SOFTWARE—PRACTICE AND EXPERIENCE

Softw. Pract. Exper. 2006; **99**:01–99 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Automated bug isolation via program chipping

Chad D. Sterling[†,‡] and Ronald A. Olsson[*,§]

Department of Computer Science, University of California, Davis, CA 95616-8562 USA

Note to editor and reviewers:

This paper is based on our earlier paper (Reference [32]), which appeared in the AADEBUG 2005 conference. The present paper extends that earlier paper with additional details about our overall approach, specific simplifications, further experimental results, and additional discussion.

Also, the paper is formatted using LaTex's `\baselinestretch{2}`, as per the instructions. Some of the figures and tables look a bit odd this way.

[†]Partially supported by a GEM fellowship via the National Consortium for Graduate Degrees for Minorities in Engineering and Science, Inc.
[‡]Current affiliation: Hewlett-Packard (San Diego, CA USA)
[*]Correspondence to: Ronald A. Olsson Department of Computer Science, University of California, Davis, One Shields Avenue, Davis, CA 95616-8562 USA
[§]E-mail: olsson@cs.ucdavis.edu
Contract/grant sponsor: National Science Foundation, equipment grant; contract/grant number: EIA-0224469

Received day Month year

SUMMARY

This paper introduces program chipping, a simple yet effective technique to isolate bugs. This technique automatically removes or *chips* away parts of a program so that the part that contributes to some symptomatic output becomes more apparent to the user. Program chipping is similar in spirit to traditional program slicing and debugging techniques, but chipping uses very simple techniques based on the syntactic structure of the program. We have developed a chipping tool for Java programs, 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; even if it took overnight, that would be reasonable if it saves the developer time.

KEY WORDS: Program chipping, debugging, bug isolation, data slicing, program slicing

> "A fool-proof method for sculpting an elephant: first, get a huge block of marble; then you chip away everything that doesn't look like an elephant." – Unknown

> "The Sculptor produces the beautiful statue by chipping away such parts of the marble block as are not needed — it is a process of elimination." – Elbert Hubbard (American editor, publisher and writer, 1856-1915)

1. INTRODUCTION

Software developers spend an enormous amount of their time in debugging and testing their products. When a user discovers a bug in a program, the user typically sends the entire program to the developer (or 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 isolating 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 user 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 require some manual intervention. For example, suppose that the bug is that the program loops infinitely. Typically, the developer would use a debugger to find where the program is looping and then slice the program on variables involved in that loop to reduce the size of the program.

This previous specific 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 then 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*.

We have developed a program chipper for Java, called ChipperJ. ChipperJ has shown promising results when chipping a variety of programs. The programs on which we have experimented range from small straightline programs to large programs such as the Java Compiler. The results show that the programs are generally reduced to roughly 20-35% of their original size. Depending on the size of the program and the complexity of the symptom, ChipperJ can take from seconds on small programs to just under an hour on large programs to perform this reduction.

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 [39] (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 allow us to produce automatically a reasonable number of reduced programs in a reasonable amount of time. Program chipping is also related to program slicing (see Section 6). However, with slicing, the user looks for behavior with respect to a variable or 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 paper is organized as follows. Section 2 presents a general overview of our approach and introduces terminology. Section 3 describes ChipperJ, our program chipper for Java. Section 4 discusses the effectiveness of ChipperJ over a range of small and large buggy programs with various symptoms. Section 5 outlines some of our design decisions and some limitations. Section 6 presents related work and compares chipping with delta debugging and program slicing. Finally, Section 7 describes future work and concludes the paper. This paper extends our initial paper on program chipping (Reference [32]). Reference [31] presents additional details.

2. OVERVIEW OF APPROACH

We use the following terminology:

- *symptom*: identifying characteristic in behavior of interest. It may be something in the program's output, something not in the program's output, that the program loops infinitely, etc. It may represent a bug.
- *variant*: program derived from the original program
- *good variant*: a variant that exhibits the symptom
- *bad variant*[†]: a variant that does not exhibit the symptom

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

Prepared using speauth.cls

- *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 (which may be the original program)

We use the term *smallest* with respect to some application specific measure, e.g., number of nodes in a parse tree or number of lines of source code.

As indicated in Section 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. 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 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

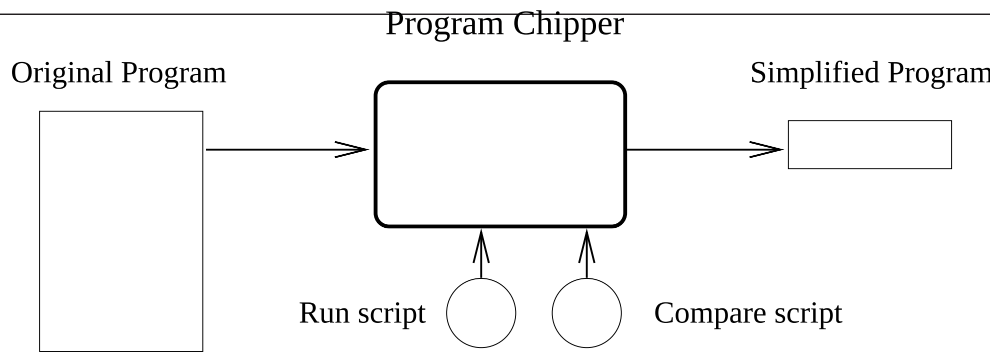
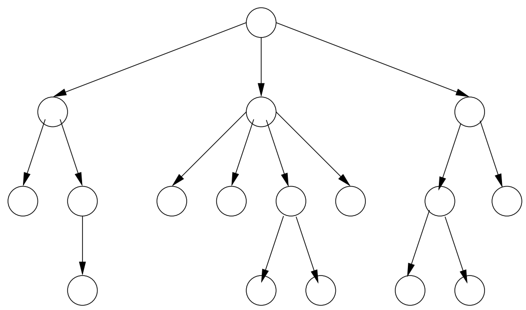


Figure 1. Overview of program chipping

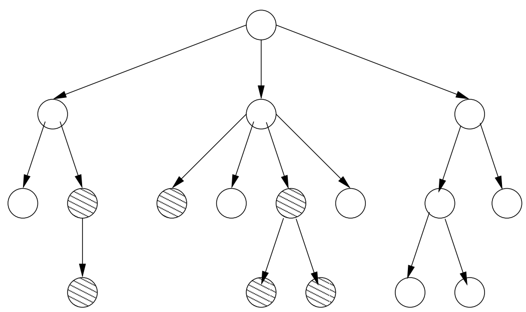
(e.g., shorter in length) than the original, so that finding the symptom in the remaining code will be easier.

In more detail, the program chipper builds a parse tree corresponding to original program, as shown in Figure 2(a). From this parse tree, the chipper tool deletes or modifies one or more nodes to generate other parse trees, e.g., Figure 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 the Run and Compare scripts). This process continues until the chipper has completed, having identified a best variant (parse tree), e.g., Figure 2(c).

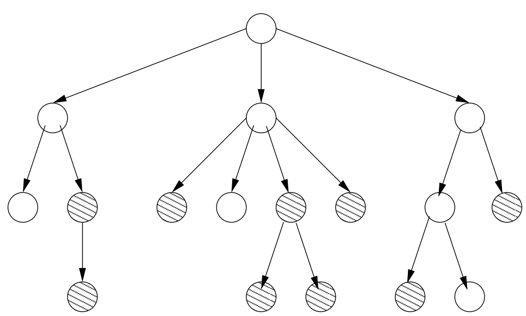
The prior description does not specify the exact process that the tool follows to reduce parse trees, and to determine when to terminate. One possibility is to generate and test all possible variants. Certainly, though, for any reasonably-sized program, that approach is impractical. The key, then, is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



(a) Parse tree for original program



(b) Parse tree for an intermediate variant



(c) Parse tree for best variant

Figure 2. Parse trees during program chipping. Shading indicates the original node has been deleted.

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 section describes the user view of ChipperJ, presents ChipperJ's simplification algorithm, introduces a GUI-based tool that is often used with ChipperJ, describes ChipperJ's specific simplifications, and discusses how ChipperJ deals with a key pragmatic issue: infinite loops.

3.1. User view

The user gives ChipperJ the original program, specifies how to run the program and how to check for the symptom, via the Run and Compare scripts, as shown in Figure 1. ChipperJ then produces the best 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 run the program and what constitutes the symptom.

A typical Run script compiles the variant program, runs it on some specified input, and saves the program's standard output to a file. Figure 3 shows a simple example of a Run script

```
#!/bin/sh
filePrefix=$1;  mainClass=$2;  outputFile=$3
(echo compiling the Variant; javac $filePrefix.java) \
&& (echo executing; java $mainClass > $outputFile 2>&1)
```

Figure 3. Example of a simple Run script

that takes three arguments: the prefix of the file to be compiled, the name of the program's main class, and the name of the output file in which to store the results of the variant's execution. It compiles the variant program by invoking the *javac* command and then executes the variant by invoking 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 3.5) and to allow for alternative methods of variant compilation (such as internal compilation discussed in Section 3.6).

A typical Compare script searches the program's output for the given symptom. Often these are just shells script that uses one or more *grep*s and other standard UNIX tools. 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 it chipped away.

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

Figure 4. Example of a simple Compare script

Figure 4 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 checked. In this case, the symptom is that the program being chipped outputs "This Is The Symptom." The script checks the output for a symptom by invoking the *grep* command to search for the desired string in the output file. The script uses its exit status to communicate to the chipper whether or not it found the symptom.

3.2. Simplifying transformations

ChipperJ follows the overall simplification approach shown in Figure 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. For each variant, ChipperJ checks whether the variant exhibits the same symptom as the original program. If it does, ChipperJ tries 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 by earlier simplifications 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.

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 simplification continues in this new good variant. If the variant is bad, then simplification 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; if no simplifications remain to be tried, simplification continues at the next node.

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, run, and compare the bad variants.

Figure 5 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 Section 3.4.

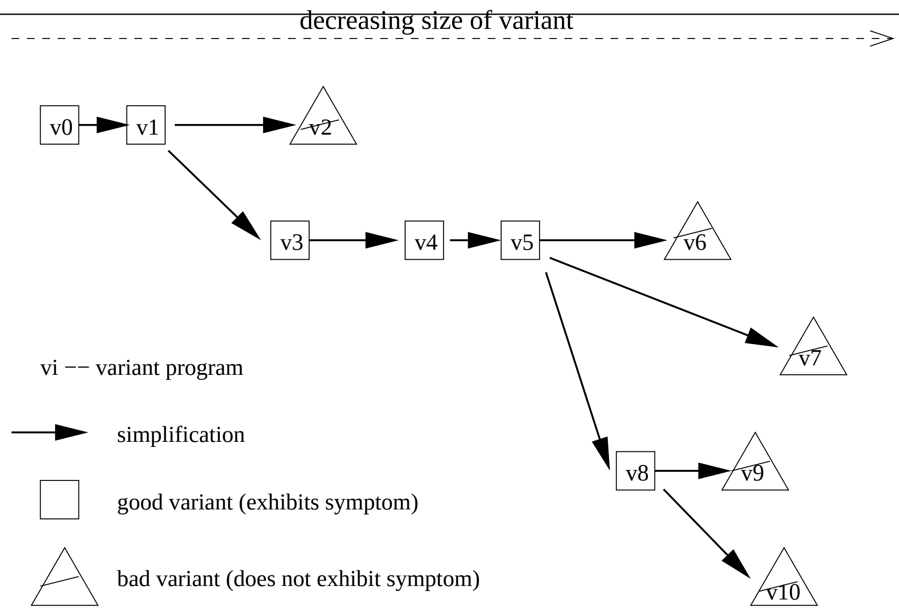


Figure 5. Sample ChipperJ simplification process

3.3. The *v-diff tool*

We also developed a GUI tool called *v-diff*. It displays a graphical representation of the variants created by ChipperJ, similar in spirit to that shown in Figure 5. The *v-diff* tool is useful to see the differences between any two variants or to watch in "movie mode" the differences between successive variants in a sequence. 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. It is especially helpful when looking at the sequence of good variants and not including the bad variants in-between them.

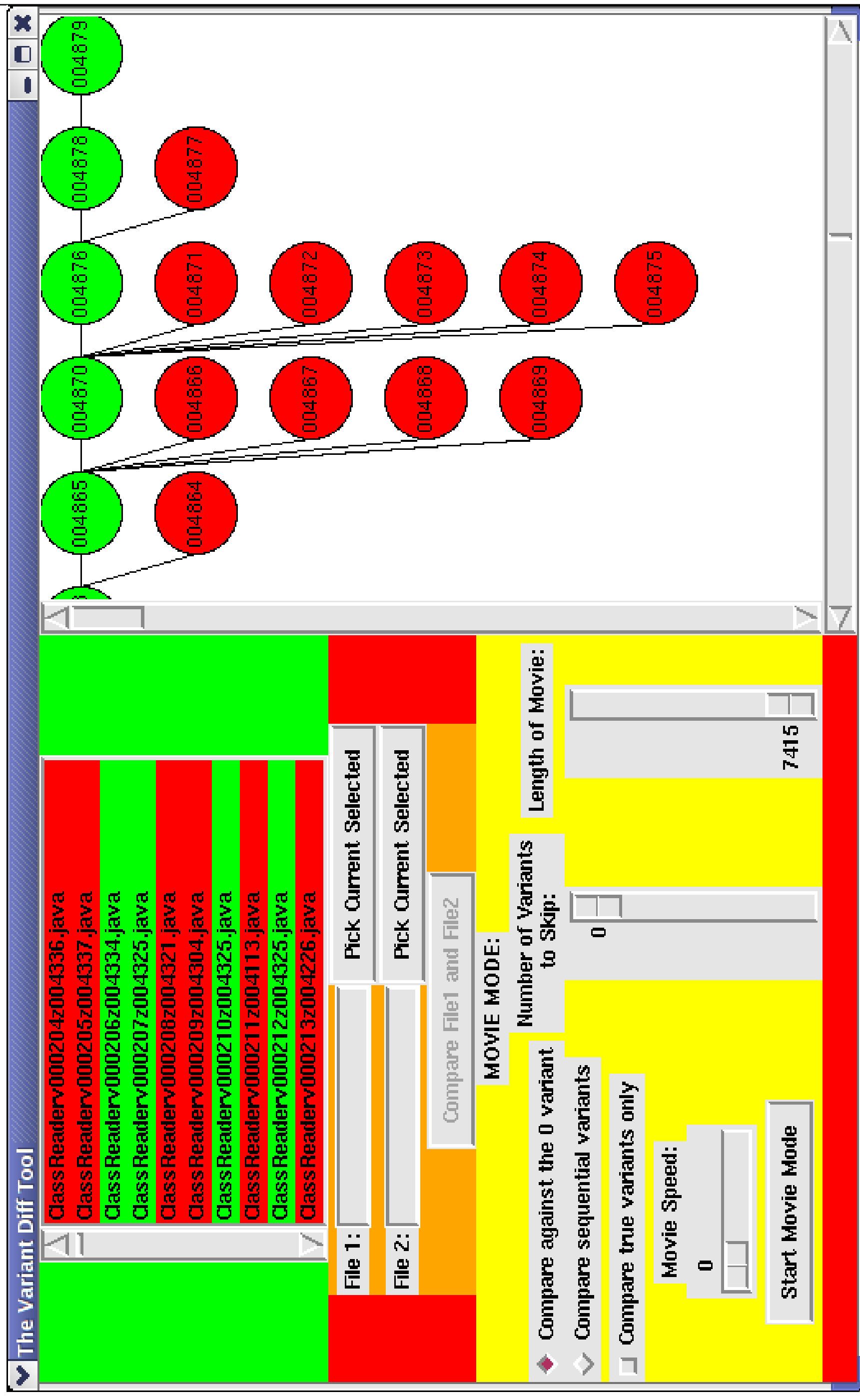


Figure 6. Screenshot of the *v-diff* tool

Figure 6[‡] shows a screenshot of *v-diff* in action. The right panel contains a scrollable, clickable canvas that shows the good variants (light shading) and bad variants (dark 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.

In movie mode, the variants are shown two at a time in a frame by frame succession. 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. It also allows the user to specify whether to automatically advance to the next comparison after a user-specified amount of time or to instead have the user manually interact when ready. 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.

The implementation of *v-diff* uses *tkdiff* [30] to show the differences in two variants. *tkdiff* is a graphical front end to the standard UNIX *diff* program. It provides a side-by-side view of two files with the differences highlighted.

[‡]Because this paper 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.

We originally developed *v-diff* to help us debug the chipper and ensure that the sequence of simplifications happened as expected. However, we have found it to be a very useful tool during the process of isolating the symptom as well.

3.4. Specific simplifications

3.4.1. Simplifications on blocks

ChipperJ tries, in the order given, the following simplifications on blocks.

- eliminate the entire block.
- "binary search", also known as *block splitting*. ChipperJ splits the statement list in block B into two halves, $B1$ and $B2$.[§] 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. Otherwise, ChipperJ tries this simplification on $B2$. If simplification fails on each of $B1$ and $B2$, then ChipperJ restores B into the parse tree.
- eliminate one statement at a time from the block.
- simplify each remaining statement. For example, if a while loop cannot be entirely eliminated (by the previous step), then ChipperJ will try to simplify (recursively) the body of the loop.

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

Table I. Type-appropriate replacement blocks ChipperJ generates

Return type of method	Type-appropriate replacement block
void	{ }
char	`{return ' ';}`
int or any numerical type	`{return 0;}`
boolean	`{return false;}`
any kind of Object or String	`{return null;}`

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, so ChipperJ would restore the method body and try the remaining simplifications. Although restoring the method body is not harmful to the simplification process, the size of the best variant ChipperJ finds will not be as small as it might be. So, instead of replacing the method body with an empty block, ChipperJ replaces it with a block containing a type-appropriate return statement as shown in Table I. ChipperJ also inserts such return statements in other cases, e.g., when it splits a method body into two blocks.

3.4.2. Simplifications on loops

ChipperJ tries to simplify any of the three kinds of loops in Java by replacing the loop by one iteration of the loop's body. For example, `while (x!=0) {S}` becomes just `{S}` and `for (int i = 0; i < N; i++) {S}` becomes `{int i = 0; {S}}`. (One can view this

Prepared using speauth.cls

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 loop structure will make it all the more evident to the user. ChipperJ continues simplification by simplifying the loop's body. Most often the body is a block, so ChipperJ proceeds with the block simplifications (Section 3.4.1).

3.4.3. Simplifications on if statements

ChipperJ treats a sequence of if-else if-else if ... -else statements together for simplification purposes.

ChipperJ first tries to replace the sequence with the body of one of its arms, one arm at a time. Figure 7 illustrates this simplification. Figure 7(a) is the original `if` statement. Figures 7(b)-(d) show the variants that ChipperJ creates when trying to replace the sequence of `if-else if... -else` statements with the body of one of the arms. If any one of these 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. However, if no body of these arms by itself produces a good variant, ChipperJ starts over with the sequence. It next tries, starting at the beginning of the sequence, to remove each arm in turn and retains only those arms that are needed in a good variant.

As usual, ChipperJ recursively simplifies any arms that still appear in the good variant resulting from these simplifications.

```
...
if (44==i){
 k=20-i;
 println(k);
}
else if (15==i){
 n=5;
 println(k*10);
}
else{
 k=10-i;
 println(k);
}
...
```

(a)

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

(b)

```
...
{
 n=5;
 println(k*10);
}
...
```

(c)

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

(d)

Figure 7. Example illustrating the simplification of an `if` statement: (a) shows the original code, (b-d) show the “one arm” variants the chipper tries

3.4.4. Simplifications on `switch` *statements*

The simplification for `switch` statements is a hybrid of the `if` simplification (Section 3.4.3) and the block simplification (Section 3.4.1).

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.

3.4.5. No other simplifications

ChipperJ does no other simplifications. It does *not* attempt to simplify assignment statements or expressions, discard entire methods (but it might discard their bodies, as noted in Section 3.4.1), etc.

3.5. 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 some of 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.

```
#!/bin/sh

# Run script

filePrefix=$1;  mainClass=$2;  outputFile=$3

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

```
#!/bin/sh

# runHelper script

filePrefix=$1;  mainClass=$2

(echo compiling the Variant; javac $filePrefix.java) \

&& (echo executing; java $mainClass)
```

Figure 8. Example of a Run script using a timeout program and a helper script

ChipperJ solves this problem simply: the Run scripts use a timeout program to limit the variant's execution time. (The specific timeout interval is specified by the user based on the time taken by original program.) 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. (The Compare script can specify "not looping" behavior by using *grep -v* for the special message.)

For example, the simple Run script in Figure 3 has no way to stop a variant that contains an infinite loop. For such a variant, the chipper would wait forever for the variant to finish running. Figure 8 shows an example of the Run script using a timeout program. The usage for the timeout program is:

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

This example actually uses two scripts: "Run" to start up the timeout program and a helper script named "runHelper" This Run script takes the same three arguments as the Run script in Figure 3. The Run script starts the timeout program on the runHelper script and allows it to run for at most five seconds. The runHelper script is responsible for compiling and running the variant; if the variant has not been compiled and run to completion in five seconds, the timeout program will kill the variant and allow the chipper to proceed. If the variant has exhibited the symptom, the chipper will consider it good, regardless of whether or not the variant was killed by the timeout program; otherwise, the chipper will consider the variant bad.

3.6. External or internal chipping

ChipperJ can operate in one of two modes: external or internal. In external mode, ChipperJ writes each variant it produces from the parse tree to a file and then compiles and runs that variant according to the Run script and looks for the symptom according to the Compare script. 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 internal mode, since compilation can modify the parse tree, ChipperJ takes care to make copies of any parse subtree that it might need later to restore the current variant if the new variant is bad (Reference [31] gives details). Internal mode, of course, can be substantially faster, as will be seen in Section 4.

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

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).[‖] The results are averages of several test runs, which had insignificant variances.

Table II 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 II represents non-blank, non-comment lines. The data in Table II show that external chipping takes about from 2-5 times as long as internal chipping (Section 3.6).

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

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

Table II. ChipperJ results

	source lines		parse tree nodes		# of	time (H:MM:SS.00)		
test	original	best	original	best	variants	external	internal	notes
T1	21	5	109	13	11	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

one of the classes (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 "2005" 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 (Section 3.4.2).

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 [33]; 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*.

Our current approach for actually finding the source of the symptom 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 symptom 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. Using this method, we were able to use the best variant of T11 to trace the symptom to lines 511-514 of the file Enter.java. In another example, the best variant of T12 was used to trace the symptom to lines 844-845 of the file ClassReader.java where a hardcoded value will not allow the Java 1.3 compiler to use classes from a more recent version of Java. In these two examples, the source of the symptom was tracked down in less than ten minutes.

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 3.6, 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.

5. DISCUSSION

5.1. Refinements

5.1.1. Chipping direction

Our original approach to simplifying statements within a block was to start with the first statement and work toward the last. However, we can sometimes generate better variants if we work in the reverse direction. For example, consider the code fragment in Figure 9(a) for the symptom of `Yes` in the output. Chipping the body of `methA` proceeds as follow:

- block splitting. The block has four statements. (Recall that a sequence of if statements is treated together.)

```
1  void methA(){
2    int x;
3    x = ...;
4    if (x == 30)
5      ...
6    else if (x > 30)
7      ...
8    else
9      ...
10   println("Yes");
11 }
```

(a)

```
1  void methA(){
2    int x;
3    x = ...;
4
5
6
7
8
9
10   println("Yes");
11 }
```

(b)

```
1  void methA(){
2
3
4
5
6
7
8
9
10   println("Yes");
11 }
```

(c)

Figure 9. Example for direction of chipping: (a) shows the original code, (b) shows the best variant from forward chipping, and (c) shows the best variant from reverse chipping

  - removing lines 2-3 fails – `x` is used later
  - removing lines 4-10 fails – line 10 outputs symptom

- simplifying each statement in turn, from top to bottom
  - removing line 2 fails – `x` needs to be declared for later statements
  - removing line 3 fails – `x` needs to be initialized (to avoid an error from the Java compiler)
  - removing lines 4-9 succeeds
  - removing line 10 fails – it outputs the symptom.

Thus, the best variant, using forward simplification, consists of lines 2-3 and 10, as shown in Figure 9(b).

However, if we reverse the order

- simplifying each statement in turn, from bottom to top
  - removing line 10 fails – it outputs the symptom.
  - removing lines 4-9 succeeds
  - removing line 3 succeeds – `x`'s value is not used later
  - removing line 2 succeeds – `x` is not used later

Thus, the best variant, using reverse simplification, consists of line 10, as shown in Figure 9(c). The key point is that reverse simplification reflects how dependencies flow from the top of a block to its bottom.

Table II reported results for forward chipping. Table III lists portions of those earlier results and new results from using reverse chipping on those same tests. The data show that reverse chipping is never worse than forward chipping. In many cases, it is just slightly better, i.e., by a few percentage points (although any reduction in the size of the best variant is helpful). For one case, test T4, reverse chipping made a significant difference because that program is structurally similar to the program in Figure 9.

5.1.2. Statement shielding

The program chipper treats every statement, including variable declarations, as a candidate for chipping. We considered shielding some statements from the chipping process. In particular, we experimented with shielding variable declarations. Our hope was 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

Table III. Results for forward vs. reverse chipping direction

	parse tree nodes			forward–reverse	
	original	forward	reverse	nodes	percentage
test	number	best	best	difference	difference
T1	109	13	13	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%

```
void methA(){
  int x = -1;
  String y = "there";
  int z = 75;
  println("Yes "+b(x));
  String a = "hi "+y;
  if (z > 30)
    ...
  else if (z > 30)
   ...
  else
   ...
}
int b(int j){
  ...
  if (j < 0)
   return 30;
  ...
}
```

(a)

```
void methA(){
  int x = -1;
  String y = "there";
  int z = 75;
  println("Yes "+b(x));
}
int b(int j){
  ...
  if (j < 0)
   return 30;
  ...
}
```

(b)

```
void methA(){
  int x = -1;
  println("Yes "+b(x));
}
int b(int j){
  ...
  if (j < 0)
   return 30;
  ...
}
```

(c)

Figure 10. Example for statement shielding: (a) shows the original code, (b) shows the best variant from chipping `methA` without statement shielding of variable declarations and (c) shows the best variant from chipping `methA` with statement shielding of variable declarations

fragment in Figure 10(a) for the symptom of `Yes 30` in the output. Chipping 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`.

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

  At this point, only the statements on lines 2-5 remain in the block.

- 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 2-5, as shown in Figure 10(b). The chipper produced eight variants en route to this best variant.

Consider again the code in Figure 10(a). If the chipper instead shields the variable declarations 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 2-4 so that they are not removed. Thus, the variant tested consists of the shielded statements on lines 2-4 and the block statements on lines 5-12. This produces a good variant.
  - the chipper recurses on the block splitting algorithm and takes the first half of the second block (line 5) along with the shielded statements (lines 2-4) to produce another good variant.

  The chipper determines that the block is too small to split again, so it continues to the next part of 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 5, but this results in a bad variant because line 5 outputs the symptom.

  At this point, only statements on lines 2 and 5 remain in the block.
- 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.

The best variant with shielding of variable declarations has two lines, as shown in Figure 10(c). Thus, using statement shielding for this program, the chipper is able to produce a smaller best variant (two lines vs. four lines for `methA`'s body) while creating fewer variants (seven vs. eight).

Unfortunately, statement shielding has mixed effectiveness, which depends largely on the coding style used in the program being chipped. The example in Figure 10 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 because variable `x` is not initialized before it is used — a semantic error in Java. Thus, the chipper would not encounter a good variant during the block splitting step of block simplification. The chipper 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 fact, statement shielding can even produce larger best variants by keeping extra variable declarations that do not contribute to the symptom and, similarly, it may also produce a larger number of variants during simplification.

We experiemented with ChipperJ with statement shielding on the collection of programs in Table II. 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 the number of variants for T14 and T16, respectively.

Although statement shielding did not perform as well as hoped, the statement shielding framework should be useful in implementing other chipping features. One such feature is mentioned in Section 7.

```
public class A{
...
 int Ameth(){
  ...
  return 10;
 }
...
}
public class B{
...
 void Bmeth(){
  A a = new B();
  int i = a.Bmeth;
  if(i >0)
   print("symptom");
  else
   print(i);
 }
...
}
```

(a)

```
public class A{
...
 int Ameth(){
  ...
  return 10;
 }
...
}
public class B{
...
 void Bmeth(){
  A a = new A();
  int i = a.Bmeth;

   print("symptom");


 }
...
}
```

(b)

```
public class A{
...
 int Ameth(){
  // body removed
  return 0;
 }
...
}
public class B{
...
 void Bmeth(){



   print("symptom");


 }
...
}
```

(c)

Figure 11. Rechipping example: (a) shows the original code, (b) shows the result after the first pass of the chipper, and (c) shows how successive passes of the chipper further simplified the code

5.1.3. Order of file chipping and 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 11 shows how rechipping works. Figure 11(a) shows the original code of two classes. Notice that class `B` makes a call to `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 11(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 reduces this example to the code in Figure 11(c), in which the body of `Ameth` is now empty except for the type-appropriate return statement.

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

5.2. 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 best 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 the cause of a single bug (but, as indicated in Section 4, sometimes suffices).

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 12. 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

```
methD(){
  C c = null;
  c = new C(...);
  ...



  c.f(...);
}
```

Figure 12. Example for null pointer exception

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 multithreaded 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 shows 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 text boxes, 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 [36].

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 [39] 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 [39], 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 IV summarizes the results of running this delta chipper on a few of the smaller tests in Section 4.** As expected, compared to ChipperJ, the delta chipper 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.

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

Table IV. Delta chipping results (external mode only) on a few tests from Table II

	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

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 the Delta Debugging algorithm to all of the nodes in the same level in the parse tree's hierarchy using a top down approach. 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. A key difference between HDD and program chipping is that HDD strictly uses the Delta Debugging algorithm whereas chipping allows for a greater variety of strategies to be applied at various nodes in the parse tree according to node type. Unfortunately, there is no direct performance comparison between program chipping and HDD. HDD attempts to minimize C programs that causes a failure in *gcc*, while program chipping focuses on minimizing symptomatic Java programs.

As noted earlier, our work is related to program slicing. Program slicing was introduced in Reference [37]. Much work on slicing has followed, e.g., as surveyed in Reference [34]; as exemplified in specific variations of slicing such as described in Reference [35], and including interprocedural slicing such as described in References [21] and [4] and program dicing, introduced in Reference [38] 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 (e.g., the Bandera Slicer [8]) 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.

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).

7. CONCLUSION

This paper has introduced the idea of program chipping and described a chipper for Java programs. The results are encouraging. Our approach has limitations, as noted in Section 5.2, although they have not been a major problem in our experience. Still, though, we need to gain more experience in using ChipperJ in real practice.

Our experience described in this paper 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 consider additional techniques to speed up chipping, such as using multiple threads to detect early in a variant's execution whether the variant is good (which would then allow a potentially long-running variant to be killed off sooner). We believe program chipping will work equally well for non-object-oriented code and we hope to develop a ChipperC for C programs. Reference [31] discusses the above ideas further.

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. The tool will be GUI-based and will allow the developer to easily see the differences between variants as they are generated. (We plan to extend our *v-diff* tool, described in Section 3.3.) The statement shielding framework described in Section 5.1.2 will be useful in implementing this feature.

ACKNOWLEDGEMENTS

Trung Pham participated in discussions of this work and tested an early version of ChipperJ. The anonymous referees and shepard (Dieter Kranzlmueller) for the AADEBUG 2005 conference provided thoughtful comments and guidance on an earlier version of this paper (Reference [32]). Our colleagues Zhendong Su and Ghassan Mishergi have also helped us better formulate our ideas and presentations.

REFERENCES

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 JavaTM 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. 2005. in preparation.

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. C. D. Sterling. Automated bug isolation via program chipping. Master's thesis, Dept. of Computer Science, University of California, Davis, August 2005.

32. C. D. Sterling and R. A. Olsson. Automated bug isolation via program chipping. In *Sixth International Symposium on Automated and Analysis-Driven Debugging (AADEBUG 2005)*, pages 23–32, Monterey, California, Sept.19–21 2005.

33. Sun Microsystems. The Java bug database. http://bugs.sun.com/bugdatabase/index.jsp.

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

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

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

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

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

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

