Incremental Dynamic Impact Analysis for Evolving Software Systems

James Law
Computer Science Dept.
Oregon State University
Corvallis, OR
law@cs.orst.edu

Gregg Rothermel
Computer Science Dept.
Oregon State University
Corvallis, OR
grother@cs.orst.edu

Abstract

Impact analysis – determining the potential effects of changes on a software system – plays an important role in helping engineers re-validate modified software. In previous work we presented a new impact analysis technique, `PathImpact`*, for performing dynamic impact analysis at the level of procedures, and we showed empirically that the technique can be cost-effective in comparison to prominent prior techniques. A drawback of that approach as presented, however, is that when attempting to apply the technique to a new version of a system as that system and its test suite evolves, the process of recomputing the data required by the technique for that version can be excessively expensive. In this paper, therefore, we present algorithms that allow the data needed by* `PathImpact` *to be collected incrementally. We present the results of a controlled experiment investigating the costs and benefits of this incremental approach relative to the approach of completely recomputing prerequisite data.*

1. Introduction

Successful software systems evolve, and as they evolve, the changes made to those systems can have unintended, expensive, or even disastrous effects [13], Software change impact analysis is a technique for predicting the potential effects of changes before they are made, or measuring the potential effects of changes after they are made, with the goal of reducing the costs and risks associated with releasing the changed software into the field, and helping ensure that the system remains dependable and reliable. In particular, change impact analysis can be used to predict or identify portions of a system that will need to be retested as a result of changes. The results of this analysis can be used to determine whether the potential impact of changes on re-testing effort is low enough to allow the changes to be made, or to guide test engineers in determining how to allocate their regression testing efforts on the modified system.

The importance of the impact analysis problem has led many researchers to propose specific impact analysis techniques [2, 3, 14, 16, 21]. In [8] we present a new impact analysis technique, `PathImpact`, that differs from these previous impact analysis techniques in its reliance on dynamic program behavior information gathered for a specific set of executions (e.g, relative to operational profiles and/or specific test suites), coupled with its reliance on relatively high-level, low-cost (e.g., the level of procedure invocations and returns) system execution information. Empirical comparisons of `PathImpact` to two other prominent techniques (transitive closure on call graphs and static slicing) [8], show that it can obtain significantly greater recall than the former, and significantly greater precision, with greater efficiency, than the latter.

Impact analysis can be expensive in terms of time and computational effort. `PathImpact` is lower-cost than many other impact analysis techniques due to its focus on relatively high-level execution information. For the purposes of this investigation we focus on procedure calls and returns. Even so, following system modifications — including both code modifications and changes made to test suites in response to those modifications — time constraints may make it difficult to re-collect the data needed by the algorithm. This can be the case even after only small modifications to code and test suites have been made. To employ `PathImpact` cost-effectively across entire system lifetimes, techniques are needed to efficiently update the data required by the algorithm as those systems and their test suites evolve.

In this paper we present such techniques. Our approaches take the form of recursive graph algorithms, which together handle the various types of program and test suite modifications that need to be accommodated to keep the data required for the application of `PathImpact` up-to-date and available without resorting to complete recomputation from scratch. Specifically, these algorithms respond to the addition, modification, or deletion of system components, or to the addition, modification, or deletion of test cases from the system's test suite. (We use the term "components" here generically to refer to parts of a software system, such as procedures, methods, or classes, rather than specifically in the sense of reusable code or COTS components.) We present the results of a controlled experiment investigating the costs and benefits of using this incremental approach, relative to the alternative approach of completely recomputing prerequisite data, that illustrates the tradeoffs between the approaches.

2. Background

In this section we review related work on impact analysis, and describe our `PathImpact` approach.

2.1 Impact Analysis and Related Work

Impact analysis techniques can be partitioned into two classes: those based on traceability analysis and those based on dependence analysis [3]. Our work focuses on the latter.

Dependence-analysis-based impact analysis techniques [9, 10, 11, 19] attempt to assess the effects of change on semantic dependencies between program entities, typically by identifying the syntactic dependencies that may signal the presence of such semantic dependencies [17]. The techniques used to identify these syntactic dependencies typically include static (e.g., [4, 22]) and/or dynamic (e.g., [1, 6]) slicing techniques. (For an overview of slicing techniques see [5, 20]). Other techniques using transitive closure on call graphs [3] attempt to approximate slicing-based techniques, while avoiding the costs of the dependence analyses that underlie those techniques.

Other common attempts to approximate dependence-based impact analysis approaches have involved expert judgement and code inspection; however, these are not easily automated. Moreover, expert predictions of the extent of change impact have been shown to be frequently incorrect [12], and performing impact analysis by inspecting source code can be prohibitively expensive [16].

2.2 The PathImpact Approach

Existing, dependence-based approaches to impact analysis have both advantages and disadvantages. These advantages and disadvantages are discussed in detail in [8], we summarize them here:

- Transitive closure on call graphs is relatively inexpensive, but it can also be inaccurate, claiming impact where none exists and failing to claim impact where it does exist.
- Static slicing can predict change impact conservatively (safely); however, by focusing on all possible program behaviors, it may return impact sets that are too large, or too imprecise relative to the expected operational profiles of a system, to be useful for maintainers.
- Dynamic slicing can predict impact relative to specific program executions or operational profiles, which may be useful for many maintenance tasks, but it sacrifices safety in the resulting impact assessment.
- Static and dynamic slicing techniques are relatively computationally expensive, requiring data dependence, control dependence, and alias analysis, whereas call graph based analysis is comparatively inexpensive.
- Finally, all three approaches rely on access to source code to determine the static calling structure or dependencies in that code, and require a relatively large effort to recompute the information needed to assess impact on subsequent releases of a software system.

These advantages and disadvantages motivated the creation of our impact analysis technique, `PathImpact`, which uses relatively low-cost instrumentation to obtain dynamic information about system execution, and from this information builds a representation of the system's behavior which it uses to estimate impact.

2.2.1 Summary of the Approach

The `PathImpact` approach relies on lightweight instrumentation to collect information from a running software system. The information is collected in the form of multiple execution traces in the form shown in Figure 1. Each trace lists each method encountered, and each return taken (symbol 'r'), and the system exit (symbol 'x'), in the order in which they occur during execution. These traces are processed by a compression algorithm called `SEQUITUR` [7] into a space-efficient whole-path directed acyclic graph, or whole-path DAG, such as the one shown in Figure 2. `SEQUITUR`, however, is an "online" algorithm: the sequence of traces can be processed as they are collected, obviating the need to store large amounts of trace data. The resulting whole-path DAG has been shown to achieve compression by a factor of approximately 20 (a 2GB trace reduced to a 100MB whole-path DAG[7]).

`PathImpact` walks the DAG to determine an impact set given a set of changes. The complete algorithm for performing this DAG walk is presented in [8], together with examples of its operation and empirical results on its use, and due to space limitations we cannot present it in detail here. One way to visualize its operation, however, is to consider beginning at a changed component's node in the DAG, and ascending through the DAG performing recursive forward and backward in-order traversals at each node, and stopping when any trace termination symbol is found. By

M B r A C D r E r r r r x M B G r r r x M B C F r r r r x

Figure 1. Multiple execution traces. Upper case letters denote component names, "r" denotes a return, and "x" denotes a system exit.

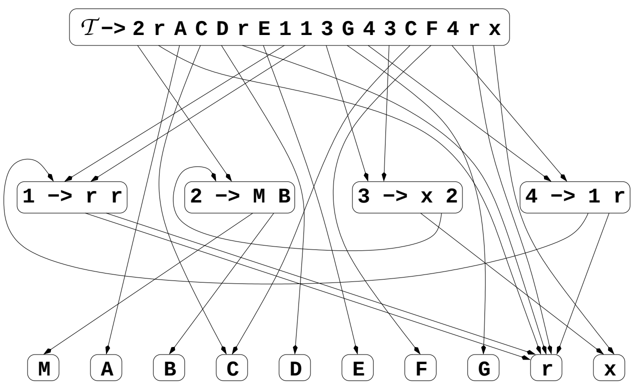

Figure 2. Whole-path DAG created from the execution traces in Figure 1.

traversing forward in the DAG we can find all program components which execute after the change and therefore could be affected by the change. By traversing backward in the DAG we can determine by examination which components execution returns into.

In the case of our example, when component *A* is changed, the resulting impact set is {M, A, C, D, E}: these are all the components into which execution is determined, through the DAG walk, to have encountered, or re-encountered subsequent to the changed component *A*. Note that *B* is not included in the impact set because it returns before *A* is called, and therefore could not be affected by any change in *A*. This can be determined by examining the sequence of nodes during the backwards traveral. Since *B* is immediately preceeded by a return, *B* cannot be returned into, and is excluded from the impact set. This also can be seen by examining the original trace in Figure 1.

2.2.2 Cost-Benefits Tradeoffs for Path-Impact

`PathImpact` provides a different set of cost-benefits tradeoffs than impact analysis techniques based on transitive closure, static slicing, or dynamic slicing – a set which can potentially be beneficial for an important class of impact analysis tasks not accommodated by those techniques. In particular:

- `PathImpact` is fully dynamic, requiring no static analysis of the system. The resulting impact estimate is thus not safe — it cannot predict all possible impacts that can occur under all possible inputs. However, because this estimate is drawn on operational behavior, it can be used in cases where safety is not required to provide more precise information relative to a specific operational profile or set of test executions than static analysis can provide.

- `PathImpact` is call-based, and thus identifies impact at a coarser level than that provided by fine-grained analyses, but it is much more precise than call graph based analysis, and requires no dependence analysis.

- The instrumentation on which `PathImpact` is based has a relatively low overhead, and can be performed on binaries, so the technique does not require access to source code.

3. Incremental, Dynamic Whole Program Path-Based Impact Analysis

When a software system is modified, the execution traces collected for the previous version of the system, and any impact-analysis-facilitating representation built from those traces, may become incorrect. Even with lightweight instrumentation, the effort to regenerate dynamic information, in cases where this means reexecuting all the tests and building a completely new impact representation, can exceed available resources. In such cases it may be preferable to identify the outdated or incorrect information, remove it, and incrementally generate new information where required.

To create such an incremental approach for our `PathImpact` technique, we need to consider two possible sources of modifications that can be made to the software system and its related information: changes to the test suite, and changes to system components. In the remainder of this section, we overview our approach to handling these types of changes, and then in the next section we provide the algorithms that comprise this approach.

Where changes to the test suite are concerned, when maintainers remove a test case, the whole-path DAG then contains an invalid trace, which must be removed from the DAG. Likewise, if a test case is modified, the existing trace must be removed from the DAG, the test case must be reexecuted, and its new trace must be added to the DAG. Finally, traces for any new test cases must also be added to the DAG.

Next, consider changes to system components. Changes to components may involve code changes, configuration changes, environmental changes, or other changes that affect the runtime behavior of the component. If a component is removed from the system, each trace that includes that component must be removed from the DAG. If a component is modified, or the conditions that may affect a component's behavior are changed, each trace that could be affected by that modification (as a conservative approximation, each test case that executes the component) must be removed. In both of these cases, test cases must be reexecuted and their new traces added to the DAG. Finally, addition of new components must also be accommodated.

Two of the foregoing cases can be handled relatively easily. First, addition of new traces, or traces gotten from reexecuting test cases, to an existing DAG is easily performed using the `SEQUITUR` algorithm described in 2.2.1, by simply concatenating the new traces to the end of the existing whole-path DAG. Note further that these existing algorithms ensure that this process works even when the new trace contains procedures not previously listed in the DAG.

Second, addition of new system components affects only traces that reach calls to added procedures. Such traces can be identified by the handling of code modifications and through identification of new invocations of those components (these may be invocations of new components, as well as invocations of components that themselves, transitively, invoke other new components).

Our approach to incremental updating requires tagging individual test executions with unique keys. These keys appear at the beginning of each execution trace, while each execution trace ends with a system exit symbol. Any number of such traces can be concatenated and processed by `ModSEQUITUR`, a modified version of the `SEQUITUR` algorithm, which we present in Section 4.1.

Given these unique execution keys, individual traces for deleted or modified test cases can be removed from the whole-path DAG using the algorithm we present in Section 4.2. Test cases that have simply been modified are then reexecuted and their traces added to the DAG using the process for handling new test cases.

The remaining problem is how to accommodate deletions or modifications of system components. If a component is removed from the system, each execution trace containing that component must be removed from the DAG. If a component is modified, each trace containing that component must be removed, each test corresponding to a removed trace must be reexecuted, and the new traces must then be added to the DAG.

In [8] we outlined an algorithm for removing traces containing a given component from a whole-path DAG. We anticipated that it would be easy to ascend into the middle of a trace in the DAG and perform delete operations both forwards and backwards to remove a trace. However, the introduction of unique execution keys that index the *end* of traces complicates the removal of traces from the middle by requiring extensive DAG traversal to maintain the test keys needed to support test-level incremental updating. Fortunately, given a component in the system, it is relatively simple to determine which test cases contain that component, and then remove these test cases. In this manner, a change to a component in the system can be reduced to a collection of test case deletions and additions. In Section 4.3 we present the algorithm for determining the appropriate execution keys related to a system change.

An important aspect of our algorithms for handling changes is that, although the DAG resulting from a complete rebuild and the DAG resulting from updating may not have equivalent structures, the results of impact analyses performed on these DAGs will be equivalent. The truth of this statement rests with the fact that both the rebuilt and updated DAGs contain exactly the same execution traces, only the order of the traces may change. The traversal that determines the impact set is insensitive to the order of the traces since the search stops at any execution boundary. Since the same traces are searched, the impact sets are equivalent.

4. Algorithms

We now present our new algorithms, which we refer to collectively by the name `EvolveImpact`, to distinguish them from our earlier `PathImpact` algorithms which did not accommodate updates to the DAG.

First, we use a modification of the `SEQUITUR` data compression algorithm [15], which we call `ModSEQUITUR`, to reduce the size of the traces that are collected and store the required execution keys that allow updating of the DAG.

Our technique requires programs to be instrumented at component entry and exit (generally, this means procedure or method entry). Unique execution keys are added to the trace before each execution begins. This produces a sequence of traces, each of which is composed of a sequence of tokens containing a unique execution key, a sequence of procedure names and function returns, and a program exit, in the order in which they occurred.

In the following Sections we present `ModSEQUITUR`, followed by our algorithms for handling test case removal and procedure removal from the whole-path DAG.

4.1. The `ModSEQUITUR` algorithm

The `ModSEQUITUR` algorithm examines a trace generated by a program and removes redundancies in the observed sequence of events by creating a grammar that can *exactly* regenerate the original trace. The trace may contain loops and backedges. As stated earlier, `ModSEQUITUR` is an online algorithm – this means it can process traces as they are generated, rather than requiring the entire trace to be available. To facilitate re-use of the collected trace, only the resulting `ModSEQUITUR` grammar need be stored.

`ModSEQUITUR` (see Figure 3), uses two hash tables for each grammar rule to store the end point of each trace. The end of each trace is marked by a unique test key encountered at the beginning of each trace. One hash table ($TEnd$) stores the link in the rule's production where a test

```
algorithm  ModSEQUITUR( S )
 input     Execution Trace S
 output    Grammar G

    Grammar G
    Rule 𝒯
    Hash Table TEnd, key = testkey, value = tlink
    Hash Table TLink, key = link, value = list
    Test Key tkey
    for each token in S
      if( token is a Test Key )
        tkey = token
        continue
      append token to end of production for 𝒯
      if( token = systemexit )
        Store tkey, currlink in TEnd.
        Get tlist from TLink using key = currlink
        Append tkey to end of tlist
        Store currlink, tlist in TLink.
      if duplicate digram appears
        if other occurrence is a rule g in G
          replace new digram with non-terminal of g.
        else
          form a new rule and replace duplicate
          digrams with the new non-terminal.
      if any rule in G is used only once
        remove the rule by substituting the production.
    return G
```

Figure 3. `ModSEQUITUR` **algorithm.**

`[a M B r A C D r E r r r r x [b M B G r r r x [c M B C F r r r r x`

Figure 4. Multiple execution traces with test keys.

ends, using the test identifier as a key. The other hash table ($TLink$)stores a list of test identifiers, using the link in the production where the test ends as the key. A list must be stored in $TLink$ because compression may cause multiple test cases to map to the same link in a production rule. Both hashes are necessary, since our algorithms must be able to both efficiently look up where a test ends and check whether a given link has any tests ending at that link. The test key tokens must be unique and must be easily distinguishable from the trace tokens that follow it.

Given the keys and traces shown in Figure 4, the resulting `EvolveImpact` DAG is shown in Figure 5. (In the figure, execution keys are preceded by the "[" character. Empty hash tables are not shown.) Since `ModSEQUITUR` differs from `SEQUITUR` only in constant time operations, it runs in time $O(N)$, where N is the uncompressed trace length [15]. The two hash tables grow with the number of traces rather than the length of the traces. Therefore, the size of the compressed trace is still $O(N)$ worst case (no compression possible) and $O(\log N)$ best case [15]. Programs containing loops may generate extremely long traces; however, loops will introduce redundancies that `ModSEQUITUR` takes advantage of.

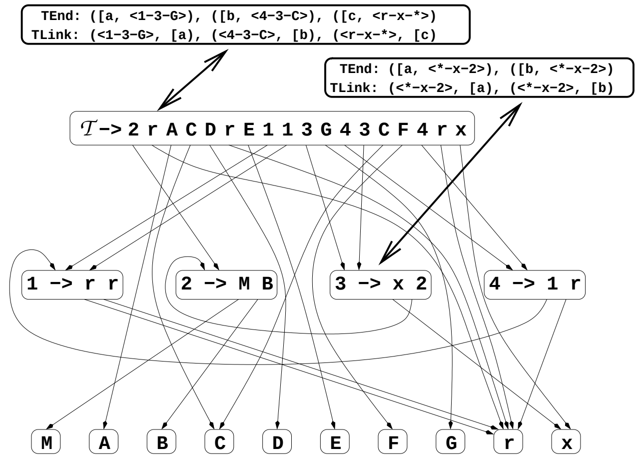

Figure 5. `EvolveImpact` **DAG.**

4.2. Test removal algorithm

The `Remove-test` algorithm (Figure 6) is used to remove a test with key $tkey$ from the DAG. `Remove-test` works by finding the location of the end of the execution with key $tkey$ and then selectively expanding ($\mathcal{T}$), and then (if necessary) expanding the symbol at the beginning of the trace that will be removed. `Expand` (Figure 7) is used to expand a single symbol in the top level rule. After `Expand` creates the expanded representation of a symbol, `Insert` (Figure 8) is used to replace the symbol in $\mathcal{T}$ with the expansion. If the list of tests obtained from $TLink$ contains a predecessor to $tkey$ we know that the entire trace is compressed into this one symbol in $\mathcal{T}$. In this case there is only one symbol to expand, after which we can delete the trace from $\mathcal{T}$. If there is no predecessor in the list, the algorithm searches backwards in $\mathcal{T}$ until it finds the end of the preceding trace and expands this symbol also.

Once the beginning and end of the trace have been located and expanded, the trace is deleted by `Remove-trace` (Figure 9) which starts at the end symbol and deletes predecessors in $\mathcal{T}$ until a system exit is encountered. The intervening symbols are known to be in the trace we are deleting due to the traversal that established the end of the preceding trace. Therefore, we can delete the intervening symbols in the production of $\mathcal{T}$ without any deeper traversal of the DAG.

Expanding a symbol may require traversing the entire DAG, in the worst case. Traversing the entire DAG is equivalent to traversing the uncompressed trace, in $O(N)$ steps, where N is the uncompressed trace length. In the best case, expanding a symbol would require $O(\log N)$ steps. `Insert` has comparable running time since it may be called on to insert an expansion of the entire DAG. Likewise, then `Remove-trace` would have to traverse the fully expanded

```
algorithm Remove-test( tkey )
 input    Test Key tkey
 output   DAG E

     Sequence Exp
     List ExpTestList
 1.  Get tlink from TEnd using key = tkey.
 2.  Get tlist from TLink using key = curr_link
 3.  ExpTestList = tlist
 4.  Remove tkey from TEnd hash table.
 5.  Remove tlink from TLink hash table.
 6.  Find tkey in tlist.
 7.  Exp = Expand( tlink, tkey )
 8.  Insert( tlink, Exp )
 9.  if tkey has a predecessor in tlist
10.      Remove-trace( tkey )
11.      return;
12.  else
13.      plink = predecessor of tlink
14.      while( plink exists )
15.          Get plist from TLink using plink
16.          if plist is not empty
17.              break
18.          else
19.              plink = predecessor of plink
20.          Add plist to beginning of ExpTestList
21.          Exp = Expand( plink, tkey )
22.          Insert( plink, Exp )
23.          Remove-trace( tkey )
24.  while( any rule with zero uses exists )
25.      Remove the zero use rule.
```

Figure 6. `Remove-test` **algorithm.**

DAG. Therefore the running time for `Remove-test` is $O(N)$, and the algorithm may require $O(N)$ space. Since most program paths exhibit some regularity [7], however, this worst case behavior is unlikely.

As an example, consider removing test key "[b" from the DAG in Figure 5. From $TEnd$ we find that $tlink$ points to $< 4 - 3 - C >$. Expanding the symbol *3* gives *x M B*. Inserting gives $\mathcal{T} \rightarrow 2rACDrE113G4xMBCF4rx$. Since there is no predecessor in $tlist$ (line 8 in `Remove-test`), we must search backwards for the end of the previous trace. Moving backwards in $\mathcal{T}$ one symbol at a time, we check for the presense of a non-empty $tlist$ in the $TLink$ hash table. We find a non-empty list at the link $< 1 - 3 - G >$. Expanding *3* and inserting gives the uncompressed trace, from which the trace with key "[b" is easily removed. The last two lines of `Remove-test` attempt to clean up the grammar by removing any rules which are no longer used. However, since a symbol in the grammar may expand into many symbols when inserted, deleting traces generally results in a larger DAG (some loss in compression) after the deletions.

4.3. Procedure removal algorithm

When a procedure is changed or removed from the program we use `Find-Key-List` (Figure 10) to find the test keys corresponding to the traces in which this procedure occurs. `Find-Key-List` begins at the terminal in the DAG representing the procedure or component and calls `Move-up` (Figure 11) to move upwards in the DAG to each rule which uses the component. In each rule, `Move-up` searches forward in each production rule for test keys and adds them to $TKeyList$. If no keys are found in the production, `Move-up` uses `Descend` (Figure 12) to recursively search subtrees in the DAG for test keys. After the keys are found the corresponding traces are removed using `Remove-test`, described in the previous section.

```
algorithm Expand( expLink, tkey )
 input    Link expLink
 input    String tkey
 output   List ExpList

     List ExpList, global, static
     String elem
     Link elink, tlink
     List tlist
     Grammar Rule irule
     irule = rule pointed to by expLink
     if( irule is a terminal )
         Append irule to end of ExpList.
         return
     elem = first symbol in production of irule.
     elink = link in irule production containing elem
     if( elem = system exit )
         Append elem to end of ExpList.
         elink = link in ExpList pointing to elem
         Remove tkey from TEnd and TLink for irule.
     else
         Remove tkey from TEnd and TLink for irule.
         Expand( elink, tkey )
     while( elem has a successor in production )
         elem = successor of elem in production of irule
         elink = link in irule production containing elem
         if( elem = system exit )
             Append elem to end of ExpList
             Remove tkey from TEnd and TLink for irule.
         else
             Expand( elink, tkey )
     return ExpList
```

Figure 7. `Expand` **algorithm.**

```
algorithm Insert( inslink, ExpList, ExpTestList )
 input    Link inslink
 input    List ExpList
 input    List ExpTestList

     List tmplist
     String tmpprod
     String tmptest
     while( ExpList is not empty )
         tmpprod = Remove first element of ExpList
         Insert tmpprod after inslink
         Move inslink to tmpprod
         if( tmpprod = system exit )
             tmplist = new List
             tmptest = Remove first element of ExpTestList
             Add tmptest to tmplist
             Store tmptest, inslink in TEnd for T
             Store inslink, tmplist in TLink for T
```

Figure 8. `Insert` **algorithm.**

```
algorithm Remove-trace( T, tkey )
 input    Rule T
 input    String tkey
 output   Rule T

     List dlist
     Link dlink
     String currentSymbol
     Get dlink from TEnd for T using tkey
     Remove tkey from T TEnd hash table.
     Remove dlink from T TLink hash table.
     currentSymbol = symbol pointed to by dlink
     while( currentSymbol is not a system exit )
       Remove currentSymbol from T production
     return T
```

Figure 9. `Remove-trace` **algorithm.**

```
algorithm Find-Key-List( procedure )
 input    String procedure
 output   List TKeyList

     List TKeyList, global, static
     for each rule where procedure appears in the production
       Move-up( rule, link )
     return TKeyList
```

Figure 10. `Find-Key-List` **algorithm.**

The worst case running time of `Find-Key-List` is also $O(N)$, where N is the uncompressed trace length, since it may also result in traversing the entire DAG.

Consider the effect of modifying procedure F in the DAG in Figure 5. `Find-Key-List` would use `Move-up` to examine each rule that has F in its production, in this case $\mathcal{T}$. `Move-up` uses `Descend` to recursively search forward in $\mathcal{T}$, examining 4, 1, and r, before finding x and encountering a non-empty $tlist$ containing the key "[c". Then, F can be removed from the DAG by removing test key "[c" using `Remove-test`.

5. Empirical Validation

The primary research question we wish to investigate is whether incrementally computing impact analysis information with the `EvolveImpact` algorithms is more efficient than recomputing that information from scratch. We also wish to determine whether this relationship changes with numbers of changes.

We thus performed a controlled experiment comparing the time required to update an initial `EvolveImpact` DAG using our algorithms to remove and add traces to the time required to completely rebuild an `EvolveImpact` DAG, after various numbers of test suite or program changes. We also compared the sizes of the resulting DAGs and traces.

```
algorithm Move-up( rule, link )
 input    Rule rule
 input    Link link

     List nlist
     Stringproc
     Boolean keeplooking
     proc = symbol link points to in production
     while( proc has a successor )
       proc = proc successor
       nlink = link in rule production containing proc)
       Get nlist from TLink using nlink
       if( nlist is not empty )
         Append nlist to TKeyList
         return false
       else
          keeplooking = descend( proc )
          if( keeplooking = false )
            return false
      for each rule where proc appears in the production
        move-up( rule, link )
```

Figure 11. `Move-up` **algorithm.**

```
algorithm Descend( rule )
 input    Rule rule

     String proc
     List tlist
     Link tlink
     if( rule a terminal )
       return true
     else
       proc = first element of production
       tlink = link in production containing proc
       Get tlist from TLink using tlink
       if( tlist is not empty )
         Append tlist to TKeyList
         return false
        else
          if( descend( proc ) == false )
            return false
          while( proc has a successor in production )
            proc = proc successor in production )
            if( descend( proc ) == false )
              return false
      return true
```

Figure 12. `Descend` **algorithm.**

5.1. Experiment materials

As a subject we used a program developed for the European Space Agency; we refer to this program as `space`. `Space` is an interpreter for an antenna array definition language (ADL). `Space` has approximately 6200 non-blank, non-comment lines of C code and 136 functions.

To provide input sets for use in determining dynamic behavior, we used test cases and suites created for `space` for earlier experiments [18]. The test cases consisted of a pool of 10,000 randomly generated test cases, plus 3,585 test cases generated by hand to ensure coverage of every branch in the program by at least 30 test cases. This pool of test cases was used to generate 1000 branch coverage adequate test suites containing on average approximately 150 test cases each. We randomly selected 50 of these test suites. The selected test suites in aggregate contained 7,773 test cases including 2,886 duplicates.

We implemented the `EvolveImpact` algorithms in the Java language, using the earlier `PathImpact` implementation as a starting point. The `EvolveImpact` implementation consists of 35 classes and approximately 9700 lines of non-comment code.

5.2. Experiment methodology

5.2.1 Measures

To measure the efficiency of the various algorithms, we collected measures of (1) the time required to collect an initial set of traces and build an initial DAG, (2) the time required to reexecute the changed program or test suite and rebuild another DAG from scratch, and (3) the time required to incrementally update the initial DAG by removing and adding traces using our new algorithms. When testing the `EvolveImpact` algorithms with program modification, we recorded how many test keys were removed and subsequently added when updating the initial DAG. We also collected the size in bytes of each trace and each resulting DAG.

All measures were collected on five identical Sun Ultra 5 workstations. Each workstation had a 360MHz SPARC processor, 256 MBytes of main memory, 1 GByte of swap space, and used 100 MBit ethernet attached storage. There were no other users on the machines during the course of gathering information for this experiment.

Elapsed time information was collected using the standard Java library system function which returns the current system time in milliseconds. The sizes of the traces and the DAGs was measured after the trace or the DAG had been stored in a disk file.

5.2.2 Design

In this experiment, the three measures of time in milliseconds and six measures of size in bytes are our independent variables. Our dependent variables are the technique applied (rebuild from scratch versus incremental), and either the number of changes in the test suite or the number of procedures in the program that were modified.

We ran the experiment in two stages. The first stage examined the effect of changes in test suites. For each test suite T in our set of fifty test suites, we built the initial DAG for T. We then chose a random number, k, between one and the number of test cases in T, and then, k times, chose randomly whether to add or delete a test case to or from T. Added test cases were randomly drawn from the pool of available test cases, and deleted test cases were randomly extracted from the test cases currently remaining in T, excluding any of the test cases just added. The resulting modified test suite was then used to rebuild a DAG from scratch. The updated DAG was created by deleting the test cases that were removed from the test suite from the initial DAG, executing the added test cases to create a trace file, and then adding this trace to the updated DAG. We performed this sequence of operations 15 times for each of our 50 test suites.

In the second stage of the experiment we examined the effect of making changes to procedures in `space`. For each of our test suites T, after building the initial DAG for T, we selected a random number of procedures in `space` (between one and four), found the list of test cases in the DAG containing these procedures using `Find-Key-List`, removed the listed test cases from the DAG, reexecuted the test cases, and added the new traces to the DAG. We also rebuilt the DAG from scratch, as in the previous stage. Note that in this case, since the test suite was unchanged, and there were no actual modifications made to `space`, the rebuilt DAG exactly recreated the initial DAG. While this would not always be the case in practice (changes to procedures may or may not alter test traces) when procedures have been modified, this process provided a necessary experimental control, allowing us to avoid conflating the effects of procedure changes with the associated test case change effects. We also used the rebuilt DAG to provide a comparison and consistency check against the initial DAG. As in the first stage, we repeated this sequence 15 times for each test suite.

5.2.3 Threats to validity

Like any controlled experiment, this experiment faces threats to validity, and these must be controlled for, or accounted for when interpreting results.

Threats to internal validity concern our ability to properly draw conclusions about the connections between our independent and dependent variables. In this experiment, our primary concern involves instrumentation effects, such as errors in our algorithm implementations or measurement tools, which could affect outcomes. To control for these threats we carefully validated these implementations and tools on known examples.

Threats to external validity concern our ability to generalize our results to larger classes of subjects. In this experiment we have studied only one program, and although it is a “real” program we cannot claim it is representative of programs generally. Also, the test suites we use represent only one type of suite that could be found in practice. Finally, the program and test suite modifications were simulated, not actual, modifications. Additional studies of other

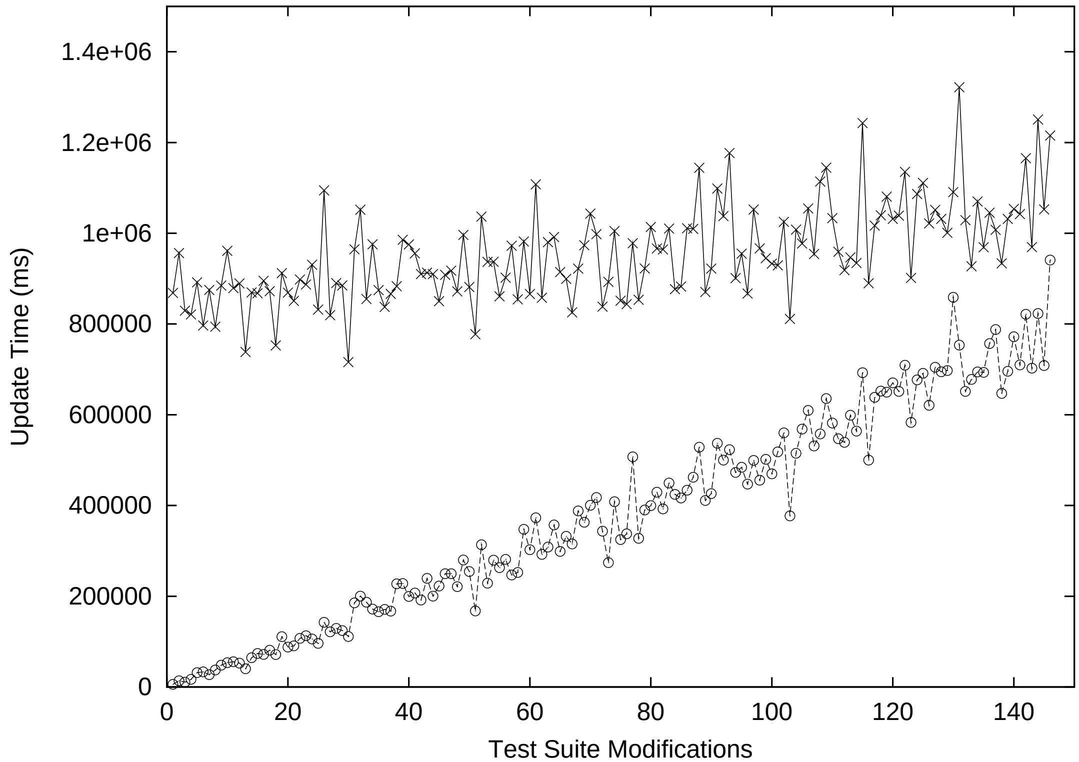

Figure 13. Rebuild times and Update times in milliseconds for test suite modifications. The average time required to rebuild the DAG for each number of modifications is represented using an "x". The average update time is shown as a circle.

programs and types of inputs are needed to address such threats to external validity.

Threats to construct validity concern the appropriateness of our measures for capturing our dependent variables. Our measures of time and space capture one aspect of these variables, but other factors (e.g. the amount of memory rather than disk consumed) may be important in particular situations. Also, Space executes very quickly, and its execution time contributes little to the initial, rebuild, and update times. This causes algorithm run time to play a more important role in comparisons of techniques than it would for a program requiring longer execution times. Longer execution times, however, would be expected in general to increase the benefit gained by the incremental algorithms.

5.3. Analysis and results

We now present our results, beginning by presenting and discussing data. Section 5.4 discusses implications.

5.3.1 Data

In the first stage of our experiment, over each of the 50 test suites, we measured three times in milliseconds, six sizes in bytes, and the number of changes made to the test suite. The time required to build the initial DAG was collected as an experimental control and consistency check. (Since the initial DAG would be created in practice regardless of the decision to rebuild or update the DAG after changes, its creation time and size has no effect on the research question we are investigating.)

The graph in Figure 13 provides a view of the rebuild and update time measurements collected during the test suite modification stage. Each data point represents the average time in milliseconds for 15 trials for each number of random test suite modifications.

Stage two of our experiment simulated changes to procedures in `space`. The number of program modifications was randomly chosen between one and four, inclusive. However, since the number of modifications is transformed into a list of test cases to remove, we reasoned that the number of test cases removed from the DAG was a more reliable indicator of the extent of changes made for each trial. We refer to the number of test cases that were removed as the "Test Change Factor", or TCF. Figure 14 shows a box plot of the number of modifications made to the program compared to the observed TCF for 750 trials. In general, the more pro-

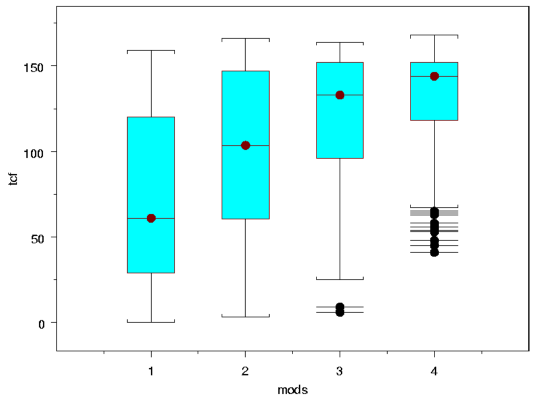

Figure 14. **Number of program modifications versus Test Change Factor (TCF).**

cedures that were modified, the more test cases in the test suite were affected. However, it was not uncommon for a modification of a single procedure to affect the majority of the test cases in a test suite.

Finally, Figure 15 shows TCF versus the rebuild and update times for the procedure modification stage of the experiment.

5.3.2 Analysis of time differences

When responding to test suite modifications, the average time required to update the DAG is less than the average time required to rebuild the DAG across the entire range of test suite modification sizes. The average rebuild time was 958,156 milliseconds with a standard deviation of 172,616 milliseconds. The computational effort required to update the DAG after test suite modifications appears to be linear in the number of test cases added to or removed from the DAG under the circumstances in which we collected our data. Rebuild time grows closer to update time as test suite size increases, but never reaches it, even as maximum test suite size is reached.

The time required to update the DAG after procedure modifications shows a more complex relationship in relation to rebuild times. The average rebuild time was 830,417 milliseconds with a standard deviation of 95,340 milliseconds. For program modifications that affect fewer than approximately 25 test cases in the DAG, updating the DAG required less time than recomputing the DAG. For large TCF values, the computational effort required by the two approaches appears relatively equal. Between a TCF of 25 and approximately 145, however, updating the DAG was consistently (with one outlier, a TCF of 33 having an average update time of 668,389 milliseconds versus an average rebuild time of 830,653 milliseconds) less efficient than recomputation. Update times were also relatively variable in this range, most likely due to variation in the internal compression of the traces in the DAG. In areas of high compression, less graph traversal is required when searching for the appropriate test keys.

5.3.3 Analysis of size differences

We present the size measurements for the test suite modification stage of our experiment first, followed by the size measurements for the program modification stage.

The average size of the initial traces generated by the test suites during the first stage of the experiment was 982,460 bytes, with a standard deviation of 49,221 bytes. The average size of the initial DAG was 80,685, with a standard deviation of 3,967 bytes. The average size of the initial DAG relative to the initial trace was 7.78%. The standard deviation was 0.57%.

The recomputed DAGs in the first stage had an average size of 81,906 bytes, and standard deviation of 5,197 bytes. The traces used to rebuild the DAGs had an average size of 1,039,282 bytes with a standard deviation of 90,874 bytes. The relative size of the DAG compared to its corresponding trace was 7.74%, with a standard deviation of 0.56%.

Updated DAGs had an average size of 92,832 bytes with a standard deviation of 8,220 bytes. The updated DAG was on average 1.13 times the size of the rebuilt DAG, with a standard deviation of 0.085. Since the size of the corresponding trace reflects only test cases that were added, a better relative comparison of the compression obtained can be obtained by observing the size of the updated DAG in relation to the size of the traces used to rebuild the DAG. In this case the updated DAG was 8.44% the size of the traces used to rebuild the DAG, and had a standard deviation of 0.72%.

For our experiments with program modifications, the average size of the initial DAG was 80,651 bytes, and had a standard deviation of 4,407 bytes. The average size of the initial traces was 982,461 bytes with a deviation of 49,221 bytes. On average the size of the DAG was 7.80% the size of the trace, with a standard deviation of 0.57%.

The traces used to recompute DAGs, and the recomputed DAGs, exactly matched the size and variation of the initial traces and DAGs, since they were based on the same test suite and the run-time behavior of `space` did not change.

The updated DAGs had an average size of 103,623 bytes and a standard deviation of 23,833 bytes. In comparison to the recomputed DAGs, the updated DAGs were 1.28 times larger, with a standard deviation of 0.290. The updated DAGs were 10.0% the size of the traces used to create the recomputed DAGs, with a standard deviation of 2.4%.

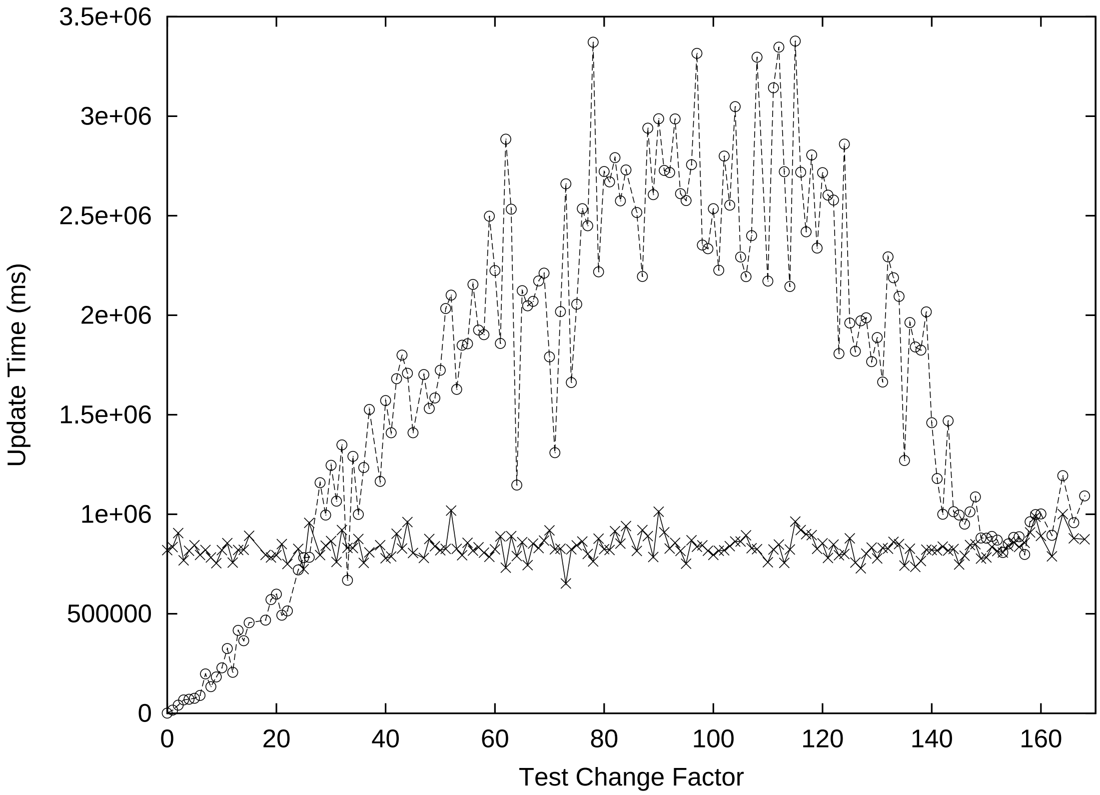

Figure 15. Rebuild times versus Update times in milliseconds by TCF. The average time required to rebuild the DAG for each TCF is represented using an "x". The average update time is shown as a circle.

5.4. Discussion

Our primary goal was to compare the time required to update the DAG to the time required to recalculate the DAG. In all observed cases, it was more efficient to update a DAG than to regenerate it when the program's test suite was modified. The computational effort measured by elapsed time grows slowly with the number of changes made to the test suite, never becoming equivalent to the recomputation effort even when all test cases in the suite are modified.

When a procedure or procedures in the program were modified, results varied with the number of test cases containing the modified procedures. In `space` we found it is not uncommon for one procedure to be hit in a majority of the test cases. The computational effort required to update the DAG was less than the effort of recomputation only for program modifications that affect approximately 25 test cases or fewer. One possible explanation for this result is that `Find-Key-List` may perform more graph traversal operations than expected, or that the traversal may be more computationally expensive than was apparent, above a certain modification threshold.

An additional goal of our experiments was to measure the absolute and relative sizes of the traces `space` generated and the `EvolveImpact` DAGs constructed from them. The sizes of the traces and DAGs was consistent and showed little variation. A compression factor of approximately 10 was consistently observed. Updating the whole-path DAGs caused only minor loss of compression.

6 Conclusions and Future Work

In this paper, we have presented algorithms that allow our `PathImpact` approach to dynamic impact analysis to be applied incrementally as a system evolves, processing the set of test case or procedure modifications made to a program, and avoiding the overhead of completely recomputing the information needed for impact analysis.

Our empirical study of these algorithms suggest that changes in a test suite can be accommodated more efficiently using `EvolveImpact` than by rebuilding the DAG from scratch. The results also suggest that when procedures are changed, the cost-effectiveness of the approach will vary depending on the number of test cases which reach those procedures. After a system change in `space`, it becomes more efficient to rebuilt the DAG than to use the `EvolveImpact` algorithms for a relatively small number of affected tests. This may be due to the behavior of `space` and the particular test cases we used in our experiment, but could be expected to hold for other workloads too. An immediate point of investigation is to find a practical metric with which to determine when this threshold is reached, both by extended experimentation with `space` and by examining the behavior of other programs. We will also investigate the possibility of creating a more efficient algorithm for removing components from the DAG, one that removes components directly rather than using `Find-Key-List` and then removing each test.

Future work includes experimentation with a wider range of programs, an examination of the sensitivity of our techniques to test input data, and consideration of object-oriented and distributed programs. While this research has investigated the application of an approach at the level of a procedure, we intend to apply this model to other levels, such as components. We intend to investigate scalability to large systems by adapting the local level of modeling according to various criteria, such as local usage or complexity measures.

We also plan additional empirical comparisons with conventional dynamic slicing techniques. While typical dynamic slicing techniques operate at a different level (statement-level versus procedure-level) than our techniques, the behavior of impact analyses based on dynamic slicing may be more directly comparable to our techniques than analyses based on other approaches such as static slicing, and the techniques may offer different cost-benefits tradeoffs.

Acknowledgements

This work was supported in part by the NSF Information Technology Research program under Award CCR-0080900 to Oregon State University. Phyllis Frankl, Alberto Pasquini, and Filip Vokolos provided the `space` program and randomly generated tests. A special thanks to Chengyun Chu for creating the additional tests for `space`.

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings: SIGPLAN '90 Conference on Programming Language Design and Implementation. SIGPLAN Notices.*, pages 246–56, White Plains, June 1990. ACM.

[2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the International Conference on Software Maintenance*, pages 292–301, Montreal, Que, Can, Sept. 1993. IEEE.

[3] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):27–60, Jan. 1990.

[5] M. Kamkar. An Overview and Comparative Classification of Program Slicing Techniques. *Journal of Systems Software*, 31(3):197–214, 1995.

[6] B. Korel and J. Laski. Dynamic slicing in computer programs. *Journal of Systems Software*, 13(3):187–95, 1990.

[7] J. Larus. Whole Program Paths. In *Proc. SIGPLAN PLDI 99*, pages 1–11, Atlanta, GA, May 1999. ACM.

[8] J. Law and G. Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *ICSE 2003*, Portland, Oregon, May 2003. IEEE.

[9] M. Lee. *Change Impact Analysis of Object-Oriented Software*. Ph.D. dissertation, George Mason University, Feb. 1999.

[10] M. Lee, A. J. Offutt, and R. T. Alexander. Algorithmic Analysis of the Impacts of Changes to Object-oriented Software. In *TOOLS-34 '00*, 2000.

[11] L. Li and A. J. Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of the International Conference on Software Maintenance*, pages 171–184, Monterey, CA, USA, Nov. 1996. IEEE.

[12] M. Lindvall and K. Sandahl. How well do experienced software developers predict software change? *Journal of Systems and Software*, 43(1):19–27, Oct. 1998.

[13] J. L. Lions. ARIANE 5, Flight 501 Failure, Report by the Inquiry Board. *European Space Agency*, July 1996.

[14] J. P. Loyall, S. A. Mathisen, and C. P. Satterthwaite. Impact analysis and change management for avionics software. In *Proceedings of IEEE National Aerospace and Electronics Conference, Part 2*, pages 740–747, Dayton, OH, July 1997.

[15] C. Nevill-Manning and I. Witten. Linear-time, incremental hierarchy inference for compression. In *Proc Data Compression Conference (DDC 97)*, pages 3–11. IEEE, 1997.

[16] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, Englewood Cliffs, NJ, 1998.

[17] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–79, Sept. 1990.

[18] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test Case Prioritization: An Empirical Study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, Oxford, UK, Sept. 1999.

[19] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *ACM SIGPLAN – SIGSOFT workshop on on Program analysis for software tools and engineering*, pages 46–53. ACM Press, 2001.

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

[21] R. J. Turver and M. Munro. Early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1):35–52, Jan. 1994.

[22] M. Weiser. Program slicing. In *5th International Conference on Software Engineering*, pages 439–49, San Diego, CA, Mar. 1981. IEEE.