Reducing regression test size by exclusion.

Keith Gallagher[1]
Tracy Hall[2]
Sue E. Black[3]

[1] Centre for Software Maintenance and Evolution, Durham University
[2] School of Computer Science, University of Hertfordshire
[3] School of Electronics and Computer Science, University of Westminster

Reducing Regression Test Size by Exclusion

Keith Gallagher
Centre for Software Maintenance and Evolution
Durham University
South Road
Durham DH1 3LE
United Kingdom
k.b.gallagher@durham.ac.uk

Tracy Hall
School of Computer Science
University of Hertfordshire
College Lane
Hertfordshire AL10 9AB
United Kingdom
t.hall@herts.ac.uk

Sue Black
Harrow School of Computer Science
University of Westminster
Watford Road
Harrow HA1 3TP
United Kingdom
s.black3@westminster.ac.uk

Abstract

Operational software is constantly evolving. Regression testing is used to identify the unintended consequences of evolutionary changes. As most changes affect only a small proportion of the system, the challenge is to ensure that the regression test set is both safe (all relevant tests are used) and inclusive (only relevant tests are used). Previous approaches to reducing test sets struggle to find safe and inclusive tests by looking only at the changed code. We use decomposition program slicing to safely reduce the size of regression test sets by identifying those parts of a system that could not have been affected by a change; this information will then direct the selection of regression tests by eliminating tests that are not relevant to the change. The technique properly accounts for additions and deletions of code.

We extend and use Rothermel and Harrold's framework for measuring the safety of regression test sets and introduce new safety and precision measures that do not require a priori knowledge of the exact number of modification-revealing tests. We then analytically evaluate and compare our techniques for producing reduced regression test sets.

1 Introduction

Michael Cusumano, writing in the July 2006 issue of Communications of the ACM regarding Microsoft's March 2006 announcement that the delivery of the new Vista operating system (known internally in Microsoft as "Longhorn") would be (again) delayed to early 2007, notes:

> Making even small changes in one part of the product led to unpredictable and destabilizing consequences in other parts since most of the components were tied together in complex and unpredictable ways. Even 4,000 or so software developers and an equivalent number of testers was not enough to get Longhorn working [5].

Making a change to existing software, whether in production or development, system generates two problems. The first is validating that the change is correct. The second and more difficult problem is determining that no error has been inadvertently introduced. Ostrand, et al., found a 50-80% probability of introducing an error while making a change to code [11]. Regression testing is the method to uncover the unintended side effects of making changes to software. Regression testing can uncover the unintended side effects of making changes to software. It demonstrates, insofar as any testing method can, that no changes have rippled into other parts of the system [3].

The main difficulty of regression testing is determining test sets that are both safe and inclusive. Inclusive means that only tests that could possibly uncover an error are used. Safe means to include all tests that could possibly uncover an error. (Simply running all tests would clearly be safe, but not inclusive.) A good set of regression tests is inclusive and safe: the suite covers all parts of the system that may have been affected by a change, but does not cover parts of the system that could not have been affected by that change.

We are interested in finding safe and inclusive regression tests. Moreover, most changes are relatively small with respect to the size of the entire system,

1-4244-1256-0/07/$25.00 © 2007 IEEE

so while the test set to determine that the change is correct could be quite small, the number of regression tests required could be relatively large. In an informal application of Amdahl's Law that small improvements in large efforts may surpass large improvements in small efforts [1], we aim to turn the problem of inclusion and safety on its head by finding tests that could be excluded, and excluded in such a way that running the excluded test would not show any detectable change in the program's behaviour.

This approach has the decided advantage of always being safe. The cost is a loss of precision; some ineffectual tests may be run. There is always a cost associated with selecting regression tests. It is not difficult to imagine that in some cases the cost of finding the regression tests may exceed the cost of merely re-running all the tests. Our technique proposes, that in the larger scheme, running a few unnecessary tests is a small price to pay. It is certainly better to run a few unnecessary tests than to miss a necessary one.

1.1 Contribution and Impact

We demonstrate how decomposition slicing [9, 10] can enable the reduction of regression test set size by test elimination. Decomposition slicing is a program slicing technique that identifies all lines of code that are related to a variable of interest and thus provides a technique to identify *unchanged* parts of the system. The technique correctly accounts for additions and deletions. The technique can be compared to other regression test set inclusion techniques by extending an existing framework for the comparison of inclusion techniques [12]. New metrics are introduced to address the enhanced framework.

There are two contributions to reducing regression test set size. The first is alluded to in the Cusumano quotation above: change testing is expensive and any help in reducing the effort is significant. The second effect is subtle: an a priori estimate of the size of regression tests needed could direct the style of the change (even, perhaps, changing the priority of a change request due to regression test cost).

1.2 Outline

After this introductory section, the paper is organized as follows. Section 2 gives the background, discussing the conceptual framework for the subsequent presentation. The work that we build upon is also summarized. Section 3 re-frames the conceptual framework in what we believe is a more coherent way; the new-found coherency allows an easy extension to the framework. Metrics that correspond to the enhanced framework are introduced. Section 4 describes our new technique and analysis in the revised framework. Section 5 gives some examples and Section 6 summarizes the contribution and details future directions.

2 Background

The aim of regression testing is to validate unmodified parts of software systems, to make sure that the modification process does not introduce any new errors and provide confidence that the modifications are correct. Regression testing is a very necessary, but very expensive, software development and maintenance activity. Ensuring that an accurate regression test suite is chosen is of critical importance in reducing the cost of software engineering activities.

Regression test selection techniques attempt to reduce the amount of time needed to retest a modified program by selecting a subset of the existing test suite. Techniques such as path analysis, program dependence, cluster identification and program slicing are used to varying effect to reduce the amount of tests that need to be run thus reducing the amount of effort and therefore cost.

Regression testing is not about validating that the change is correct; that the change is correct is the bailiwick of the maintenance engineer and not in the scope of this paper. However, in subsequent versions of a system, tests that were created to validate a change in the system become part of the regression test suite. A new feature that is added to a system must also be protected against unintended change.

2.1 Regression Test Selection Analysis

Rothermel and Harrold [12] (hereafter referred to as "RH96") describe a framework for evaluating regression test selection techniques in terms of inclusiveness, precision, efficiency and generality. Table 1 summarizes their framework.

The analysis framework described in RH96 is functional. Tests for code fragments that are syntactically changed, but not semantically changed, would be classified as non-modification-revealing. Thus any regression test selection technique that would select the tests for such a code fragment would be considered as imprecise. For instance, consider the example of Figure 1, taken from RH96. While it is a bit contrived, it does show a technical side of the definition of precision. Since the code is functionally equivalent,

Measure	Description
Inclusiveness	The measure of the extent to which a technique chooses the tests that will cause the modified program to produce different output than the original program and thereby expose faults caused by modifications *Safety is a property of inclusiveness.*
Precision	The measure of the ability of a technique to avoid choosing tests that will not cause the modified program to produce different output than the original program. *No property of precision is given in [12].*
Efficiency	The computational cost, and thus, practicality, of a technique.
Generality	The ability of a technique to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications

Table 1: Summary of Rothermel and Harrold Terminology. Efficiency and generality are shown for completeness, but are not a focus of this work.

```
read (x)                  read (x)
if ( x <= 0 )             if ( x <= 0 )
   if ( x == 0 )             if ( x == 0 )
      print ( x + 2 )           print ( x + 2 )
      exit                      exit
   end                       else
                                print ( x + 3 )
                                exit
  end                       end
  print ( x + 3 )           print ( x + 3 )
  exit                      exit
```

Figure 1: Semantically Equivalent Code Fragments.

any regression technique that selects tests for this code is imprecise.

In RH96, a modification-revealing test (hereafter "*mr-test*") is one upon which P and P′ differ; P is the original program and P′ is the modified version. *Inclusiveness* is the measure of the number of modification-revealing tests that regression test selection technique M finds, *relative to the total number of modification-revealing tests.* (Our emphasis.) The selection technique is *safe* when inclusiveness is exactly 1. Note that since safety measures only modification-revealing tests, that it is bounded above by 1.

A non-modification-revealing test (hereafter "*nmr-test*"), is a test, `t`, where results of running `t` on P and P′ yield the same result. In RH96, *precision* is the measure of the number of nmr-tests that regression test selection technique N excludes, *relative to the total number of non-modification-revealing tests.* (Our emphasis.) The selection technique is *precise* when the ratio is 1, and like safety, bounded above by 1.

In RH96, inclusiveness and precision are complementary measures, and safety is a property of the measure inclusiveness. When investigating these ideas, we applied some simple software metric analysis. "Measurement is the process by which *numbers* ... are assigned to *attributes* of *entities* ..." [7]. (Our emphasis.) We found the following adjustments in the terminology helpful. The entities that we will measure will be techniques for obtaining regression tests. The attribute that we will measure will be inclusiveness / exclusiveness. The measure will be *safety* (for inclusiveness) and precision (for exclusiveness). Table 2 summarizes our changes. A singular advantage to this re-casting is that now we can apply safety measures to exclusive techniques and precision measures to inclusive techniques. More discussion of this idea follows, but at this point we note that one implication is that precision and safety are no longer bounded above by 1.

Our definition of precision is different from RH96; they use precision to measure what we call the attribute of exclusion and have no term analogous to safety. We use safety to measure inclusion; and we use precision to measure exclusion. The change in terminology from RH96 is so that there are pairs of complementary terms: inclusion and exclusion as attributes; and safety and precision as measures. RH96 has inclusion and precision as complementary measurement terms, and no complementary term for safety.

RH96 defines inclusiveness, etc, in terms of mr-tests. They point out that other definitions may be

Attribute	Description
Inclusiveness	The extent to which a technique chooses the tests that will cause the modified program to produce different output than the original program and thereby expose faults caused by modifications *Inclusiveness is measured by safety.*
Exclusiveness	The ability of a technique to avoid choosing tests that will not cause the modified program to produce different output than the original program. *Exclusiveness is measured by precision.*

Table 2: Our Changes to the Rothermel and Harrold Terminology

appropriate. For instance, data-flow techniques can be used to define *definition-use*-inclusiveness.

2.2 Regression Test Selection Techniques

This subsection surveys two techniques for selecting regression tests. It is by no means complete; it does however, relate to the technique we present. Both techniques are used for selections. The reason that these were chosen is that the first uses dynamic information that is in turn attached to the static source. The second uses program slicing.

2.2.1 Textual Differencing

Vokolos and Frankl [13] use textual differencing, which compares program sources. Source statements are mapped to the basic block in which they are contained; the basic blocks are mapped to the test cases which execute them. When a textual difference is observed in a basic block, the tests that execute that block are selected. The textual differencing technique is robust enough to observe changes in basic block structure; in this case, a basic block at a shallower nesting level than the changed code is used to determine which tests to select.

The empirical evidence reported by Vokolos and Frankl is promising. The reductions are significant and easily obtained. The tool used to construct the test set reduction, *Pythia*, was a straightforward cobbling of Unix tools, a decided advantage. They do not report, and evidently had no way of determining, if any regression tests were missed.

Chen, et al., [4] use a similar technique in *TestTube*. Instead of using basic blocks, à la Vokolos and Frankl, they identify which macros, functions, types and variables are covered by a test, then select tests that correspond to a changed entity.

It is interesting to note that these techniques are considered imprecise in the RH96 framework. The test for the changed code illustrated in Figure 1 would be selected. It seems that this is exactly what a tester wants to know. It may be that the function of code fragment has not changed, but many software evolution activities do not require, and in some instances, do not permit a changed functionality. However, non-functional changes have to be tested, too.

2.2.2 Semantics Guided Selection

Binkley [2] uses a semantics-guided selection technique. Semantics in this case is the sequence of states of computation through which a program passes. The program must be "rolled out" (all procedure calls appropriately substituted in-line). Once the semantics is clarified, the differences between a `certified` and a `modified` variant of a program can be determined. The technique then selects tests that exercise components of `certified` that have similar semantics (state traces) in `modified`. Some care must be exercised in defining "similar semantics." The test set selection is guided by *calling context slicing*, a technique for calculating precise interprocedural slices. The calling context slices are used to find common execution patterns, which in turn are used to find the similar semantics. The outcome is that tests which are possibly modification revealing are selected.

2.3 Decomposition Slicing

Program slicing is a software analysis technique that extracts program statements relevant to a particular computation. Informally, a slice provides the answer to the question "which program statements potentially affect the computation of variable v at statement s?" While conventional slices capture the value

Program parts	Includes statements which are ...
Independent	in the decomposition slice taken with respect to v that are not in any other decomposition slice.
Dependent	in the decomposition slice taken with respect to v that are in another decomposition slice.
Complement	not independent, i.e., statements in some other decomposition slice, (but not v's).

Table 3: Classification of Decomposition Slice Variables. Assume the computation of variable v is the variable of interest

of a variable at a particular program point, a decomposition slice captures all computations of a variable and is independent of program location [9, 10]. Decomposition slicing can be used to partition a program into three parts as described in the Table 3.

For software testing, only independent and dependent statements i.e., the decomposition slice taken with respect to v are of interest. The complement statements *cannot* be affected by the change. While a program has as many decomposition slices as it has variables, many of the decomposition slices on different variables are exactly the same [8]. Empirical analysis showed that reductions by equivalent decomposition slices range from 9-87% with a mean percentage between 50% and 60% ($p < 0.005$ with 95% confidence). This means that any test that exercises any variable in a cluster will satisfy every variable in that cluster and a tester may be assured that all the variables in the cluster are covered by one test. Test selection is tremendously simplified by this clustering feature of decomposition slicing.

Figure 2 illustrates the reductions that are possible. The graphs in Figure 2 are decomposition slice graphs of a differencing program. The nodes are partially ordered by set inclusion. A lower node is properly contained in any node above it that it is connected to. The graph on the left has 77 nodes and 271 edges, from a source size of about 700 lines. The graph quickly grows into unwieldy proportions. The graph on the right of Figure 2 is the reduced graph showing only computationally equivalent nodes, with only 34 nodes and 43 edges. The equivalence is determined by simple set equality. Another advantage to the (visual) reduction is the easy estimate of the size of the change and regression test effort. Any node that has a change must have all the nodes that contain it (i.e., are above it and connected in the graph) tested also [10].

Unlike decomposition slicing, other slicing approaches do not identify all additional statements (i.e. the dependent part) of the slice that is related to the original change. Consequently these other slicing approaches are all unsafe as inclusion techniques. Using slicing as an exclusion technique is automatically safe. The advantage of a decomposition slicing based approach for test exclusion is two-fold: any variable that is in the complement part is automatically excluded, as are all the variables in its computationally equivalent set.

3 Extension

Changes are typically small with respect to the size of the entire system. The natural inference is that the mr-test set is also proportionally small. As the set of nmr-tests is thus proportionally large, even small additions to its size will have a large payoff. Reducing the number of regression tests, by excluding ones that will show no change, can have immediate practical impact.

3.1 Example

We start with an example. Suppose we have two ways of obtaining regression tests. Technique **M** selects tests to be run and so is an inclusive technique. Technique **N** omits tests that do not need to be run and so is an exclusive technique.

Now suppose test suite T contains 1000 tests; for change from P to P′, suppose there are 200 mr-tests and 800 nmr-tests. If regression test selection technique **M** selects 150 of the 200 mr-tests, its inclusiveness has a safety measure of $150/200 = 0.75$; if a regression test selection technique excludes **N** 700 nmr-tests, it has a precision measure of $700/800 = 0.875$.

It is safe to run the 300 tests that are not excluded by **N**. It is not inclusive in the RH96 measurement sense; in this case the RH96 inclusive measure cannot be applied because we do not actually know that there are 200 mr tests. If we somehow had the knowledge of the actual number of mr tests the ratio would be > 1,

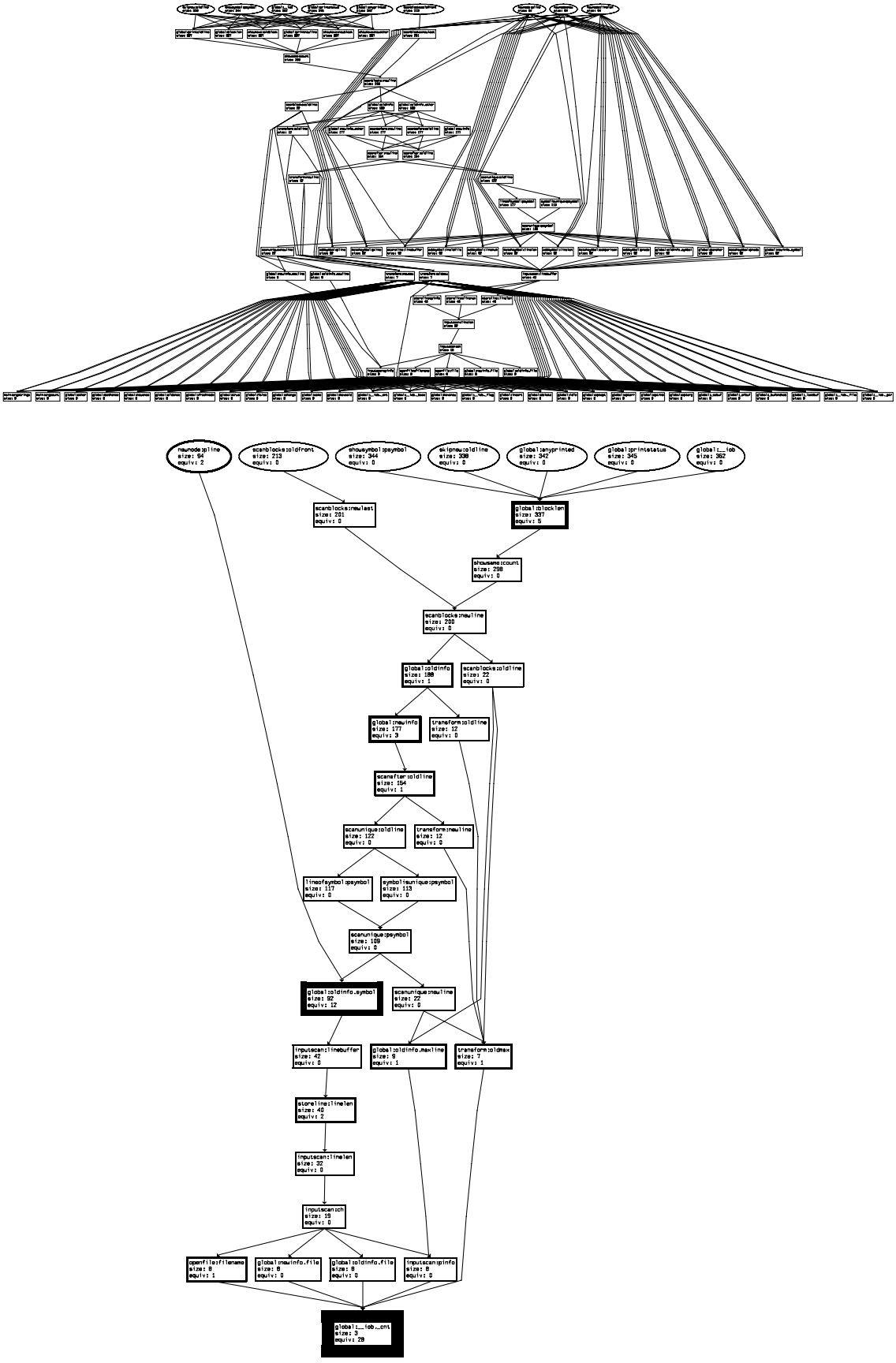

Figure 2. A Decomposition Slice Graph and Its Reduction by Equivalent Slices

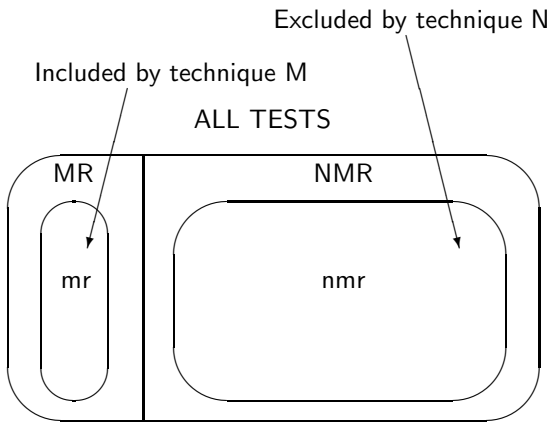

Figure 3: The set of all tests is partitioned into MR and NMR sets. A particular inclusion or exclusion technique does not find all the tests that should be included / excluded.

not possible in the RH96 framework. The true cost is running the 100 extra tests that we know to be nmr. A cost / benefit calculation will determine whether or not this is suitable.

Suppose each technique is increased by 10%. We now have 165 mr-tests and 770 nmr-tests. The safety of **M** is 165/200 = 0.825. This safety measure is still unknown to us, for the exact number of mr-tests is not known. With 770 nmr-tests, the precision measure of **N** is given by 770/800 = 0.9625. If we now use the safe 230 tests not excluded by **N** as a regression test suite, our cost drops significantly, but we still don't know exactly how much.

In general, let `t` be total number of tests, `n` be the number of mr-tests, `s` be the number of nmr-tests; let `m` be the number of mr-tests found by inclusive technique **M** and let `r` be the number of nmr-tests found by technique exclusive technique **N**. While mr- and nmr-tests are always arithmetically complementary, safety and precision are precisely related only when they are equal to 1. Whenn a inclusive technique is safe, `m = n`; when an exclusion technique is precise, `r = s`. That is, a safe inclusive technique is precise and a precise exclusive technique is safe. Also note that when `n` and s are known, the regression testing problem is solved. Indeed, the problem is finding that partition.

3.2 New Measures

Figure 3 shows a simplified diagram of the RH96 framework. Figure 3 shows all possible regression tests are decomposed into all modification revealing (MR) and all non-modification revealing (NMR) tests. An inclusive selection technique (M) identifies a sub-set of modification revealing tests (mr). Inclusiveness is calculated as mr / MR and safe inclusive test sets are when mr = MR. Similarly an exclusion-based selection technique (N) will find a sub-set of all non-modification revealing tests (nmr). RH96 identifies only modification revealing test sets; they do not further decompose the measurement of precision.

The premise of the measures and crucial to the example above is the *a priori* knowledge of `m` and `n`, the actual number of mr- and nmr-tests. If these values are not known, how can the precision and safety be estimated?

To estimate safety and precision when `m` and `n` are not known, we propose using exclusion to measure an inclusive technique's safety, and inclusion to measure an exclusive technique's precision.

Thus we define the *safety of inclusive technique* **M** *with respect to exclusive technique* **N** as the ratio of the number of tests found by inclusive technique **M** to the number of tests that exclusive technique **N** does *not* find. We call this *e-safety(M,N)*.

In the example above, inclusive technique **M** selects 150 tests; exclusive technique **N** misses 300 tests. Thus e-safety(M,N) is 150/300 = 0.5

Similarly, we define *i-precision(N,M)* as the ratio of the number of tests found by exclusive technique **N** to the number of tests that inclusive technique **M** does *not* find. In the example, i-precision(N,M) is 700/850 = 0.82. An e-safe technique is safe when **N** is precise; an i-precise technique is precise when **M** is safe.

Continuing with the example, when inclusive technique **M** finds 10% more (15) mr-tests e-safety(M,N) is 165/300 = 0.55; and i-precision(N,M) is 700/815 = 0.86. When exclusive technique **N** finds 10% more 70 nmr-tests e-safety(M,N) is 150/230 = 0.65,; i-precision(N,M) is 770/850 = 0.91.

The examples show an application of Amdahl's law. By improving the actual value of nmr, the number of non-modification-revealing tests, we can have an computable upper bound on MR, the (unknown) number of modification-revealing tests. What we have is a practical way, an engineering way, a way to work safely in the presence of unknown data. Referring back to Figure 3, the technique and the associated measures, are trying to enlarge the "bubble" that is labeled `nmr`.

4 A New Technique

An exclusion-based approach is likely to be more effective than an inclusion-based approach in two ways.

First, in terms of safety, as it is possible to more confidently identify all non-modification revealing tests where it is not possible to confidently identify all modification revealing tests. Second, in terms of the impact of this approach, as changes are typically small with respect to the size of the entire system, consequently the modification revealing test set is also proportionally small. As the set of non-modification revealing tests is thus proportionally large, even small additions to its size will have a large payoff. Reducing the number of regression tests, by excluding ones that will show no change, can have immediate practical impact. Thus, determining ways to exclude tests should have a large payoff.

The technique is reasonably straight-forward:

1. *Decompose and Reduce System Version n.* Construct the decomposition slices for the system under consideration. Note that the decomposition slice graph does not need to be constructed (although it would be nice to have); we only need the decomposition slices and the slices equivalences.

2. Match Tests with Code. Use techniques like those of Vokolos and Chen to connect test cases to decomposition slices. Note that this yields a decomposition of the test cases, just as the slices form a decomposition of the software; an amalgamation of of equivalent test cases are also obtained by matching the test cases with the equivalent decomposition slice cluster.

3. *Decompose and Reduce System Version* $n+1$. As in step 1. The systems will readily compare. Obtain the tests for decomposition slice clusters that remain unchanged.

4. *Use tests that remain after removing those obtained in step 3.* Any tests for unchanged code are not needed.

The advantage to the tester of partitioning both the code and the tests should not be underestimated. A test that satisfies any testing criteria for one element in the decomposition slice cluster will satisfy the same criteria *for every slice in the cluster.*

What we have done is similar to Vokolos in that we match test to code, but in our case, we use the computational element embodied in the decomposition slice. The advantage of slicing is a clear win here. A code block may be changed that has a ripple effect out into the computation contained in another file. Slicing will locate the affect. The same argument applies to the techniques of Chen.

We use slicing in a much simpler way than Binkley. We do not need to roll-out the code and look for semantic differences. We can statically examine the code (Binkley also does his analysis statically), perform some elementary set operations, and have the result.

RH96 notes that any technique that does not account for the effects of new and deleted code cannot be safe. Our technique accounts for deletions and additions. This fact, in and of itself, does not demonstrate safety. It does make our technique safer than those that do not account for additions and deletions.

4.1 Measuring

Using this method to obtain regression meshes nicely with measurement concepts presented is Section 3.2. We know how many tests we have; we can count the size of the set of tests that we do not use; this set is nmr in Figure 3. We can measure the e-safety of any other selection technique with respect to that number of nmr-tests. In the technique we have presented, we do not even bother to look for mr-tests. We just use what is left as an approximation to MR. However, with an approximation of MR in hand, we can compare empirically the various regression test selection techniques.

Our technique can also be "inverted." Instead of locating the tests for the unchanged decomposition slice cluster and tossing those out of the regression test suite, we can locate the tests for the changed clusters and investigate the idea that they are sufficient as a regression test suite.

Essentially, we have invented another kind of inclusiveness and exclusiveness, as RH96 pointed out and we noted at the end of Section 2.1.

5 Examples

RH96 has some examples: some revolve around statement deletion which is a singularly difficult problem in regression test selection; others are pathological in the sense that they help clarify the technical definitions. We go through these examples in turn to demonstrate our technique.

We begin by noting that the change noted in Figure 1 would have its tests included in the regression suite, because the code has changed. This is just as the techniques of Vokolos and Chen.

A similar argument applies to the code of Figure 4. RH96 notes that both fragments output 1 on input 0, while on input -2, the left fragment outputs 1 and

```
while ( ++x < 0 )
 {
  while ( ++x < 0 )
   {
    while ( ++x < 0 )
     {
     }
  }
  print ( x )
```

```
while ( ++x < 0 )
  {
   while ( ++x < 0 )
    {}
   while ( ++x < 0 )
    {}
   while ( ++x < 0 )
    {}
  }
 print ( x )
```

Figure 4: "Pathological" `while` statements.

```
if P then
 a = 2
end
```

```
if P then
  a = 2
  b = 3
end
```

Figure 5: Adding Code

the right fragment outputs 3. A functional regression test method would be imprecise if the first test were selected. Our slice and test method appropriately selects all tests for this fragment.

The code of Figure 5 nicely illustrates our technique. In the (unlikely) scenario that the only change to the system is the addition of the assignment `b = 3`, the tests that exercise the fragment on the right would be eliminated, as they can have no effect on `b`. In decomposition slicing terminology, the fragment on the right properly extends the decomposition slice on `P`; the decomposition slice on `a` is not involved in the change. It is the maintainers job to see that the new code is correct according to its new specification; this task is not part of regression testing.

The code of Figure 6 is interesting. From its evident context, the pair of decomposition slices for the two function calls do not interfere. The tests that exercise the call to `PutTermGFX()` would not be excluded, and therefore run; the tests that exercise the (evidently) unmodified function `DrawLine(x,y)` would be excluded, and not used. This is a strict application of the technique; in reality, one can presume that the `DrawLine(x,y)` function accesses information that is processed by `PutTermGFX()`. (It is hard to image a drawing function not accessing the terminal.) Because the the decomposition slices would show the computational relationship between these functions, our technique would succeed.

```
call PutTermGFX()
call DrawLine(x,y)
```

```
call DrawLine(x,y)
```

Figure 6: Deleted Function Call

```
if P then
 do_something
 a = 2
end
if Q then
 do_something
 print a
end
```

```
if P then
 do_something
end
if Q then
 do_something
end
```

Figure 7: Changed Decomposition Slice

The example of Figure 7 is a modification to the decomposition slice on `a`. Tests that exercise the remaining code on the right are included, as the decomposition slice on `a` was modified.

6 Summary and Future Work

There are five main areas of novelty in this presentation:

1. Exclusion. Previous work in the area has used test inclusion rather than test exclusion as a basis for reducing regression test sets. We believe that our novel approach is significantly more effective at reducing the size of regression test sets than previous approaches.

2. Decomposition slicing. No previous work has used decomposition to direct regression testing. Decomposition slicing uniquely exploits computational equivalence to safely identify reduced testable components. This means that a decomposition slicing directed approach to determining excluded test sets allows more precision in test set reduction than previous approaches.

3. Additions and deletions. Both additions and deletions are correctly accounted for.

4. Extended analysis framework. Our work extends the measurement framework proposed originally by Rothermel and Harrold in RH96.

5. New practical metrics. We are able to establish an empirical baseline for the comparison of regression test selection techniques.

This approach can be used very quickly and easily by commercial companies to reduce their regression test sets.

Continuing Work

The Software-artifact Infrastructure Repository (SIR) [6] is a repository of software-related artifacts designed to support rigorous, controlled experimentation with program analysis and software testing techniques. It was created to support the understanding and assessment of software testing and regression testing techniques and as such is an invaluable resource for our continuing work. The repository contains many Java and C software systems in multiple versions, together with supporting artifacts such as test suites, fault data, and scripts. It also includes documentation on how to use these objects in experimentation, supporting tools that facilitate experimentation, and information on the processes used to select, organize, and enhance the artifacts, and supporting tools that help with these processes. We will use the SIR as an empirical test-bed.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[3] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance*, 13(4):263, 2001.

[4] Y.-F. Chen, D. Rosenblum, and K.-P. Vo. Testtube: A system for selective regression testing. In *16^{th} International Conference on Software Engineering*, pages 211–220, 1994.

[5] M. Cusumano. What road ahead for Microsoft and Windows? *Commun. ACM*, 49(7):21–23, 2006.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[7] N. Fenton and S. L. Pfleeger. *Software Metrics: a rigorous and practical approach*. PWC, Second edition, 1997.

[8] K. Gallagher and D. Binkley. An empirical study of computation equivalence as determined by decomposition slice equivalence. In *Proceedings of the 10th Working Conference on Reverse Engineering, WCRE–03*, pages 316 – 322, 2003. ISBN: 0-7965-2087-8.

[9] K. Gallagher, M. Harman, and S. Danicic. Guaranteed inconsistency avoidance during software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(6):393–415, 2003. ISSN: 1532-060X.

[10] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8), August 1991.

[11] T. Ostrand, E. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[12] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[13] F. Vokolos and P. Frankl. Empirical evaluations of the textual differencing regression testing technique. In *IEEE Conference on Software Maintenance, 1998*, pages 44–53, 1998.