All of the other syntactic structures ChipperJ operates on are statements. Because statements always occur within a surrounding block, we make certain assumptions. Recall that in item SB-4 (see above) ChipperJ tries to remove each statement that remains in the block. If, after that step, a statement still remains in the block, it is only because the statement's removal has somehow contributed to a bad variant. Thus, the following simplifications do not remove the statements entirely, but instead chip away at them in hopes that a simpler form will still exhibit the symptom.

3.4.2 Simplifications on `if` Statements

ChipperJ treats a sequence of `if-else if-else if ... -else` statements together as a single if statement. ChipperJ first tries to replace the sequence with the

```
...
if (short == i){
 k = 20 - i;
 out.println(k);
}
else if (15==i){
 k = 15 - i;
 n = 5;
 out.println(k*10 );
 out.println(n);
 out.println(long);
}
else{
 k=10-i;
 out.println(k);
}
...
```

(a)

```
...
{
 k = 20 - i;
 out.println(k);
}
...
```

(b)

```
...
{
 k = 15 - i;
 n = 5;
 out.println(k*10);
 out.println(n);
 out.println(long);
}
...
```

(c)

```
...
{
 k=10-i;
 out.println(k);
}
...
```

(d)

Figure 3.5: Example illustrating the simplification of an `if` statement

body of one of its arms, one arm at a time. Figure 3.5 illustrates this simplification. Figure 3.5(a) is the original `if` statement. Figures 3.5(b)-(d) show the variants that ChipperJ would create when trying to replace the sequence of `if-else if ... -else` statements with the body of one of the arms. If any one of the replacements produces a good variant, the chipper proceeds by simplifying that body as the finishing step for the `if` simplification and then moves on to the next node in the parse tree without restoring the rest of the `if` statement. If, however, each of these replacements produces a bad variant, ChipperJ restores the sequence before continuing with the next simplification.

It next tries, starting at the beginning of the sequence, to remove each arm one at a time. If the removal of an arm produces a bad variant, the arm is restored, and ChipperJ attempts to remove the next arm. However, if a good variant is produced, the arm is removed permanently before continuing on to the next arm in the sequence.

Afterwards, ChipperJ simplifies any arms that still appear in the good variant that results from these simplifications. If an arm body is a block, then ChipperJ proceeds with the block simplification. If instead an arm body is a statement for which we have developed a simplification, then that simplification is applied to the statement. Otherwise, ChipperJ moves on to the next remaining arm body and, upon completion, on to the next node in the parse tree.

3.4.3 Simplifications on Loops

Java supports three kinds of loops: `while`, `do-while`, and `for`. ChipperJ tries to simplify each kind of loop by replacing the loop by one iteration of the loop's body. For example, the `while` loop:

```
while (x!=0) {S}
```

becomes just:

```
{S}
```

the `for` loop:

```
for (int i = 0; i < N; i++) {S}
```

becomes:

```
{int i = 0; {S}}
```

and the `do-while` loop:

```
do {S} while(x > 1)
```

becomes:

```
{S}
```

(One can view this simplification as "loop unrolling", as used in compiler optimizations, but here we replace the loop by only its the initial iteration.)

Thus, if the symptom is coming from loop's first iteration, removal of the surrounding loop structure will make it all the more evident to the user.

Notice that the simplification for the `for` loop is slightly more specialized than that of the `while` and `do-while` loops. The construct of the `for` loop contains an initialization expression. Often, the initialization expression is a variable declaration for some variable used inside of the loop body. If the declaration was omitted with the rest of the surrounding loop structure when trying the loop simplification, it would increase the chances that trying to simplify the loop to one iteration of the loop's body would fail automatically because of a compile time error. Furthermore, initializing the loop's variable ensures that the one-time iteration of the `for` loop will occur as it would if the loop were to execute through the first iteration as normal.

ChipperJ continues simplification by simplifying the loop's body. The body simplification is handled in the same manner as the `if` arm body simplification at the end of Section 3.4.2.

3.4.4 Simplifications on `switch` Statements

The simplification for `switch` statements is a hybrid of the `if` simplification and the block simplification.

ChipperJ starts by trying to replace the entire `switch` statement with the body of one of its `cases`. If any of these replacements produces a good variant, ChipperJ proceeds by trying the block simplification on that `case` body before proceeding to the next node. If, however, all of those replacements produce bad variants, ChipperJ continues trying to simplify the `switch` statement.

ChipperJ next tries block splitting at the `case` level. It starts by splitting the list of cases into two halves and testing to see which half will produce a good variant. When splitting no longer produces a good variant, ChipperJ takes the remaining case statements with their associated bodies and one at a time tries to remove them. Subsequently, the chipper simplifies the bodies of the remaining `case` statements by using the aforementioned block simplification strategy.

ChipperJ operates on the `switch` statement roughly in the same fashion it operates on blocks, but it operates on the `case` level instead of the statement level. The first step in this simplification is a binary search step similar to the block simplifications. ChipperJ splits the `case` list C into two halves, $C1$ and $C2$.[4] It first removes $C2$ from the parse tree. If the resulting variant is good (i.e., still exhibits the symptom), then ChipperJ attempts (recursively) the `switch` block simplifications on the first half. Otherwise, ChipperJ removes $C1$ from the parse tree and checks to see if restoring

[4]ChipperJ uses the number of cases in the `switch` statement to determine where to split the list of cases.

$C2$ will create a good variant. If so, ChipperJ attempts (recursively) the `switch` block simplifications on the $C2$. If simplification fails on each of $C1$ and $C2$, then ChipperJ restores C into the parse tree.

3.4.5 No Other Simplifications

ChipperJ does no other simplifications. It does *not* attempt to simplify assignment statements or expressions, discard entire methods, etc. In general, the only way that ChipperJ can affect other statements is to remove them, for example as part of the block simplification (Section 3.4.1). The one exception is the `return` statement, which can be replaced by a type-appropriate return statement with a default return value (Table 3.1).

3.5 Tree Chipping Algorithm Details

This section presents an in-depth look at the intricacies and idiosyncrasies of ChipperJ's simplification algorithms. Subsections 3.5.1 through 3.5.4 correspond with those in Section 3.4. Subsection 3.5.5 gives an analysis of the complexity of the simplification algorithms.

For many of these simplification algorithms, it was necessary to implement a special print mode for certain syntactic structures. The special print mode is used when ChipperJ wants to try a simplification that changes the structure type of the syntactic structure that is being simplified. Using a special print mode allows ChipperJ to create the desired variant by changing the way the structure is printed out, without losing other structure-specific information contained within the structure (just in case the variant is bad). The simplifications in Sections 3.5.2, 3.5.3, and 3.5.4 use the special print mode.

3.5.1 Block Simplification Details

The block simplification algorithm (Algorithm 1) starts by splitting the block. A general block must have at least two statements in order for ChipperJ to try to do the

Algorithm 1 Block Simplification Pseudocode

```
Block Splitting
for each remaining statement i do
    remove statement i
    test variant
    if variant is bad then
        restore statement i
    end if
end for
for each remaining statement j do
    try to simplify statement j
end for
```

binary search or block splitting part of the block simplification algorithm (Algorithm 2). ChipperJ uses two statements as the base case for its recursion in Block Splitting. The rationale is that if ChipperJ intends to split a block into two blocks, both of those blocks must be non-empty. If a type-appropriate `return` statement has been added to the block (Table 3.1), the minimum size of the block must be three statements. In this case, the `return` statement is added to the block (Algorithm 2, lines 2 and 8), in a previous iteration of the Block Splitting Algorithm. If there were less than three statements in the initial block, then either one or both of the blocks created would be empty or one of the blocks would contain only the type-appropriate `return` statement. The former case has already been discussed. The latter case, however, is simply a repeat of the previous variant as described in Section 3.4.1 in item SB-1 where the entire block is replaced by a type-appropriate generic block. Thus, three is the minimum number of statements that will fulfill our requirement of non-empty blocks while still creating new variants when a type-appropriate return statement has been added.

ChipperJ saves the original block of statements in a temporary variable and replaces it with an empty block. The next step is to insert the first $n/2$ statements into the block (Algorithm 2, line 1). The chipper checks to see if this block requires a re-

Algorithm 2 Block Splitting Pseudocode

```
replace block of n statements with the first n/2 statements
add type-appropriate return if necessary
test variant
if variant is good then
    recurse on the Block Splitting Algorithm
else
    replace block with the last n/2 statements
    add type-appropriate return if necessary
    test variant
    if variant is good then
        recurse on the Block Splitting Algorithm
    else
        restore the block
    end if
end if
```

turn value, and whether or not the last statement is a `return` statement (Algorithm 2, line 2).

It is fairly simple to check the last statement in a block, with only two concessions. The Java Language allows the last statement in a block that requires a `return` statement to be an `if` statement or a `switch` statement only if all of the arm bodies of the sequence of `if-else if-else if ... -else` statements end with a `return` statement or if all of the `case` bodies in the `switch` statement end with a `return` statement, respectively.

If ChipperJ determines that a `return` statement is necessary, it uses parameters passed in from the calling method to discern which type-appropriate `return` statement is necessary and appends it to the end of the block. Now that the block is prepared, ChipperJ checks the variant to see if it still exhibits the same symptom. If so, it recurses on the block splitting method (Algorithm 2, line 5).

If not, ChipperJ replaces the block with the empty block and then inserts the second half of the original block into the block's statement list (Algorithm 2, line 7). It proceeds with the second half of the block in the same manner as the first. Once the block no longer produces good variants by being split, the chipper moves to the next simplification step.

Next (Algorithm 1, lines 2-8) ChipperJ walks through the list of remaining statements in the block and attempts to remove each statement one at a time (Algorithm 1, line 3), to determine which statements must be kept in order to produce a good variant. This portion of the algorithm is very straightforward. The chipper iterates through the list of statements. On each iteration, ChipperJ saves the statement in a temporary variable and removes it from the list. Next, ChipperJ checks to see if the variant produced still exhibits the symptom. If so, ChipperJ proceeds to the next statement and the removed statement is disregarded. If this simplification produces a bad variant, ChipperJ restores the statement before proceeding to the next statement (Algorithm 1, line 6).

After trying to individually remove the remaining statements, ChipperJ iterates through the remaining statements in the block and attempts to simplify each one (Algorithm 1, line 10). Recall that not every statement has a simplification. If

ChipperJ encounters a remaining statement that it cannot simplify, it does nothing to that statement and moves to the next statement. If the statement encountered is a statement for which a simplification has been developed, then ChipperJ attempts that simplification. The details of the remaining simplifications are in the following three sections.

3.5.2 `if` Statement Simplification Details

In order to understand how the `if` statement simplification works, it is important to first understand how a sequence of `if-else if-else if ... -else` statements is represented in the parse tree. Figure 3.6 shows sample code for an `if` statement.

```
if ( Expression1 ){
  Block1
}
else if ( Expression2 ){
  Block2
}
else{
  Block3
}
```

Figure 3.6: Code for a sample `if` statement

When parsed by the chipper, this code is represented by the nodes in the `if` tree as shown in Figure 3.7. Notice that the `else if` component of the `if` tree is represented as another `if` tree located in the *elsepart* of the first `if` statement. The `else` statement is represented as a regular block (or a singular statement) in the *elsepart* of the last `else if` statement.

In order to do the `if` block arm replacement part of Algorithm 3 (line 2), ChipperJ builds a list of arm bodies, where each arm body comes from the *thenparts* in the `if` tree. ChipperJ has to work within the construct of the `if` statement to create the effect of replacing the entire `if` statement with a single block. To replace the `if` statement with an arm body, ChipperJ must ensure that the condition of the initial `if` is not printed and that only one block is printed. It does this by enabling the

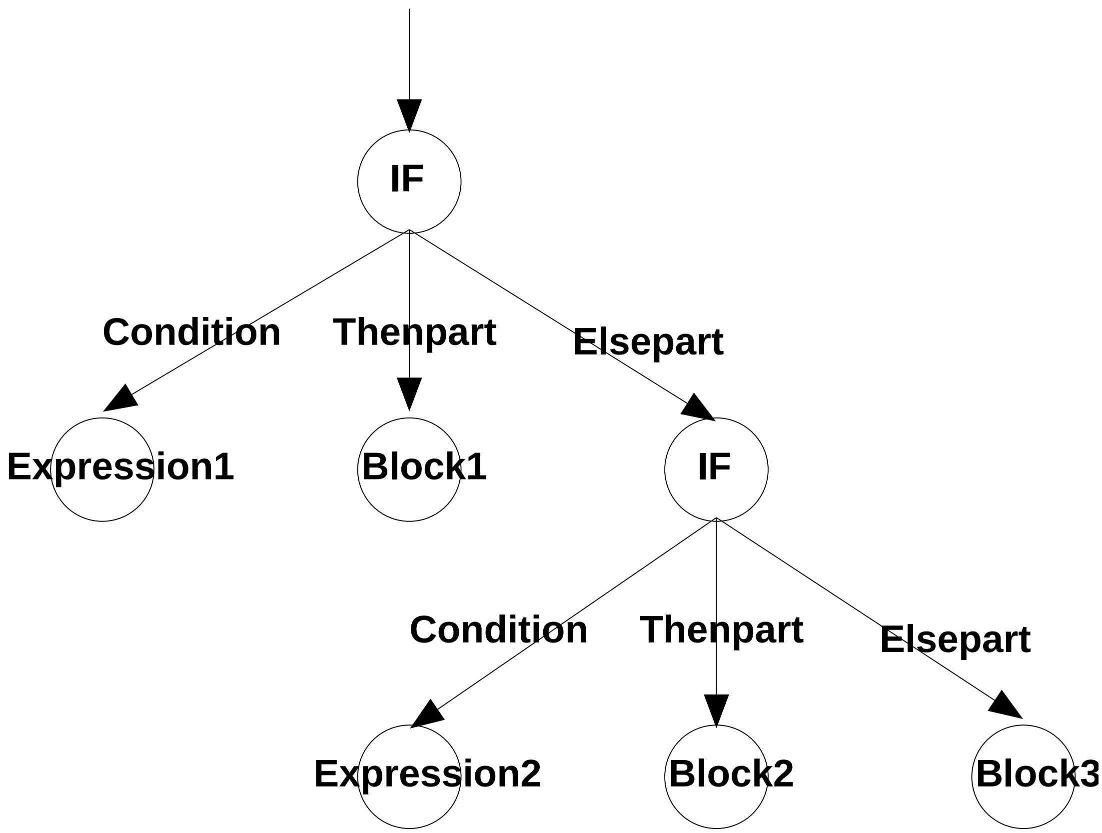


Figure 3.7: Parse tree for the `if` statement in Figure 3.6

Algorithm 3 `if` Simplification Pseudocode

```
for each arm i in the if statement do
  replace the entire if statement with the body of arm i
  test variant
  if variant is good then
    simplify body of arm i
    return
  end if
end for
for each remaining arm j do
  remove arm j
  test variant
  if variant is bad then
    restore arm j
  end if
end for
for each remaining arm k do
  simplify the body of arm k
end for
```

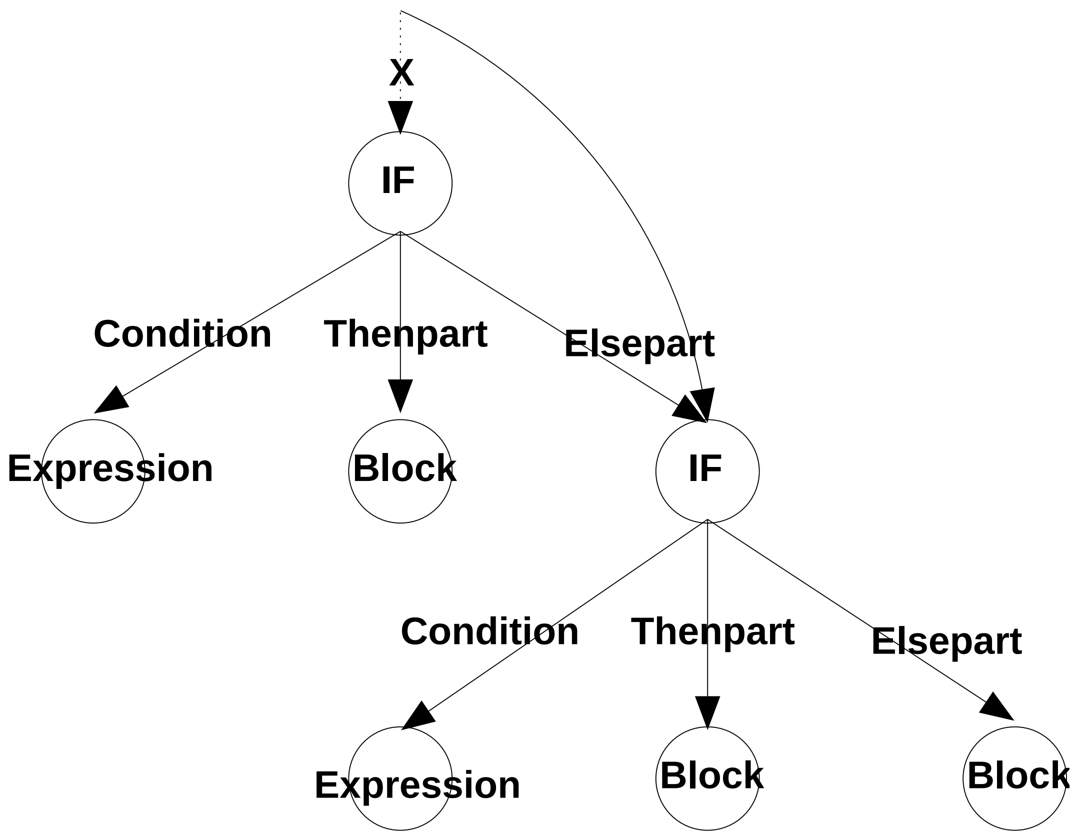


Figure 3.8: Example of the elimination of an arm of the `if` statement

`if` statement's special print mode, which was referred to at the beginning of Section 3.5. This special print mode only prints the first *thenpart* of the entire `if` statement. Because we have this option, it is easy to then replace the topmost *thenpart* with each of the arm bodies one at a time. ChipperJ checks each arm body to see if a good variant is produced. If not, it checks the next arm body until the list is exhausted. If it does find a special arm body that produces a good variant, ChipperJ simplifies that arm body before exiting the `if` statement simplification algorithm.

If no good variant is produced, ChipperJ restores the original `if` statement by restoring the correct *thenpart* in the topmost `if` statement and turning off the `if` statement's special print mode. Next, ChipperJ tries to remove one arm at a time (Algorithm 3, line 10). Because each `else if` statement is an `if` statement that is located in the *elsepart* of the preceding `if` statement, ChipperJ can remove an arm by setting the sequence of `if-else if-else if ... -else` statements to start at the `if` tree located in the *elsepart* of the `if` tree that originally starts the sequence as shown in Figure 3.8.

This has the effect of eliminating the preceding arm without actually having to permanently remove it. If the arm removal produces a good variant, then we leave that arm out of the `if` statement. However, if the variant is bad, we restore the arm before proceeding to the next arm.

After ChipperJ has attempted to remove each of the arms and has restored the arms necessary to produce a good variant, the chipper tries to simplify the remaining arm bodies. If a body is a block, ChipperJ uses Algorithm 2 to attempt to simplify it. Otherwise, ChipperJ will utilize an appropriate simplification algorithm depending on the composition of the loop body. If no simplification has been defined, ChipperJ moves on to the next node (Algorithm 3).

3.5.3 Loop Simplification Details

Algorithm 4 Loop Simplification Pseudocode

```
1: remove loop's enclosing structure
2: test variant
3: if  variant is bad then
4:    restore the loop's enclosing structure
5: end if
6: simplify the loop's body
```

The simplifications for all three of the loops that Java supports (Section 3.4.3) are similar, thus they will be covered together. In order to replace the loop by one iteration of the loop's body, ChipperJ must remove loop's enclosing structure (Algorithm 4, line 1). During simplification of a loop, we only have access to the loop's subtree. Thus ChipperJ works within the construct of the loop to replace the loop with one iteration of the loop's body. This is done by using the loop's special print mode (the beginning of Section 3.5). The loop special print mode ensures that the loop is printed correctly for a one-time iteration. In the case of the `while` loop, this means that the `while` keyword, the beginning condition, and the parentheses enclosing the condition are not printed, leaving only the `while` loop body. In the case of the `do-while` loop, the reserved words `do` and `while`, the condition, and the parentheses enclosing the loop's condition are not printed.

As mentioned in Section 3.4.3, the `for` loop is a slightly specialized case of the loop simplification strategy. The specialization comes from the way that special print mode is implemented for `for` loops. The condition for a `for` loop consists of three expressions: loop variable initialization, loop termination, and loop variable step. The initialization expression sets up the loop. It is executed one time at the beginning of the loop and, as its name indicates, it is meant to initialize the loop variable so that it falls within the desired ranges as defined by both the initialization expression and the termination statement. The discussion in Section 3.4.3 describes why the initialization expression is not omitted during `for` loop simplification.

Thus, the special print mode for the `for` loop removes the `for` keyword, the parentheses, the termination expression, and the step expression. It does not get rid of the initialization statement or the loop body. In addition, the special print mode encloses the initialization and loop body in a set of braces.

If the initialization expression contains a variable declaration, that variable is only local to the body of the loop. The chipper prints the initialization outside of the loop body. If, however, there was another for loop or other syntactic structure wherein a variable of the same name was declared, simply printing the initialization statement outside would cause the compiler to complain that the variable was already defined. For example, Figure 3.9 shows a sample of code in a valid Java program where two different `for` loops use the same name for the loop variable. Simplifying the first for loop (lines 1-4) to a one time iteration proceeds as follows:

```
{int i = 1;
{
    System.out.println(i);
}}
```

Notice the second set of braces that encloses the initialization statement and block.

```
...
1  for(int i = 1; i < 10; i++)
2  {
3      System.out.println(i);
4  }
...
5  for(int i = 20; i > 0; i--)
6  {
7      System.out.println(i);
8  }
...
```

Figure 3.9: Code snippet of for statement

These braces restrict the scope of the variable `i` to be local only to the loop body. Thus, it will not conflict with the second variable `i` declared in the initialization statement of the second `for` loop (lines 5-8).

If removing the various components of the loop's structure produces a bad variant, the loop is restored (Algorithm 4, line 4). This is done easily by simply turning off the loop's special print mode. ChipperJ then tries to simplify the body of the loop. If the loop's body is a block, then it will use Algorithm 1. Otherwise, ChipperJ will utilize an appropriate simplification algorithm depending on the type of statement that comprises the loop body. If no simplification has been defined, ChipperJ moves on to the next node.

3.5.4 Switch Statement Simplification Details

The `switch` statement simplification algorithm is split into four steps, as outlined in Algorithm 5.

In order to do the `case` body replacement, ChipperJ builds a list of `case` bodies such that each item in the list is a block from all of the cases in the `switch` statement.

ChipperJ has to work within the construct of the `switch` statement to create the effect of replacing the entire `switch` statement with a single block. To replace the `switch` statement with an arm body, it has to ensure that the condition of the `switch` is not printed and that only one `case` body is printed. We do this by enabling the

Algorithm 5 Switch Simplification Pseudocode

```
for each case i in the switch statement do
  replace the entire switch statement with the body of case i
  test variant
  if variant is good then
    simplify body of case i
    return
  end if
end for
Block Splitting on the case level
for each remaining case j do
  remove case j
  test variant
  if variant is bad then
    restore case j
  end if
end for
for each remaining case k do
  for each statement l (in case k) do
    try to simplify statement l
  end for
end for
```

`switch` statements special print mode (see the beginning of Section 3.5). This special print mode only prints the first `case` of the `switch` statement. We also have to enable the special print mode for the `case` statement so that the `case` keyword will also not be printed. Because we have these options, it is easy to then replace the first `case` with each of the `case` bodies (enabling the special print mode for each one as we proceed) one at a time (Algorithm 5, line 2). ChipperJ checks each `case` body to see if a good variant is produced. If not, it checks the next `case` body until the list is exhausted. If it does find a special `case` body that produces a good variant, ChipperJ attempts to simplify that `case` body as the final step of the Switch Simplification (Algorithm 5, line 5). If, however, none of the `case` block arm replacements produce a good variant, ChipperJ proceeds to the next part of the simplification.

The next step in the `switch` statement simplification is once again "block splitting" (Algorithm 5, line 9 and for further details see Algorithm 2). The difference here is that we operate from the `case` level instead of from the statement level (as in the Block Simplification). ChipperJ saves the remaining list of cases in a temporary variable and replaces the list of cases in the `switch` statement with the first $n/2$ cases. ChipperJ checks the variant to see if it still exhibits the same symptom. If so, it recurses on the `switch` simplification starting at the block splitting step.

If not, ChipperJ replaces the list of cases with second half of the original list of cases. It proceeds with the second half of the `case` statements in the same manner as the first. If ChipperJ determines that this variant is good, it again recurses, starting again at the block splitting for cases step. If this variant is bad, ChipperJ restores the list of cases and moves on to the next part of the `switch` simplification algorithm.

Next, ChipperJ walks through the list of remaining cases in the `switch` statement and attempts to remove each `case` one at a time, to determine which cases we must keep in order to produce a good variant (Algorithm 5, line 11). This portion of the algorithm is very straightforward. The chipper iterates through the list of the cases. One at a time, each `case` is saved in a temporary variable and it is removed from the list. Next ChipperJ checks to see if the variant produced still exhibits the symptom. If so, ChipperJ proceeds to the next `case` and the removed `case` is disregarded. If this simplification produced a bad variant, ChipperJ restores the `case` before proceeding to the next `case` (Algorithm 5, line 14).

After trying to individually remove the remaining `case` statements, ChipperJ iterates one final time through the remaining `case` bodies in the `switch` statement and attempts to simplify the remaining statements in each `case` body (Algorithm 5, line 19).

3.5.5 Complexity Analysis

Recall from Section 3.2 that one of the main goals for ChipperJ was to limit to a reasonable number the amount of variants ChipperJ produces. The actual experience in using the chipper (complete with number of variants, simplification times, and size of the best variant's parse tree) is documented in Chapter 4. The purpose of this section is to give a rough idea of the complexity associated with the various algorithms discussed in Section 3.4. During this analysis, the following notation will be used:

- N represents the total number of nodes in the parse tree of the program being chipped (Section 3.2)
- S represents the number of statements in a block
- A represents the number of arm bodies in an `if` statement
- C represents the number of `case` bodies in a `switch` statement

The block syntactic structure can be composed of various kinds of statements. Among them are the `if`, `switch`, and loop (`for`, `while`, and `do-while`) statements, each of which is represented by a node or set of nodes from the N nodes in a parse tree. In addition, an arm body of an `if` statement, a `case` body of a `switch` statement, and a body of a loop may consist of either a block, or a singular statement. The recursive nature of Java's syntactic structure allows the parse tree to vary substantially from program to program. It can range from being very balanced in structure, to very unbalanced where syntactic structures are nested inside of other syntactic structures.

Each node in the parse tree is the focus of simplification only one time, although it may play a role in the simplification of one of its ancestor nodes. In addition, there is a bounded number of simplifications tried by the chipper for each node type (Section 3.4). From this point of view, we process each node at most one time during the entire chipping process, so for a parse tree of N nodes, the overall chipping process is linear or $O(N)$ in the amount of simplifications it tries. It is important to note, however, that each simplification scheme very much depends on the type and size of the syntactic structure being simplified. Thus, there is not always a constant amount of work associated with them, meaning that the overall chipping process is more than linear.

Algorithm 2 showed the recursive Block Splitting Pseudocode. On each execution of this algorithm (initial or recursive), the chipper must replace the original block of S statements with the first $S/2$ statements. This is done in time linear in the size of S. However, on each recursive execution of the algorithm, the initial list of statements is at most half the length of the original list of statements from the previous execution. Thus, the work to do the splitting can be summed as:

$S + S/2 + S/4 + ...S/S$

which is a simple geometric or exponential series. Simple reduction shows that the summation can be reduced to $2S$ for increasingly large values of S. Thus, the complexity of splitting the statements is $O(S)$.

Every time a recursive call is made the statement list is reduced by half and for each call, a new variant is tested. Thus, the upper bound on the number of tests is $O(\log_2 S)$. Thus, the overall complexity of block splitting is

$$O(\log_2 S + S) \qquad \text{which can be reduced to}$$
$$O(S)$$

The portion of the Block Simplification (Algorithm 1) where the block itself is the focus of the simplification contains a call to the Block Splitting Algorithm followed by a `for` loop that iterates through each statement in the block. The analysis for the Block Splitting algorithm has been shown to be $O(S)$. The `for` loop creates a variant and tests it for each statement in the block. Thus, it also takes only $O(S)$

time. Note that the last `for` loop in the Block Simplification Algorithm takes the focus of simplification away from the block and is not considered when determining the complexity of Block Simplification.

The portion of `if` statement simplification (Algorithm 3) that focuses on the `if` statement's structure contains two non-nested loops. The first loop replaces the `if` statement with each of the A body arms sequentially. This loop is linear in the number of arms in the `if` statement: $O(A)$. The second loop again iterates through the arms of the `if` statement. The chipper creates a new variant and tests it. This is done for each arm of the `if` statement. Thus, this second loop is also linear: $O(A)$, making the overall `if` statement simplification linear in the number of arm bodies in the `if` statement. Again, the last `for` loop in the If Simplification Algorithm takes the focus of simplification away from the `if` statement and is not considered when determining the complexity of `if` simplification.

The chipper focuses on a loop during loop simplification (Algorithm 4) for a single step. Removing the loop's enclosing structure is facilitated using the special print mode (Section 3.5). So the actual Loop simplification runs in $O(1)$ or constant time before the focus of the chipper shifts from the loop to its body (Algorithm 4, line 6).

Simplification for the `switch` statement has more steps than the previous algorithms (Algorithm 5). As in the `if` statement simplification, the entire `switch` statement is replaced by each of its C `case` bodies. This happens in time linear in the number of cases, $O(C)$. Next, block splitting on the `case` level happens. The analysis for the Block Splitting algorithm has been shown to be linear in the number of statements in the block. Because the chipper is using this algorithm on the `case` level, it runs in $O(C)$ time. The last simplification attempted before the chipper shifts its focus from the `switch` statement is to iterate through the list of cases and remove them one at a time. This step is also linear in the number of cases contained by the `switch` statement. In total, the `switch` statement simplification runs in $O(3C)$ or $O(C)$ time.

3.6 Dealing with Infinite Loops

The original Java program might itself contain an infinite loop. Indeed, an infinite loop might be the symptom of interest. Moreover, a variant generated by ChipperJ might similarly contain an infinite loop. In some cases, these loops arise due to the simplifications described in Section 3.4. For example, ChipperJ might generate a variant that omits from the body of a `while` loop the assignment statement that increments the loop's index variable.

The simple Run script provided in Figure 3.1 has no way to stop a variant that contains an infinite loop, so for such a variant, the chipper would wait forever for the variant to finish running. Our experimentation with the chipper and encountering infinite loops necessitated the use of the Run script in conjunction with a *timeout* program. By setting a timeout, the user is allowed to set an upper bound on the length of time that each variant can run. The user is responsible for setting the length of the timeout. This is done by timing how long it takes the symptom to be exhibited by the original program and adding the amount of time it takes to compile the code. The more accurate the user is in determining what the timeout should be, the faster the chipper will kill the variants containing infinite loops, and the faster the overall chipping process will be. However, in our experience it is better to slightly overestimate the time. The extra time allows for daemons or other processes running concurrently that may also be vying for CPU time. If a good variant is killed before it has exhibited the symptom, ChipperJ will regard it as a bad variant. While this may have no harmful effects in the grand scheme of isolating the symptom, it is likely to produce a larger than necessary best variant.

The timeout program outputs a special message if the program does not terminate before the timeout occurs. The Compare script can specify to search the program's output for this special message if the symptom is infinite looping.

Figure 3.10 is an example of the Run script using a timeout program. The usage for the timeout program that we use is:

```
./timeout [-s <kill signal>] <timeout length>
```

This example actually uses two scripts: "Run" (Figure 3.10, top) to kick off the

```
#!/bin/sh
#Run script
filePrefix=$1
mainClass=$2
outputFile=$3

./timeout -s 9 5 ./runHelper $filePrefix $mainClass  > $outputFile 2>&1

-----------------------------------------------------------------------

#!/bin/csh -f
#runHelper script
filePrefix=$1
mainClass=$2

(echo compiling the Variant; javac $filePrefix.java) \
&& (echo executing; java $mainClass)
```

Figure 3.10: Example of a Run script using a timeout program and a helper script

timeout program and a helper script named “runHelper”(Figure 3.10, bottom). This Run script takes the same three arguments as the Run script in Figure 3.1. Instead of doing all of the processing on its own, the Run script sends to the runHelper two arguments: the prefix of the file that should be compiled and the name of the main class. In addition, the Run script starts the timeout program to kill the helper script in five seconds. The runHelper script is then responsible for compiling and running the variant. In this case, if the variant has not been compiled and run to completion in five seconds, the timeout program (initiated in the Run script) will kill the variant and allow the chipper to proceed. If the variant has exhibited the symptom in the alloted time, it will be considered good by the chipper, regardless of whether or not the variant was killed by the timeout program.

3.7 External or Internal Chipping

ChipperJ can operate in one of two modes: external or internal. In external mode, ChipperJ writes each variant it produces to a file. The Run script is then responsible for compiling, running, and saving the output of the variant (Section 3.1.1). Next, the Compare script checks the output of the variant's run to see if it contains the symptom (Section 3.1.2). The chipper takes the results from the Compare script to designate a variant as bad or good and then proceeds to the next appropriate step. In internal mode, ChipperJ does not need to write each variant to a file. Instead, it compiles the variant directly from the parse tree, and then applies the Run and Compare scripts to the generated *.class* files. In this case, the Run script is only responsible for running and saving the output of the variant[5]. The duties of the Compare script, however, remain the same. Internal mode can be substantially faster, as will be seen in Section 4. The next section discusses an important aspect of implementing internal chipping.

3.7.1 Snapshot

Two main issues complicate the process of compiling the variant directly from the parse tree that is being chipped:

- Recall from Section 3.5 that certain simplifications require a special print mode. This mode is used when ChipperJ wants to try a simplification that changes the structure type of the syntactic structure that is being simplified. Unfortunately, the special print mode is not recognized by the Java Compiler. Compiling the variant directly from a parse tree will not take the special print mode into account, so the structure is not transformed appropriately and the variant that is created is not what the chipper intended.

- During the process of compiling the variant, the Java Compiler makes modifications to some of the nodes in the parse tree. If compilation of the variant fails because of a semantic error, the changes that the compiler makes to tag and information fields associated with nodes inside of the parse tree can make the parse tree uncompilable from that point on. In this case, none of the future variants would compile, i.e., each of them will be a bad variant.

[5] ChipperJ passes a parameter to the Run script so the script knows whether or not it needs to compile the program.

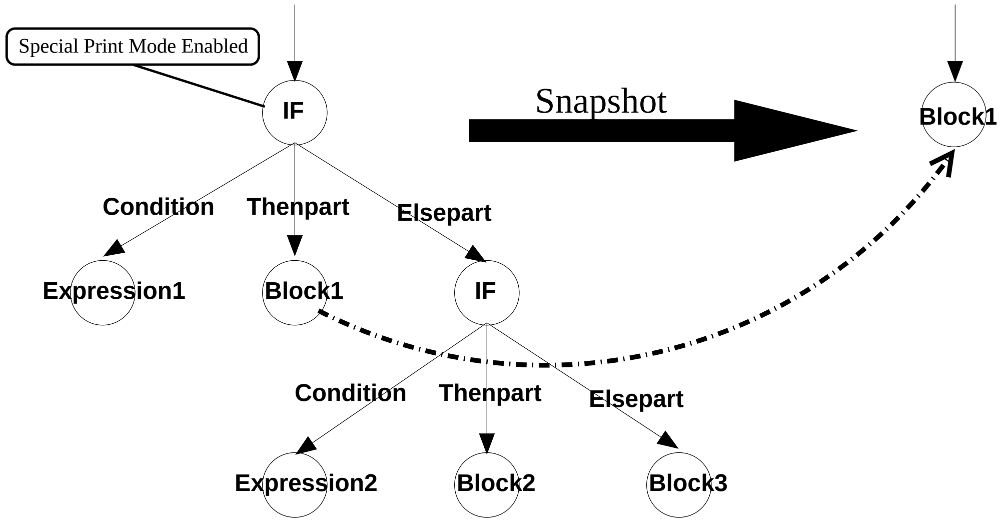


Figure 3.11: Example illustrating the snapshot of an `if` statement

The chipper handles both of these issues by taking a special *snapshot* of the parse tree and feeding the snapshot into the Java Compiler. The snapshot is just a copy of the parse tree for most of the nodes. However, syntactic structures that have the special print mode enabled are copied differently. These structures (Sections 3.5.2, 3.5.3, and 3.5.4) are copied so that the snapshot contains the new structure type as indicated by the special print mode instead of the original structure type.

For example, recall Figure 3.5, which illustrates the simplification of an `if` statement. For each of the simplifications (Figure 3.5(b)-(d)), the `if` statement is changed to a block that contains the statements in the body of one of the arms of the original `if` statement. As noted in Section 3.5.2, this simplification is done via the use of the `if` statement's special print mode. In order to replicate this using the internal compile option, ChipperJ must create a copy of the `if` statement such that the body of the arm replaces the original `if` statement. Figure 3.11 shows how the parse tree is changed in the snapshot of the original parse tree. The dashed line shows which block is used to replace the entire `if` statement.

Creating a copy of the parse tree that represents all of the syntactic structures exactly as the chipper intends ensures that the variants are compiled as the chipper intended. Also by creating a copy of the parse tree to be fed into the compiler, the chipper ensures that no compilation will affect the compilation of future variants because no changes made to the snapshot by the Java Compiler are made to the original parse tree.

Chapter 4

Experience

We have used ChipperJ on a wide variety of programs and for different symptoms. This section reports on our experience on a representative collection of these tests. Section 5 further discusses these results.

The data reported are for tests run on a dual processor 2.8GHz CPU with Red Hat Enterprise Linux 3 (kernel 2.4.21-27.0.2.ELsmp).[1] The results are averages of several test runs, which had insignificant variances.

Table 4.1 summarizes the results of running ChipperJ on a variety of small to large Java programs and for various symptoms. Recall from Section 2 that "best" is not necessarily minimal. "Source lines" in Table 4.1 represents non-blank, non-comment lines. The data in Table 4.1 show that external chipping takes about from 2-5 times as long as internal chipping (Section 3.7).

T1-T3 demonstrate ChipperJ's simplification techniques, especially block splitting, for very basic programs. These test programs consist of blocks of code with a variety of control flow statements. The Compare script for each of these tests specifies a single value to search for in the output. These results highlight the extent to which chipping can reduce a program. In these cases, ChipperJ reduces the Java source to the minimum or near minimum. T4 is also a small program with two classes. The symptom is just a line of output from a method within one of the classes

[1]Because neither the test programs nor ChipperJ is multi-threaded, using a dual processor had no benefit over a single processor.

Table 4.1: ChipperJ results

test	source lines original	source lines best	parse tree nodes original	parse tree nodes best	# of variants	time (H:MM:SS.00) external	time (H:MM:SS.00) internal	notes
T1	21	6	101	11	10	8.13	2.29	small, straightline code
T2	17	6	62	18	13	10.05	2.50	small, if statement
T3	75	20	313	37	30	23.73	5.68	small, bad parameter
T4	22	15	56	28	17	12.59	2.93	small, two classes
T5	596	344	2112	1094	225	54.56	13.98	medium, simple translator
T6	596	364	2112	1116	259	5:12.62	2:30.50	medium, simple translator
T7	701	252	1880	485	140	2:55.45	1:28.28	medium, simple translator
T8	1767	726	5939	1949	530	10:03.60	4:25.75	medium, prototype chipper for simple language
T9	1875	290	26435	3999	1146	22:11.42	7:32.97	worst case, repeated code, repeated symptom
T10	5080	1208	11899	3332	505	6:50.24	1:34.29	medium, bug injected into string library
T11	30329	7275	100137	26892	3262	1:01:52.10	30:24.29	large, *javac*, wrong class name
T12	25148	6201	80819	19978	3332	1:21:53.66	37:45.12	large, *javac* (1.3.1), version error
T13	30329	8698	100137	30680	4219	1:32:06.50	37:23.57	large, *javac*, return in `finally` (bug id 4821353)
T14	30329	9120	100137	32431	4825	1:50:02.36	46:03.63	large, *javac*, 128 length array (bug id 4917091)
T15	30329	8415	100137	29489	3873	1:24:47.73	34:08.16	large, *javac*, doubly nested inner class (bug id 4903103)
T16	30329	8682	100137	30229	4371	1:27:49.16	38:36.88	large, *javac*, string reference error (bug id 5093115)
T17	35023	7789	113106	26161	2885	47:38.54	10:51.06	large, ChipperJ, null pointer error in if
T18	35023	9349	113106	35570	4410	1:35:37.74	32:18.08	large, ChipperJ, chipper v5 compiler error

(This best variant can, with additional techniques, be further chipped, as discussed in Section 5.1.3.)

T5-T8 show how well ChipperJ works on medium-sized programs: about 600–1200 lines of code. T5, T6 and T7 are student programming assignments of a simple language translator; T8 is a chipper for a simple, non-Java language. ChipperJ is able to reduce them to about 40-65% of the original based on a variety of different symptoms.

T9 is a very simple program. It consists of a single block of code containing about 1200 loops, with a nesting depth of at most two. The Compare script searches for the occurrence of the output string `2003` 95 times. This output is generated from multiple points throughout the code. This test represents a worst case example for ChipperJ; this relatively small and simple program takes longer to simplify than larger programs such as T10. Block splitting does not work here because output comes from many parts of the code. So, ChipperJ needs to walk through entire lists of statements and attempt to simplify each in turn. In addition, many of the statements are loops, so ChipperJ tries to apply the loop simplification to those (Sections 3.4.3 and 3.5.3).

T10 is a program that uses a user-defined Java library for string manipulation [18] with features inspired by strings in the Icon language [16]. The symptom is a specific NullPointerException due to a bug that we injected into the code.

T11-T16 demonstrate ChipperJ's effectiveness on large programs. We use versions of Sun's Java Compiler *javac* (version 1.4.2, unless otherwise noted). In T11, *javac* is set to compile a file where the filename does not match the class name. The symptom for T12 occurs when trying to run the Java 1.3.1 Compiler in an environment where a more recent compiler has previously been installed. This produces a version error when trying to access Object.class.

The symptoms for T13-T16 are all registered bugs for the Java 1.4.2 Compiler in Java bug site [31]; the bug ids appear in parentheses in the Table. In T13, *javac* issues a warning that a `finally` clause cannot complete normally if it contains a `return` statement. In T14, the symptom is that *javac* issues an error for an array over 128 dimensions whereas the JVM specifications allows 255 [24]. The Compare script for this test specifies the error string “array type has too many dimensions”.

In all these tests with *javac* (i.e., T11-T16), ChipperJ is able to reduce the number of lines by roughly 70-77%.

The final two tests, T17 and T18, are tests where we use ChipperJ to chip itself. The results are in the same ranges as for chipping just *javac*.

ChipperJ's disk space usage depends on the number of variants and the size of each variant and its output. In our tests, T16, one of the tests involving *javac*, used the most disk space: 127MB. Although that is a good amount of disk space, it is not very much given the size of disks on current systems. ChipperJ can be run so as to use less disk space. As noted in Section 3.7, in internal mode, ChipperJ does not need to actually write the variants to disk. Also, ChipperJ can be told to retain on disk all variants, only the good variants, or only the best variant.

ChipperJ took us about five person-months to develop (design, debug, test). As noted in Section 3, ChipperJ is built on top of *javac* and uses, with small modifications, *javac*'s code that builds and walks parse trees. The additional code for ChipperJ totals about 5000 lines.

Chapter 5

Discussion

We used the results from chipping on the array of programs and symptoms as described in Chapter 4 to increase our understanding of how the chipper could be improved and what its limitations were. We developed (and are in the process of developing) and tested several refinements (Section 5.1) in hopes of producing more simplified best variants for all of our programs. In addition, we discuss the limitations (Section 5.2) of ChipperJ in its current form and how they might be overcome.

5.1 Refinements

This section discusses in detail the three refinements we made to ChipperJ, and one promising refinement that is still in production (Section 5.1.4). Initially, the hope for each of refinement was that it would noticeably improve the performance of the chipper by either reducing the size (in number of parse tree nodes) of the best variant or by reducing the number of variants that are created en route to finding the best variant. Each of the refinements was tested using the collection of programs described in Chapter 4 with mixed success.

```
1  methA(){
2    int x;
3    x = methB();
4    if (x == 30)
5      ...
6    else if ( x > 30)
7      ...
8    else
9      ...
10   System.out.println("Yes");
11 }
12 int methB(){
13   ...
14   return 30;
15 }
```

Figure 5.1: Example for direction of chipping

5.1.1 Chipping Direction

The two `for` loops on lines 2-8 and 9-11 respectively of Algorithm 1 in Section 3.5.1 iterate from the first to the last statement within the block. Our experience has shown, however, that the chipper can generate smaller best variants if it works in the reverse direction For example, consider the code fragment in Figure 5.1 for the symptom of `Yes` in the output. Chipping of the body of `methA` proceeds as follows:

- block splitting. The block has four statements. (Recall that a sequence of `if-else if-else if ... -else` statements is treated together as a single statement.)
  - removing lines 4-10 fails – the statement at line 10 outputs symptom and is required to produce a good variant.
  - removing lines 2-3 fails – `x` is used later in the program. The variant does not compile because the removal of the first half of the variant removes `x's` declaration. The variable must be declared in order for the variant to compile.

Thus, block splitting fails at producing a good variant. The chipper restores the entire block before moving on to the next step

- removing each statement in turn, from top to bottom
  - removing line 2 fails – `x` needs to be declared because it is used in later statements
  - removing line 3 fails – `x` needs to be initialized in order for it to be included in the expression of the `if` statement in order to avoid an error from the Java Compiler
  - removing the `if` statement on lines 4-9 succeeds
  - removing line 10 fails – This statement outputs the symptom.

  All that remains in the block are the statements on lines 2, 3, and 10.

- simplifying each remaining statement from top to bottom
  - there is not a simplification technique defined for any of the remaining statements (variable declaration, variable assignment, or statement), so the chipper performs no simplification.

Thus, the best variant, using forward simplification, consists of lines 2-3 and 10.

However, if we reverse the order so that the chipper iterates from bottom to top instead of top to bottom, chipping of the body of `methA` proceeds as follows:

- block splitting.
  - block splitting still produces no good variant.
- simplifying each statement in turn, from bottom to top
  - removing line 10 fails – it outputs the symptom.
  - removing the `if` statement on lines 4-9 succeeds
  - removing line 3 succeeds – there are no more dependencies on the value of variable `x`

  - removing line 2 succeeds – `x` is no longer used in the method

  Thus, the only remaining statement is on line 10

- simplifying each remaining statement from bottom to top
  - there is not a simplification technique defined for an output statement so the chipper performs no simplification

Thus, by reversing the order of simplification, the best variant consists only of line 10. This example exhibits a key point that can be used during the chipping process: variable dependencies flow downward from variable declaration to variable utilization. Removing the variable declaration from a block where that same variable is used may result in a semantic error found by the Java Compiler should try to remove the statements where a variable is used before removing the declarations to avoid such semantic errors. According to Java semantics, a variable declaration must come before its use, so iterating through the block of statements in the reverse direction gives the chipper the best chance for removing the most statements.

This technique was applied to the test suite that we defined in Chapter 4. Table 5.1 details the results of reversing the chipping process for the collection of tests and symptoms as described in Section 4. The results show that reverse chipping can, in fact, be useful. In the case of T18, the best variant size was reduced by 2747 nodes, the highest overall reduction in the collection of programs that were used to test ChipperJ (Chapter 4). In addition, test T4, shows the highest percentage difference between forward and reverse chipping. This small test is similar in composition to the program in Figure 5.1, which undoubtedly explains why it performs so well. It is important to note, however, that in most cases the chipper did not preform well. In most other cases, the difference was between 0% and 3%. Even in tests T11-T16, the chipper is able to remove hundreds of extra nodes, that difference is only a small fraction of the size of the best variants, which are in the tens of thousands of nodes.

Table 5.1: Results for reversing the chipping direction

	parse tree nodes			forward - reverse	
test	original number	forward best	reverse best	node difference	percentage difference
T1	101	11	11	0	0%
T2	62	18	18	0	0%
T3	313	37	37	0	0%
T4	56	28	16	12	42.85%
T5	2112	1094	1094	0	0%
T6	2112	1116	1091	25	2.24%
T7	1880	485	477	8	1.65%
T8	5939	1949	1940	9	0.46%
T9	26435	3999	3999	0	0%
T10	11899	3332	3332	0	0%
T11	100137	26892	26202	690	2.57%
T12	80819	19978	19533	445	2.28%
T13	100137	30680	30267	413	1.35%
T14	100137	32431	31947	484	1.49%
T15	100137	29489	28906	583	1.98%
T16	100137	30229	29641	651	2.15%
T17	113106	26161	26128	33	0.01%
T18	113106	35570	32823	2747	7.72%

```
methA(){
  int x = -1;
  Object y = new Object();
  int z = 75;
  System.out.println("Yes " + methB(x));
  String a = y.getClass().getName();
  if ( z > 30)
    ...
  else if ( z > 30)
   ...
  else
   ...
}
int methB(int j){
  ...
  if (j < 0)
   return 30;
  ...
}
```

Figure 5.2: Example for statement shielding

5.1.2 Statement Shielding

In our original approach, every statement is subject to being chipped away. This includes variable declarations. However, our experience has shown that chipping away variable declarations can cause the chipper to not only find larger best variants, but to create a larger number of variants in order to get to the best variant. Finding the best variant earlier during block simplification can reduce the overall number of variants that the chipper has to produce, which has the potential to reducing the overall time of the chipping process. In statement shielding, the chipper shields the variable declarations from being removed during block simplification. The hope is to eliminate (or at least greatly reduce) the errors that come from missing variable declarations during block splitting process and in doing so find a better best variant earlier. For example, consider the code fragment in Figure 5.2 for the symptom of `Yes 30` in the output. Chipping of the body of `methA` proceeds as follows:

- block splitting. The block has six statements. (Recall that a sequence of `if-else if-else if ... -else` statements is treated together as a single statement.)

  - Removing the second half of the block (lines 5-12) produces a bad variant – the symptom is not exhibited in the first half of `methA`
  - Removing the first half of the block (lines 2-4) and restoring the second half of the block also produces a bad variant. This variant does not compile because the removal of the first half of the variant removes the declaration for variables `x`, `y`, and `z`. It is required that a variable be declared before being used in order for the variant to compile.

  Thus, block splitting fails at producing a good variant. The chipper restores the entire block before moving on to the next step

- removing each statement in turn, from top to bottom
  - removing line 2 produces a bad variant. `x` needs to be declared because it is used in later statements
  - removing the declaration of the variable `y` on line 3 also produces a bad variant
  - removing the declaration of the variable `z` on line 4 also produces a bad variant
  - removing line 5 produces a bad variant. This statement outputs the symptom.
  - removing the statement on line 6 produces a good variant.
  - removing the `if` statement on lines 7-12 produces a good variant

  All that remains in the block are the statements on lines 1-4.

- Simplify each remaining statement
  - there is not a simplification technique defined for any of the types of the remaining statements (variable declaration or output statement), so the chipper performs no further simplification.

Thus, the best variant, without using statement shielding, consists of lines 1-4. The chipper produced eight variants en route to this best variant.

If the chipper instead shields the variable declaration statements from being chipped during the block splitting and statement removal phase, chipping of the body of `methA` proceeds as follows:

- block splitting.
  - Removing the second half of the block (lines 5-12) produces a bad variant – the symptom is not exhibited in the first half of `methA`
  - Before removing the first half of the block and restoring the second half, the chipper shields the variable declarations on lines 1-3 so that they are not removed. Thus, the variant tested consists of the shielded statements on lines 1-3 and the block statements on lines 4-18. This produces a good variant.
  - the chipper recurses on the block splitting algorithm and takes the first half of the second block (line 4) along with the shielded statements (lines 1-3) to produce another good variant.

  The chipper determines that the block is too small to split upon again, thus it continues to the next part of the block simplification.

- removing each statement in turn, from top to bottom
  - the chipper tries to remove line 2, but this results in a bad variant.
  - the chipper removes line 3 producing a good variant because the symptom does not depend on variable `y`.
  - the chipper removes line 4 producing another good variant because the symptom does not depend on variable `z`.
  - the chipper tries to remove line 4, but this results in a bad variant because line 4 outputs the symptom.

  All that remains in the block are the statements on lines 1 and 4.

- simplifying each remaining statement from top to bottom
  - there is not a simplification technique defined for any of the types the remaining statements (variable declaration or output statement), so the chipper performs no simplification.

Thus, using statement shielding, the chipper is able to produce a smaller best variant, while creating a total of seven variants to do so. This example shows that statement shielding can be effective for reducing the best variant size and the number of variants produced en route to finding the best variant.

Unfortunately, the effectiveness of this approach can be described as hit and miss. It largely depends on the coding style used in the program being chipped. The example in Figure 5.2 favored statement shielding because the variable was declared and initialized in the same statement (line 2). If, however, that statement was split into two statements:

```
 2    int x; x = -1;,
```

keeping the declaration on line 2 along with the second half of the block will now produce a bad variant. In this case, the variable `x` is not initialized before being used — a semantic error in Java. Thus, the chipper would not encounter a good variant during the block splitting step during block simplification. Note that it will still find the same best variant found while chipping without statement shielding, but in this case statement shielding does not produce better results.

In addition, experience has shown that the statement shielding can even produce larger best variants by keeping extra variable declarations that do not contribute to the symptom and may also produce a larger number of variants during simplification.

These are the main factors that keep statement shielding from performing as well as originally hoped. ChipperJ with statement shielding was used to chip the collection of programs as described in Chapter 4. As indicated, the improvement was marginal at best. In general, statement shielding produced best variants within 0.5% of the size (larger and smaller) of the best variants produced when chipping without statement shielding. For tests T12, T13, and T18, however, shielding performed a little better, producing best variants that were 1.44%, 1.16%, and 1.63% smaller respectively. The overall number of variants produced ranged from 2% more to 0.4% less than the number of variants produced when chipping without statement shielding. At its best, statement shielding produced a 0.47% and a 0.52% reduction in number of variants T14 and T16 respectively.

Chapter 7 introduces the concept of developer guided simplification. Presumably, the developer would be able to tell the chipper which syntactic structures to focus on during simplification and which structures to leave untouched. Although shielding has not performed as well as hoped, techniques used in the statement shielding framework will facilitate the development of this new direction in chipping.

5.1.3 Order of File Chipping, Rechipping

ChipperJ performs chipping on files in the same order as their filenames appear on its command line. In some cases, removing parts of class A (say part of a method body) can have an adverse effect on the code in class B, although unrelated to the symptom. So, the best variant for A will need to retain those parts. However, if class B were chipped first, then parts of A can be removed and smaller variants obtained.

Unfortunately, determining class interdependencies of this nature would require a significant amount of analysis. Instead, we use *rechipping*. With rechipping, ChipperJ finds the best variants of all classes, as it normally does, but then it chips them again, starting with the first. Chipping continues until no best variant changes, i.e., a fixed point is reached. (Rechipping terminates because variants can never increase in size.) Figure 5.3 shows how rechipping works. Figure 5.3(a) shows the original code of two classes. Notice that *class B* makes a call to the *Ameth* inside of *class A*. Assume that the symptom comes from the printing of the word "symptom" from within *Bmeth*. The first pass of ChipperJ produces the simplified classes in Figure 5.3(b). The symptom does not depend on *class A*, and yet there are still references to *class A* after the simplification. Using rechipping, the chipper can reduce this example to the Figure 5.3(c), which is more simplified than the result of the first pass of the chipper.

We have experimented with rechipping on the collection tests in Chapter 4. The successive passes of the chipper are faster than the original pass because a variant is

```
public class A{         public class A{          public class A{
...                     ...                      ...
 int Ameth(){            int Ameth(){             int Ameth(){
  ...                     ...
  return 10;              return 10;               return 0;
 }                       }                        }
...                     ...                      ...
}                       }                        }

----------------------- ------------------------ -----------------------

public class B{         public class B{          public class B{
...                     ...                      ...
 void Bmeth(){           void Bmeth(){            void Bmeth(){
  A a = new B();          A a = new A();
  int i = a.Bmeth;        int i = a.Bmeth;
  if(i >0)
   out.print("symptom");   out.print("symptom");    out.print("symptom");
  else
   out.print(i);
 }                       }                        }
...                     ...                      ...
}                       }                        }
```

(a) (b) (c)

Figure 5.3: Rechipping example

always smaller than the original code from which it is derived, and the chipper only rechips the best variant of a program. For the tests in Table 4.1, rechipping requires from 2-6 passes and 20-400% additional time before converging, and generates roughly 2-3 times more variants. The sizes of the best variants are reduced by 10-15% in most cases, although by about 75% in one case (T4). As a typical example, rechipping T2 removed declarations for variables that are no longer needed after the initial chipping. As the atypical example, rechipping T4, which contained two classes in separate files that are dependent on each other, removed code no longer needed after initial chipping removed the dependency. We plan to further explore and evaluate rechipping in future work.

5.1.4 Chipping with Multiple Threads

ChipperJ's largest bottleneck occurs when the chipper compiles and executes the variant and checks the output of the variant for the symptom. These steps are facilitated by the Run and Compare scripts (Sections 3.1.1 and 3.1.2). This entire process is done serially: each step in the process is run to completion before the next step starts.

Obviously, a variant must be compiled before it can be ran. However, while a variant is running, it is generating output. As soon as the symptomatic output is generated, the chipper has no need to wait for the variant to run to completion. However, the variant may produce more output after the symptom is generated. In a worst case, the variant generates the symptom early on but, because of an infinite loop, must run until the timeout program stops the variant's execution before the output can be checked for the symptom.

If variant execution and the check for the symptom were to occur concurrently, it could help reduce the bottleneck.

ChipperJ can be implemented with two threads that control the compilation and run and the check for the symptom respectively:

- RunThread – Responsible for starting and monitoring the output of the Run script.
- CompareThread – Responsible for starting and monitoring the output of the Compare script

Using synchronized methods to control access to its internal variables, the chipper is able to signal to the threads when to proceed and when to wait, the CompareThread is able to signal the RunThread that the symptom was found, and the RunThread can signal to the CompareThread that the variant has run to completion.

When ChipperJ starts, both threads wait for the program to be parsed and for a variant to be generated. After the chipper creates the variant, it signals to the threads that they can proceed. The RunThread starts by running the Run script, which compiles and runs the variant. Meanwhile, the CompareThread runs the Compare script, which is written to check the output of the variant for the first instance of the symptom. If the variant is run to completion, the RunThread signals this to the CompareThread whereupon the CompareThread stops the Compare script and indicates the bad status of the variant to the chipper. If, at any point before the variant runs to completion, the Compare script finds the symptom in the output, the CompareThread signals to the RunThread that the variant is good, whereupon the RunThread can kill the run of the variant. In addition, the CompareThread signals to the chipper that the variant is good, allowing the chipper to proceed in creating the next variant.

Using multiple threads to speed up the overall chipping process showed much promise in the implementation of a prototype chipper, however, it has yet to be implemented for ChipperJ.

5.2 Limitations

5.2.1 Variant Size

The data in Section 4 show that ChipperJ is reasonably effective in reducing the original program. However, ChipperJ is not guaranteed to find the minimal variant. For example, ChipperJ might split block B into two halves, $B1$ and $B2$. It first tests whether $B1$ exhibits the symptom. If so, ChipperJ simply discards, without any further consideration, $B2$. It is possible that $B2$ also exhibits the symptom and $B2$ would lead to a smaller best variant than $B1$ does. ChipperJ uses this simple, but not optimal, strategy so as to limit the number of variants it needs to generate and test.

The results in Section 4 show that ChipperJ is effective in substantially reducing the size of the input code. In our experience, larger blocks that contribute to the symptom in multiple places tend to increase the number of variants, and the time required to find the minimal variant. For the larger programs, such as the Java Compiler, a reduction of 75% is significant (i.e., T11-T16). However, 7000-8000 lines of code spread across 66 files is still a great deal to analyze when trying to isolate

```
 1  methD(){
 2    C c = null;
 3    c = new C(...);
 .    ...
 .
 .
90    c.f(...);
91  }
```

Figure 5.4: Example for null pointer exception

the cause of a single bug. Our current approach is to take the best variant and use a manual Java debugger (e.g., the debugger within Eclipse [14]) to step through the code to see how the error is produced. Then, putting the chipped code side by side with the original code, can give a good idea as to the location of the bug within the original program.

5.2.2 Symptom Specificity

Recall from Section 3 that typical symptoms include errors in output, infinite loops, exceptions, etc. Because Compare is a shell script, it can be as complex as desired. Most of our scripts were simple and used a few standard UNIX tools like *grep* and *cmp*, but a few of our Compare scripts also invoked other programs. So, the Compare script can also check whether the program, say, modified a particular file or had a memory leak (by searching for an error message or unhandled exception in the program's output). Similarly, the Run script can include any list of commands.

For similar reasons, ChipperJ might find an unexpected variant that exhibits the symptom. For example, suppose the Compare script specifies to search the output for "NullPointerException". ChipperJ might find a good variant that causes such behavior for reasons different from those for the original program. For example, consider the code in Figure 5.4. ChipperJ might remove lines 3-89 and the variant would still exhibit the NullPointerException symptom. Unfortunately, that sheds no light on the immediate cause of the problem, which presumably is somewhere in lines

4-89. (However, ChipperJ would also remove code outside of the method; thus, it would likely be more apparent under what conditions the method was invoked.) In such a case, the user could specify a more specific symptom. For example, if the program does some output on line 80, part of that output could be included as a conjunct in the symptom; if the program does no output between lines 4-89, the user could add some. Or, the user can include in the symptom for an exception several line numbers from its stack trace. We are also considering adding an option to ChipperJ to facilitate such cases. ChipperJ can easily instrument the variant programs with statements that output "reached line X in file F" to standard output or a log file and then such output could be included as part of the symptom in the Compare script.

5.2.3 Undesirable Side Effects

A variant program might have a harmful side effect that the original program does not. For example, a variant might execute code that removes a file whereas the original program did not execute that code at all. A related problem is that a variant can affect the environment in which subsequent variants run. Again suppose that a variant deletes a file. That file might be needed for subsequent variants to run properly so those variants would (most likely) be deemed to be bad and the approach would not yield a reasonable-sized best variant. The situation here, though, is no worse than analogous situations in mutation testing (e.g., [12, 26]) or security testing where suspicious programs are executed in a cleanroom environment.

5.2.4 Nondeterminism

ChipperJ will not work with programs whose output is nondeterministic, as is often the case for multi-threaded Java programs. If a particular variant nondeterministically displays the symptom, then it might be determined as being a bad variant, thus not allowing ChipperJ to examine variants that result from the rejected variant.

Program replay techniques (e.g., [23, 27]) might be used to force deterministic results. However, that would require substantial additional implementation effort. In some cases, the user might be able to specify deterministic symptoms for nondeterministic behaviors.

5.2.5 Program Input

Consider a program that contains two loops, each with a read statement. If ChipperJ eliminates the first loop, then the second loop is likely to read the wrong values from the input stream (i.e., those values intended for the first loop).

ChipperJ does nothing special for programs with input. It simply generate variants. If ChipperJ eliminates a critical read statement, then the variant will be found to be bad. Our experimentation show that this approach works reasonably well. If the symptom does not depend on input data, ChipperJ eliminates the reads. If input is being read in two loops and the symptom depends on data only read by the first loop, then ChipperJ eliminates the second loop. On the other hand, if the symptom depends on data only read by the second loop, then ChipperJ does not eliminate the first loop. Instead, ChipperJ simplifies the first loop to just the loop control and the read statement; i.e., it eliminates all but what the program needs to properly advance the input stream.

More sophisticated approaches are possible, and worth exploring. For example, we can instrument the original program to record what input was read by each read statement in the original program. To ensure that the variant preserves the input behavior of the original program, the variant must read in order a *subsequence* of the recorded input. This approach is simpler than that needed for replaying input for concurrent programs (e.g., [28] and [11]), but similar to that for sequential programs (e.g., [29]).

ChipperJ does not work with programs that require user input in the form of mouse clicks, the filling in of textboxes, or some other sequence of specific input events that cannot be properly run using only the Run script. ChipperJ would need a way to store and then reuse the input sequence. For such purposes, ChipperJ could be integrated with a tool like the XLAB capture/replay tool [34].

Chapter 6

Related Work

Reference [9] introduces the general notion of data slicing, on which our work is based. Data slicing aims to help a programmer locate a program bug by reducing the size of the data on which the program exhibits the bug. Our work applies data slicing to an entirely different domain, i.e., programs are our data. Also, we have actually built ChipperJ, an *automated* data slicing tool; Reference [9] presents general techniques for data slicing and suggests building such tools in general, but the authors apparently did not pursue that idea.

Delta debugging [37] can be viewed as a form of data slicing. Program chipping and delta debugging (and data slicing in general [9]) share some common techniques, such as using binary search (block splitting in ChipperJ) to reduce the size of the input data. The general idea is to use a tool that takes the input to a program and uses binary search to isolate the part of the input that is causing a bad output. Delta debugging is quite effective, as seen in the examples in Reference [37], such as automatically reducing the size of HTML input to a buggy version of Mozilla. This particular example exhibits the power of the delta-debugging technique because Mozilla is well equipped to handle input that deviates from HTML syntax, which is significant because many of the variants of the input do not conform to the HTML syntax.

Delta debugging, though, would not work so well on more structured input, such as Java programs. It is unable to use the syntactic structure of the input data (Java programs) as ChipperJ does and would therefore produce too many variants than can be reasonably tested. To confirm this assertion, we created a "delta chipper", which uses the generic delta debugging algorithm to chip away at Java programs (in external mode only). Table 6.1 summarizes the results of running this delta chipper on a few of the tests in Section 4.[1] As expected, compared to ChipperJ, the delta chipper

Table 6.1: Delta chipping results (external mode only) on a few tests from Table 4.1

	source lines		parse tree nodes		# of	time
test	original	best	original	best	variants	(H:MM:SS.00)
T1	21	19	109	95	23762	3:25:35.51
T5	596	?	2112	?	> 85000	> 24:00:00.00
T8	1767	?	5939	?	> 78408	> 24:00:00.00

generated larger best variants, generated many more variants (most of which had compile time errors because delta chipping removes individual characters), and took considerably more time. T5 and T8 did not complete within a day, so we terminated them; they were less than half complete.

A variant of Delta Debugging, Hierarchical Delta Debugging [25] (HDD), can also be viewed as a form of data slicing. Instead of debugging the input on a character by character basis, this spin-off applies Delta Debugging algorithm to the parse tree allowing it to take advantage of the syntax of the input. Using a top down approach, the Delta Debugging algorithm is applied to all of the nodes in the same level in the parse tree's hierarchy. The removal of a node at any level of a tree also removes that node's children, so the algorithm runs to completion at each level of the parse tree; the more nodes that are removed early in the process, the less children nodes there will be in lower levels of the hierarchy. This approach is similar to chipping because it too uses knowledge of syntax to trim the parse tree. One of the main differences is that it strictly uses the Delta Debugging algorithm for simplifications whereas chipping allows for a greater variety of strategies to be applied at various nodes in the parse

[1] The delta chipper actually removed the newline characters leaving only one line of code. But, to allow comparisons with Table 4.1, Table 6.1 shows the number of lines that would have remained had the newline characters not been removed.

tree according to node type.

Unfortunately, there is no direct performance comparison between program chipping and HDD. HDD operates on the C language attempting to minimize a C program that causes a failure in gcc, while program chipping focuses on minimizing symptomatic Java programs. For the C programs that were tested, HDD was able to reduce their size (in bytes) by 55% to 87%. For the Java programs that are chipped as described in Chapter 4, ChipperJ is able to reduce their size by 30% to 87% and has the added bonus that it is able to accept programs that span multiple files.

As noted earlier, our work is related to program slicing. Program slicing was introduced in Reference [35]. Much work on slicing has followed, e.g., as surveyed in Reference [32]; as exemplified in specific variations of slicing such as described in Reference [33], and including interprocedural slicing such as described in References [21] and [4] and program dicing, introduced in Reference [36] and developed further since then, e.g., in Reference [10]. Reference [6] introduces dynamic slicing, which is based on a specific input for the program being sliced. In that regard, our work is closer to dynamic slicing. However, with slicing, the user looks for behavior with respect to a variable or group of variables, and slicing tools analyze the code and build representations such as program dependence and control dependence graphs. Whereas with chipping, the user looks for behavior with respect to the overall program behavior (thus, chipping treats the program as a "black box") and chipping tools use only the program's parse tree and relatively simple simplification techniques. Moreover, program slicing is used for a variety of applications, e.g., differencing [20], whereas program chipping appears useful only for bug isolation. Perhaps our work is in line with developing less costly forms of slicing [17], albeit for the fairly specific problem domain of bug isolation. Despite these differences, it would be interesting to compare ChipperJ and more traditional slicing techniques and tools to see their relative effectiveness in terms of size of variant or slice, cost of finding variant or slice, and the complexity of techniques, tools, and use. We did attempt to do so with the Bandera Slicer [8]. Unfortunately, the bugs in the current version of the Bandera Slicer prevented us from being able to run and do these comparisons, despite our best efforts.

Our work relates to previous research in debugging. Spyder [5] is the first debugger to integrate dynamic program slicing with debugging. More recently, xSlice [7] integrates execution slicing with debugging. Other techniques and tools (e.g., Tarantula [22]) use visual information to assist fault localization and provide automated ways to "narrow the search by selecting suspicious statements that might contain faults" [13]. Our work also has a weaker connection to algorithmic testing [15], which slices out irrelevant procedure calls, but requires interaction with the user and operates at the program variable level. Our work has a similar general goal to all these previous works, but it uses entirely different techniques, as noted earlier.

Our work has some connection to work in mutation testing, which was introduced in Reference [12]. Work on mutation testing has advanced considerably, e.g., as described in Reference [26]; some work [19] uses slicing to aid in mutation. Program chipping generates "variants", which are similar to "mutants": both involve some modification to the original program. In fact, "bad variants" are similar to "dead mutants": the former does not exhibit the symptom of interest and the latter does not match a test's correct output; both are dropped from further consideration. However, program chipping is aimed at isolating symptoms by reducing program size versus finding appropriate test cases or analyzing test case coverage by mutating the source code (usually via, for example, changing specific operators).

Chapter 7

Conclusion

This thesis has introduced the idea of program chipping and described a chipper for Java programs. The results are encouraging, but we need to gain more experience in using ChipperJ in real practice.

Our experience described in this thesis represents a first step in program chipping and suggests much future work. We plan to develop further simplifications for the current ChipperJ, e.g., for expressions, method invocation, and eliminating entire methods or classes. We also plan to further investigate the efficacy of some of our present techniques, such as rechipping (Section 5.1.3), and to implement a multi-threaded chipper (Section 5.1.4).

We will also consider how program chipping might be combined with other approaches, such as traditional program slicing or debuggers. For example, chipping might be used first and then more refined slicing applied to the best variant to reduce the program further. We will explore the tradeoffs in when and how the two techniques might be used together.

We also plan to develop a hybrid, interactive chipping tool. Using such a tool, a developer who has some idea (initially or as simplification proceeds) of where the symptom is occurring, will be able to guide simplification. The developer will be able to instruct the tool to focus on the particular section of code and be able to pick particular simplifications to apply while passing over other sections of code completely. The tool will be GUI-based and will allow the developer to easily see the differences between variants as they are generated. We will use the statement shielding framework as described in Section 5.1.2 to implement this feature.

Bibliography

[1] *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, White Plains, NY, June 20–22 1990.

[2] *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 26–28 1991.

[3] *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'01)*, Snowbird, Utah, USA, June 18–19 2001. ACM.

[4] G. Agrawal and L. Guo. Evaluating explicitly context-sensitive program slicing. In ACM [3], pages 6–12.

[5] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience*, 23(6):589–616, 1993.

[6] H. Agrawal and J. Horgan. Dynamic program slicing. In ACM [1], pages 246–256.

[7] Applied Research. xslice: A tool for program debugging. `http://xsuds.argreenhouse.com/html-man/xslice.html`.

[8] Bandera Project. The Bandera slicer. `http://bandera.projects.cis.ksu.edu/`.

[9] T. W. Chan and A. Lakhotia. Debugging program failure exhibited by voluminous data. *Software Maintenance: Research and Practice*, 10:111–150, 1998.

[10] T. Y. Chen and Y. Y. Cheung. On program dicing. *Journal of Software Maintenance*, 9(1):33–46, 1997.

[11] F. Cornelis. Linux Input Replay. `http://www.elis.rug.ac.be/~fcorneli/`.

[12] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Transactions on Computers*, 12(4):34–41, Apr. 1978.

[13] R. A. DeMillo, H. Pan, and E. H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of the 20th IEEE International Conference on Computer Software and Applications (COMPSAC 97)*, pages 515–521, Aug. 1997.

[14] eclipse.org. The Eclipse development environment. `http://www.eclipse.org`.

[15] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri. Generalized algorithmic debugging and testing. In ACM [2], pages 317–326.

[16] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996. `http://www.cs.arizona.edu/icon/lb3.htm`.

[17] W. G. Griswold. Making slicing practical: the final mile. In ACM [3].

[18] A. Habra. The General String library. `http://www.tek271.com/free/gsoverview.html`.

[19] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, Dec. 1999.

[20] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In ACM [1], pages 234–245.

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[22] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th IEEE International Conference on Software Engineering (ICSE-02)*, pages 467–477, Orlando, FL, May 2002.

[23] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[24] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Sun Microsystems, second edition, 2005. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

[25] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. Submitted.

[26] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty-First Centuries*, pages 45–55, San Jose, CA, Oct. 2000.

[27] R. A. Olsson. Reproducible execution of SR programs. *Concurrency—Practice and Experience*, 11(9):479–507, August 1999.

[28] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[29] SimCon. FPT – the Fortran Partner. `http://www.simcon.uk.com/`.

[30] SourceForge. *SRC Tkdiff*, 2004. `http://sourceforge.net/projects/tkdiff/`.

[31] Sun Microsystems. The Java bug database. `http://bugs.sun.com/bugdatabase/index.jsp`.

[32] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[33] G. A. Venkatesh. The semantic approach to program slicing. In ACM [2], pages 107–119.

[34] M. Vertes. XLAB: a tool to automate graphical user interfaces. *Linux Weekly News*, May 1998. `http://mvertes.free.fr/xlab/xlab.html`.

[35] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[36] M. Weiser and J. R. Lyle. Experiments on slicing-based debugging aids. In *Empirical Studies for Programmers*. Ablex Publishing Corporation, 1986.

[37] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.