

Elided Conditionals

Manos Renieris

Sébastien Chan-Tin

Steven P. Reiss

Computer Science Dept.
Brown University
Providence, RI 02912, USA

{er,sct,spr}@cs.brown.edu

ABSTRACT

Many software testing and automated debugging tools rely on structural coverage techniques. Such tools implicitly assume a relation between individual control-flow choices made in conditional statements during a program run and the outcome of the run. In this paper, we develop the notion of *elided choices* that, viewed in isolation, have no impact on the outcome of the program. We call the conditionals that make such choices *elided conditionals*. We develop an experimental framework for discovering elided conditionals. From looking at three programs of varying complexity under this framework, we discovered that elided conditionals do occur, sometimes with alarming frequency. We discuss the impact of elided conditionals on various forms of dynamic analysis and suggest future work that would extend elision to general expressions.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Reliability, Experimentation, Verification

Keywords

Elision, Coverage Testing, Dynamic Mutation

When you come to a fork in the road, take it.

— Yogi Berra

1. INTRODUCTION

Program coverage underlies program testing and automated debugging. Intuitively, our confidence in a fragment of code increases when the fragment executes (is covered)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE'04, June 7–8, 2004, Washington, DC, USA.

Copyright 2004 ACM 1-58113-910-1/04/0006 ...\$5.00.

during a correct run, and decreases when it executes during a failing run. This intuition often leads to wrong conclusions, with adverse effects for testing and debugging. A well-known manifestation of these effects is that given a program and a test suite that covers it fully, even if none of the tests fail, we still cannot be sure that the program harbors no mistakes.

In this paper we examine a special case of this phenomenon where individual conditional choices give us misleading coverage information. Conditional choices are the basic elements of coverage: coverage is simply an aggregate view of all the control-flow choices made during a program run. Therefore, when individual choices give us misleading information about the correctness of a fragment, the aggregation will be misleading also. In the particular case we examine in this paper a choice makes no difference to the program's outcome. Thus, the coverage of any program fragment due to this choice provides no evidence as to the correctness of the fragment.

Consider the program in Figure 1. Suppose that the expected outcome of the program is 1 and the value of the complicated expression is `true`. When the `then` clause is executed coverage provides (correct) evidence that both the condition and the `then` clause are correct.

However, suppose that the `else` clause is replaced with `x:=5;`, as in Figure 2, and there exist two test cases with different values of `c` for which the expected output is 1. The program will succeed in both cases, and coverage provides no reliable information on the correctness of either the condition or the executed clause. In effect, the conditional statement made no choice, and both the condition and the executed branch could be erroneous. Conversely, if the expected output is not 1 and `c` is equal to `true`, coverage points to the `then` clause as suspect. However, since the condition made no choice, it may well have been wrong, and then the `else` clause could be the erroneous one.

```
c := (* complicated expression *)
if (c)
then x := 3;
else x := 4;
print x mod 2;
```

Figure 1: A simple if-then-else

```

c := (* complicated expression *)
if (c)
then x := 3;
else x := 5;
print x mod 2;

```

Figure 2: An elided if-then-else

We call a particular evaluation of a conditional during a run *elided* if the run has the same outcome regardless of which branch is executed. We also call the conditional itself elided. If all evaluations of a particular conditional during a run are elided, we say that the conditional was *totally elided* during the run.

Elision is a dynamic concept. The elision of a particular evaluation of a conditional is affected by both the program state at the time of the evaluation and the future of the run. In our first example, if instead of `x := 5` the `else` clause read `x := y`, `y`'s value would influence the conditional's elision. Moreover, if the conditional is reached twice, once with `y` odd and once with `y` even, it would be elided only once. The rest of the computation, and how it handles the outcomes of each branch, also determines whether the conditional is elided. In our example, the immediate effects on `x` are discernible, but the conditional is elided by the modulo operation. Therefore, the particular dynamic context affects whether a statement is elided or not.

```

x := 5;
c := y < 10;
d := true;
e := (y < 50) and (z < 10);
if (c or else d or else e)
then x = 3;
print x mod 2;

```

Figure 3: Elision with boolean operators.

At first sight, it seems unlikely that conditional elision is a common phenomenon. After all, every `if` statement in a program is there for good reason, and programs avoid recomputing the same values, as dictated by efficiency concerns and good algorithm design. But these are local and static properties: in full programs, at run time, decisions are often repeated or have no effect on the program's outcome. Programs do not output their internal state; we only have narrow windows through which to discern their behavior. In Figure 3 we cannot observe the whole value of `x`, only its lowest bit. Second, elision refers to a dynamic property of the program, and not every conditional statement has a crucial place in all contexts. In Figure 3, given that `d` is true, `c` is elided, although used¹. Lastly, the same conditional check may appear multiple times in different guises. In Figure 3 the computations of `c` and `e` both check whether `y < 50`.

To discover if a conditional is elided or not, we need to pose the following question: how does the output of the program change if the condition is inverted? To answer the question experimentally, we forcibly change the value of the condition just before the direction of the branch is decided,

¹This example also illustrates that elision is not static or dynamic slicing: `c`, `d` and `e` would be in the backward static slice of `x`, and `c` and `d` would be in the dynamic slice.

and we will monitor any differences in the observable behavior of the program. In this paper, we focus on *single elided conditionals*: in other words, we make a single inversion during a run. We repeat the process for each conditional, starting each time in the original run and inverting the condition immediately following the last condition we inverted (Figure 4).

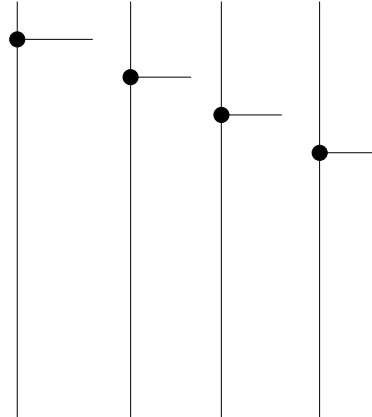


Figure 4: The structure of the elision exploration. Each of the vertical lines represents a run identical to the original run; each dot on a vertical line represents the conditional statement we examine; each horizontal line represents a run that deviates from the original in that we forcibly change the value of the condition.

Examining both branches of each `if` is akin to introducing faults into the subject program. We would replace every statement of the form `if(c)` with a statement of the form `if(c xor s)`, where `s` is a boolean “slack” variable. Then we allow only one slack variable to be `true`, exactly once through the program run, and we examine its effects. We replicate the process for each slack variable and for each evaluation of the condition during the run. In contrast with other approaches like mutation testing [3], we do not need to specify directly how the new conditional differs from the original. This allows us to separate all possible mutations of the conditional into two groups: the group that leaves all decisions intact, and the group that inverts the decision once. Then we can examine both groups without enumerating them. Therefore, we can implicitly address a large number of potential faults and examine whether a test case would expose them.

The rest of this paper is structured as follows: section 2 discusses our experimental framework, section 3 describes our experimental results, and section 4 discusses related work, including the effect of elision on automated testing and debugging. Section 5 discusses future work.

2. IMPLEMENTATION

We have built an experimental framework for discovering elided conditionals with two coupled components:

- An *instrumentation component*, that rewrites the source code of the subject program to let us explore both branches of each conditional.
- A *monitoring component*, that lets us examine the differences in the program's behavior when we follow the

branch of a conditional not dictated by its evaluation context.

Our instrumentation system is built on top of CIL [13], a source-to-source transformation library for the C language. We first apply two of CIL’s own transformations to the subject program. The first one transforms all loops into simple conditional statements, as in Figure 5. The second transforms all binary boolean operators to sequences of nested `if` statements, as in Figure 6. Both transformations insert `goto` statements as necessary.

```
while (i != EOF) {
    // loop body
}

BECOMES

while (1) {
    while_0_continue:
    if (!(i != EOF)) {
        goto while_0_break;
    }
    // loop body
}
while_0_break:
```

Figure 5: Sample CIL loop transformation

```
if (i == '#' || i == '@') {
    // conditional body
}

BECOMES

if (i == '#') {
    // conditional body
} else {
    if (i == '@') {
        // conditional body
    }
}
```

Figure 6: Sample CIL boolean transformation

After these two transformations, we apply one of our own that transforms all statements of the form `if (c)` into statements of the form `if (split(c))`. The three transformations together ensure that we call the `split` function on every branch of the program.

The `split` function, implemented on top of the Unix `fork` system call, is an identity function with an important side effect: it duplicates the running process. Following Unix terminology, we call the newly created process the *child process*. The child process differs in two major ways from the original parent process: the return of `split` is inverted, and future invocations of `split` have no side effects (and return their argument unmodified). In this way, we create a number of processes equal to the number of branches encountered during the program execution (Figure 7). This can dramatically increase the running time of a test run.

Our monitoring system is implemented as a modified C library. Our C library behaves differently for the original process, which follows all the branches as if we had not inserted the calls to `split`, and for each of the child processes,

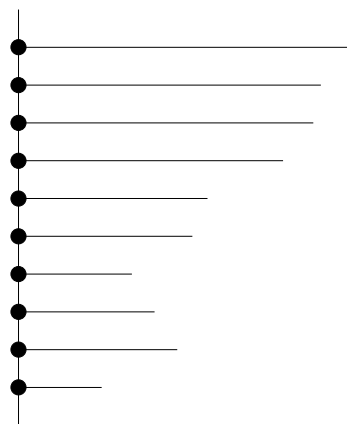


Figure 7: Implemented elision exploration, combining the prefixes of processes.

which have one inverted decision. For the original process, the monitoring system simply records all output, including writing to files and terminals; it also records the process’s total execution time. For the child processes, the monitoring system checks whether the output agrees with that of the original process, and also checks that the running time does not exceed the running time of the original process multiplied by three. We stop a child process as soon as it (irrevocably) behaves differently from the parent process². In general, any difference that a user might observe should be monitored; however, it is of equal importance not to flag differences that the user cannot observe.

We start each of the child processes in a blocked state. When the parent process finishes, we resume the child processes one at a time and monitor for any observable differences from the parent process.

3. EXPERIMENTAL RESULTS

We experimented with three C programs: *tcas*, *print_tokens*, and *space*. Two of these programs, *tcas* and *print_tokens*, come from the Siemens suite [9, 16]; the third is a program written for the European Space Agency [17, 22]. Each of these programs comes in a number of versions, all but one of which contain faults. We used the correct version for our experiments.

tcas is an aircraft collision avoidance system. It consumes 12 integers and outputs 0, 1, or 2, depending on whether the airplane should increase, keep, or decrease its altitude. It is essentially a large predicate. Few programs have the same structure as *tcas*, but many programs contain complex predicates. *print_tokens* is a lexical analyzer and *space* is an interpreter for an array definition language (ADL).

Because of *tcas*’s structure, we expect that most conditionals will be elided. We expect elision to be less extensive for the other two programs, since they output much more information about their inputs.

We run each program on a number of tests, as shown in Table 1. For each run, we kept information on how many times each conditional statement executed and how many times it was elided. Table 2 summarizes our findings. The

²We assume that the program will not undo any of its actions; for example, it will not write into the same position of a file twice.

Program	Description	LOC	#Cond	#Tests
<i>tcas</i>	altitude separation	173	33	1608
<i>print_tokens</i>	lexical analyzer	565	88	4132
<i>space</i>	ADL interpreter	6218	622	13297

Table 1: Overview of the Siemens suite. Column 3 is the number of lines of code; column 4 is the number of conditionals.

Program	Boolean Coverage	Boolean Elision	Ratio	Avg. False Coverage
<i>tcas</i>	23615	14923	0.63	9.3
<i>print_tokens</i>	238787	8765	0.04	2.1
<i>space</i>	2769599	366198	0.13	27.5

Table 2: Boolean Coverage Results

Program	Count Coverage	Count Elision	Ratio
<i>tcas</i>	24501	15809	0.65
<i>print_tokens</i>	4291376	533391	0.12
<i>space</i>	63923979	15979909	0.25

Table 3: Profile Results

first column contains the program name. The second column contains the aggregate boolean coverage of conditionals: the number of conditionals evaluated during all runs, when each conditional is counted only the first time it executes during a run. The third column contains the aggregate boolean elision: the number of conditionals elided during all runs, when each conditional is counted once for each run where it was totally elided, i.e. all its evaluations were elided. As we would expect, 63% of conditionals are elided in *tcas*, 13% in *space*, but only 4% in *print_tokens*. The fourth column contains the ratio of columns 2 and 3.

Column 5 is the average number of elided conditionals during a run. These conditionals still executed and thus are reported as covered, but this information is misleading. Nine conditionals are elided on average in the runs of *tcas*. This is out of 33 conditionals in the entire program, although not all of them execute during all runs. For *print_tokens* and *space* the numbers are lower but still significant. One dramatic example appears in *tcas*. The original, uninstrumented program contains a conditional of the form `if (a && b && c)`. The `c` part of the conditional is executed in 245 runs, yet it is totally elided in all of them. As a result, its computation remains untested, after 1608 tests. Figure 8 shows a graphical summary of all the *tcas* conditionals. Another example appears in *space*, where a nested conditional is elided in 61 of the 75 runs in which it executed.

Certain applications [1, 14], take advantage of profiles, rather than boolean coverage. Table 3 contains the same information as the first four columns of table 2 but we count each conditional evaluation individually. The elision ratios increase, because in this table we count elided evaluations of a conditional that coexist with non-elided evaluations of the same conditional in the same run.

4. DISCUSSION AND RELATED WORK

Since elided conditionals occur, we believe that dynamic analyses should take them into account. The result of a

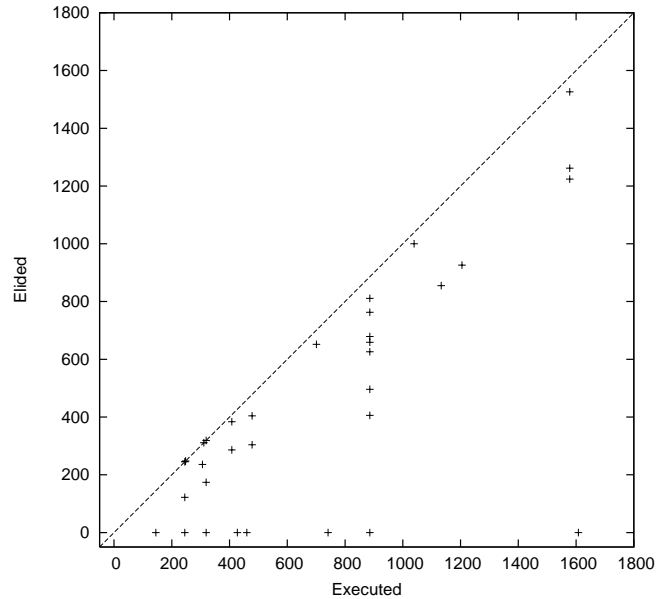


Figure 8: *tcas* coverage and elision data per conditional. Each conditional is represented as a point. The x coordinate of the point is the number of runs the conditional executed in; the y coordinate is the number of runs the conditional was totally elided in. Points close to the x axis represent conditionals that were rarely or never elided. Points close or on the diagonal line represent conditionals that were often or always elided.

dynamic analysis may depend strongly on a single run, for example a run that exposes a fault or executes an otherwise uncovered line of code. In such cases, elision of a single conditional during that important run may reduce the quality of the result significantly. In general, any information discovered during dynamic analyses that is not true for both branches of each elided conditional in the program should be discarded. Branch and statement coverage should not consider as covered the branches of elided conditionals. In def-use chain tracing, an elided conditional should not constitute a use. Path profiling for fault localization [15] should not include paths that include elided conditionals. Invariant detection [6] should not include invariants stemming from branches of elided conditionals. Automated debugging based on coverage data [10, 14, 15], the primary motivation for this work, should not include elided branches in spectra comparisons.

A number of other testing and debugging approaches modify the program behavior, either statically or at run time. Dynamic mutation testing [11] establishes the sensitivity of code by changing function return values. Delta Debugging experiments with splicing parts of the state of a successful execution onto a failing one in order to isolate cause-effect chains [23]. Our approach is more restrained: we only change boolean values in conditions. This frees us from the need for an alternate run and allows us to exhaust the space of changes.

Critical slicing [4] selectively removes statements to examine whether they affect a slice. Removing a statement is equivalent to leaving the program state intact. For conditionals, this technique would not always examine both branches.

Mutation testing [3] modifies the subject program; the user has to augment a test suite until it distinguishes all mutants from the original program. Elision can be used in this way: the user would build a test suite in which all conditionals matter at least once.

Elision can be a tool in the search for causal relations for program faults [7, 23]. By its nature, elision modifies the program in minimal ways, consistent with Lewis’s theory of causation [12]. When a program does not fail even if a conditional is changed, this conditional can clearly not be the cause of the fault.

Modified condition/decision coverage [2] requires a test suite that contains two tests for each minimal predicate (called a *condition*) of each conditional statement (called a *decision*): one test in which the minimal predicate is true and the conditional statement follows one branch, and one test in the which the minimal predicate is false, the conditional statement follows the other branch, and all other minimal predicates in the conditional statement have the same values as in the first test. In this way, the pair of tests demonstrates that changing the minimal conditional alone is sufficient, on occasion, to make the conditional statement follow the other branch. Non-elision is a stronger property: the choice of conditional branch must have an impact on the program output.

We can view elision as a dynamic information-flow problem [5]. If we consider the regular input of the program as public and the values of the slack variables (c.f. section 1) \mathbf{s} as private, then the elision question becomes equivalent to the information-flow question whether we can find the values of \mathbf{s} from the public inputs and the output. However, the enforcement of secure information flow is often local in the code [18], and in elision we are interested in remote effects.

Such remote effects are captured by the RELAY model of fault origination and transfer [19, 20]. Assuming a fault from a class of syntactic faults, RELAY constructs a necessary and sufficient condition for the observation of the fault. RELAY targets a relatively simple language (for example, the target language has no pointers), and yet the conditions it needs to construct are complex and sometimes fault specific. It is probably impractical to instrument a program to monitor the condition constructed by RELAY. Still, the model provides insight into the complexity of the problem, and could provide local necessary conditions that could be exploited to improve the efficiency of a brute-force method like ours.

Randomized execution [8] is a static analysis technique in which both branches of conditional statements are executed, and the states are combined in an affine manner at join points; this allows for efficient discovery of affine transformations and value numbering.

5. FUTURE WORK

In its present implementation, elision discovery is prohibitively expensive. Smarter state sharing, as in the Java Pathfinder [21], would facilitate its use. This would also let us experiment with multiple (boolean) faults.

Elided conditionals are but the simplest case of elision. In the more general case, we would have to perturb a single non-boolean variable at every expression. This is much harder: the alternative domain may be much larger and not well defined. Consider an integer used as an index to an array. Its domain, not taking into account its usage, is the full

domain of integers. If we take its usage into account, then its domain in an unsafe language like C remains unchanged, but in a safe language like Java it becomes the union of the valid array indexes and a single item that would cause an out-of-bounds error. The situation becomes harder when the variable we want to manipulate is a pointer. Its domain could include every item in the data structure it points to, every item of the same type as the item currently pointed to or, in unsafe languages, any address in memory. The situation dictates that we develop specific fault models for each language and perhaps each program.

This paper developed the notion of elided conditionals and we showed experimentally that they do occur in some small to medium sized programs. We believe that elision has an impact on all forms of dynamic analysis. In future work, we plan to generalize the concept and explore the significance of elision for automated debugging.

6. ACKNOWLEDGMENTS

Pascal Van Hentenryck offered valuable insight into fault models and urged the explanation of elision with perturbed conditionals. Willem Visser provided helpful comments on a draft of this paper.

7. REFERENCES

- [1] Thomas Ball. The concept of dynamic analysis. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 216–234, 1999.
- [2] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.
- [3] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
- [4] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 121–134, 1996.
- [5] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
- [6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.
- [7] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software*, volume 2648 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
- [8] Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 74–84, 2003.
- [9] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria.

- In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.
- [10] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.
- [11] Janusz Laski, Wojciech Szermer, and Piotr Luczycki. Dynamic mutation testing in integrated regression analysis. In *Proceedings of the 15th international conference on Software Engineering*, pages 108–117, 1997.
- [12] David K. Lewis. *Counterfactuals*. Harvard University Press, 1973.
- [13] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, April 2002.
- [14] Manos Renieris and Steven P. Reiss. Fault localization with nearest-neighbor queries. In *Proceedings of the 18th IEEE Conference on Automated Software Engineering*, pages 30–39, 2003.
- [15] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 432–449, September 1997.
- [16] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1998.
- [17] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.
- [18] Andrei Sabelfeld and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
- [19] Margaret C. Thompson. *An Investigation of Fault-Based Testing Using the Relay Model*. PhD thesis, University of Massachusetts at Amherst, May 1991. Available as Technical Report 1991-022, Computer Science, University of Massachusetts at Amherst.
- [20] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. An information flow model of fault detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 182–192, 1993.
- [21] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.
- [22] Filippos I. Vokolos and Phyllis G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, pages 44–53, November 1998.
- [23] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, 2002.