Whole Program Path-Based Dynamic Impact Analysis

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 when a change in one part of a program affects other parts of the program, is time-consuming and problematic. Impact analysis is rarely used to predict the effects of a change, leaving maintainers to deal with consequences rather than working to a plan. Previous approaches to impact analysis involving analysis of call graphs, and static and dynamic slicing, exhibit several tradeoffs involving computational expense, precision, and safety, require access to source code, and require a relatively large amount of effort to re-apply as software evolves. This paper presents a new technique for impact analysis based on whole path profiling, that provides a different set of cost-benefits tradeoffs – a set which can potentially be beneficial for an important class of predictive impact analysis tasks. The paper presents the results of experiments that show that the technique can predict impact sets that are more accurate than those computed by call graph analysis, and more precise (relative to the behavior expressed in a program's profile) than those computed by static slicing.

1. Introduction

A frequent problem with software is that changes to a system, however small, may have unintended, expensive, or even disastrous effects [13]. Software change impact analysis, often called simply *impact analysis*, is a family of approaches for addressing this problem [2, 3, 14, 16, 22].

Impact analysis is often used to assess the effects of a change on a system after that change has been made, but a more proactive approach uses impact analysis to *predict* the effects of change before it is instantiated [3]. For instance, software maintainers may wish to consider several methods for implementing a change and choose one that has the lowest estimated impact on cost or schedule. Predictive impact analysis can allow maintainers to work to a plan, rather than simply deal with consequences.

Impact analysis techniques can been partitioned into two classes: traceability analysis and dependency analysis [3]. We focus on the latter. Impact analysis techniques based on dependency analysis [1, 5, 6, 7, 9, 10, 11, 19, 23] attempt to assess the affects of change on semantic dependencies between program entities, typically by identifying the syntactic dependencies that may signal the presence of such semantic dependencies [17].

We focus on three traditional dependency-based impact analysis techniques: call graph based analysis, and static and dynamic program slicing [3].[1] Each of these approaches has advantages and disadvantages:

- Transitive closure on call graphs, cited as a fundamental technique for predicting change impact [3]; is relatively inexpensive; however, it can be highly inaccurate, identifying impact where none exists and failing to identify impact where it does exist.
- Static slicing (see [21] for a summary), 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 profile 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.
- 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 amount of effort to recompute the information needed to assess impact on subsequent releases of a software system.

[1] Other common techniques include expert judgment 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].

In this paper, we present a new technique for performing impact analysis based on *whole path profiling* [8]. Our technique, `PathImpact`, 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.

`PathImpact` provides a different set of cost-benefits tradeoffs than the three techniques outlined above – a set which can potentially be beneficial for an important class of predictive impact analysis tasks. The technique is dynamic, requiring no static analysis of the system. The resulting impact estimate is thus not safe, but because it is drawn only 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 executions than static analysis. The technique 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 the technique is based has a relatively low overhead and can be performed on binaries, so the technique does not require access to source code. The technique also accommodates system evolution at relatively low cost.

We present results of a controlled experiment comparing `PathImpact` to transitive closure on call graphs and static slicing, in relation to a real software system, a large set of changes, and a wide range of inputs and test suites. The results of the experiment show that the technique can predict impact sets that are far more accurate than those computed by call graph analysis, and more precise (relative to the behavior expressed in a program's profile) than those computed by static slicing.

2. Costs-Benefits Tradeoffs for Dependency-Based Impact Analysis Techniques

Call graphs are commonly used by programmers to estimate the potential impact of software change [3]. The underlying assumption is that a change in some procedure p in program P has a potential change impact on any node reachable from p in P's call graph G. The transitive closure of G calculates all potential impact relationships between procedures in P under this assumption.

Call graphs are well understood and relatively easily constructed, and transitive closure on call graphs is simple to perform. However, calling behavior is much more complex than call graphs indicate. Calling behavior may include entering and exiting a procedure without calling any other procedures; entering a procedure, calling one other procedure and exiting; or entering a procedure and making any number of calls to many other procedures in any order. A call graph captures the local structure of potential calls by a single procedure, but ignores these other aspects of calls. As a result, call graph based analysis can lead to imprecise

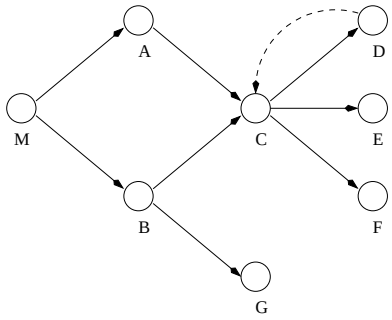

Figure 1. Example call graph. An arc from node *M* to node *A* denotes that procedure *M* may call procedure *A*; the dashed arc denotes a back-edge in the call graph and signals the presence of recursion.

impact sets. For example, in the call graph in Figure 1, *M* transitively calls all other procedures in the program. We cannot determine from the call graph what conditions cause propagation of change impact from *M* to other procedures.

Another limitation of the call graph is that it does not capture impact propagation due to procedure returns. Suppose we propose to change *E*. The effects of the change could propagate through returns to *C*, *B*, *A*, and *M*. A typical call graph has no information on returns, and inferring such information may cause errors. We can infer that *E* returns to *C*; however, we cannot infer whether, following the return to *C*, impact propagates into *A*, *B*, both, or neither.

In contrast to call-graph based techniques, techniques based on static slicing rely on calculation of fine-grained dependency relationships. Static slicing is conservative (safe); it accounts for all possible program inputs and behaviors. When safety is required, this conservatism is important; however, impact analyses that rely on the results may return large impact sets of little use to maintainers. More important, these sets may claim the existence of impacts that do not occur relative to the expected operational profile of the system; for non-safety-critical systems, it may be more cost-effective to consider impact relative to expected usage.

Dynamic slicing overcomes these drawbacks of static slicing by focusing on static dependencies that have been involved in program executions, at the potential cost of safety. Both dynamic slicing and static slicing, however, require fine-grained dependence analysis entailing additional expense in computational support and instrumentation.

Further, fine-grained analysis of dependencies may be inappropriate in many predictive impact analysis situations. Predictive impact analysis requires maintainers to specify the approximate location of changes, but knowing the precise statements that will require modification, and the precise variables that will be involved in those modifications, may not be practical. Further, when modifications involve deleting code, it may not be obvious what slicing criteria to invoke, at the fine-grained level, to predict potential impact.

Working at the coarser-grained level of functions or methods simplifies the impact analysis task, because it requires maintainers to determine only a set of functions or methods that will be changed, and not the precise syntax of the required changes. This suggests that analysis at the level of calls may be more appropriate in practice.

Finally, call-graph based and slicing-based approaches all rely on access to source code to determine the static calling structure of or dependencies in that code. In practice, systems often contain or consist primarily of components for which source code is not available. Approaches for analyzing binaries may lessen this problem, but another alternative is not to analyze them at all, but simply to instrument them, and it is this alternative that we consider.

3. Dynamic, Whole Program Path-Based Impact Analysis

The use of dynamic path information collected at the level of procedure calls helps address the foregoing problems. Our approach can be summarized as follows: if we propose to change procedure p, we concern ourselves only with impact that may propagate down any (and only) dynamic paths that have been observed to pass through p. Therefore, any procedure that is called after p, and any procedure which is on the call stack after p returns, is included in the set of potentially impacted procedures.

By examining program paths, yet limiting impact to observed call orders and call-return sequences, we can improve considerably on the precision of the more simple call graph based techniques, while also approaching the accuracy of more detailed techniques based on static slicing. Also, since program binaries can be instrumented [4, 20], we can avoid requiring program source code. While this approach remains subject to the limitations of dynamic information, it can yield impact sets that are more accurate relative to a specific operational profile.

As an example, suppose we have a single execution trace, shown by the string of letters along the top of Figure 2, for the program whose call graph appears in Figure 1. Function returns are represented in this string by r, and program exit by x. Informally, if we propose to change procedure *E*, we can estimate the dynamic impact of the change relative to this execution by searching *forward* in the trace to find procedures that are called directly or indirectly by *E* and procedures that are called after *E* returns. By searching *backward* in the trace we can discover the procedures *E* returns into. In the case of this trace, *E* makes no calls, but *E* returns into *C*, *A*, and *M*. *E* cannot return into *D* since *D* is immediately followed by a return, and likewise for *B*. Therefore, we can conclude that the set of potentially impacted procedures due to a change in *E* is { M, A, C, E }.

A small amount of bookkeeping is necessary to reconstruct the behavior of the program from a trace such as that

```
M B r A C D r E r r r r x
```

Figure 2. Single execution trace.

```
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 3. Multiple execution traces.

given in Figure 2. Unmatched function returns encountered in the trace determine which procedures are returned into. For example, counting forward from *E* we encounter three more returns (r) than procedure names (including *E*). This indicates that three functions were returned into. Searching backwards we pair the three unmatched returns with any procedure names that do not have associated returns. For instance, searching backward from *E* we look for three procedure names, but skip *D* and *B* because we encounter a return immediately before each of them indicating that they returned before *E* was called. (An alternative approach uses instrumentation that explicitly identifies the procedure responsible for each return. In this case impact can be estimated solely by searching forward in the trace. However, this has disadvantages which we discuss in Section 4.1.)

Multiple execution traces can be processed by concatenating traces, and proceeding in the same manner as for a single trace, but not crossing the termination symbol while searching either forwards or backwards. For example, Figure 3 shows a series of execution traces. If we propose to change *G*, the potentially impacted procedures are *G*,*B*, and *M*. If we propose to change *C* the potentially impacted set is { M, A, B, C, E } from the first trace and { M, B, C, F } from the third trace, giving a union of { M, A, B, C, E, F }.

An obvious difficulty with this approach involves tracking executed paths, since traces of this sort may be long. Therefore, practical techniques must restrict the length of traces to be viable. In the next section we show how methods for compressing traces [15] can be used to greatly reduce the size of the trace, constructing a directed acyclic graph representation called a whole path DAG [8]. Then we present algorithms that calculate impact directly from this DAG. We further show how this DAG can be incrementally updated, facilitating its continued use for impact analysis as a program and its operational profile evolve.

4. Algorithms

Our technique requires programs to be instrumented at procedure (or method) entry and exit. This produces a trace containing procedure names, function returns, and program exits in the order in which they occur across multiple executions. The `SEQUITUR` data compression algorithm [15] is used to reduce the size of the trace that is collected, generating a grammar. The whole path DAG representation developed by Larus [8] is used to represent the grammar. We next

review the SEQUITUR data compression algorithm focusing on its application in our context, and its representation as a whole path DAG; then, in Section 4.2 we present our dynamic impact algorithms and algorithms for incrementally updating the DAG.

4.1. The SEQUITUR data compression algorithm and whole path DAG

The SEQUITUR 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. SEQUITUR 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 SEQUITUR grammar need be stored.

Figure 4 displays the SEQUITUR algorithm.[2] The algorithm constructs a grammar by appending each token seen to the end of rule $\mathcal{T}$ and searching the resulting rule and all other production rules for redundancy. If the tail of the new end of the compressed trace matches any of the production rules the appropriate substitution is made. Then, if any redundancy is found a new production rule is added and substituted in the compressed trace. The algorithm also checks each time a replacement is made to ensure that each production rule is used more than once. If any rule is used just once its production is substituted where it is used and the rule is removed from the grammar. This prevents the retention of rules that do not contribute to compression.

In our case, we apply this algorithm to a set of one or more concatenated traces, each consisting of tokens indicating calls and returns, as shown in Figure 3. Each individual trace is terminated by a special symbol x, generated by the instrumented program on exit. For an example of the SEQUITUR algorithm's operation, consider the trace shown in Figure 3. The rule $\mathcal{T}$ is initially empty:

$\mathcal{T} \rightarrow$

Using the program trace shown in Figure 3 the algorithm begins appending symbols, starting with M and continuing until it finds a duplicate digram:

$\mathcal{T} \rightarrow MBrACDrErrr$

At each step the algorithm checks to see if the last two tokens (a digram) in the string are duplicated (without overlap). In this case rr appears twice, so the algorithm constructs a grammar rule and applies it to $\mathcal{T}$ (we use numbers as tokens to denote production rules, to distinguish them from the tokens in the execution traces):

$\mathcal{T} \rightarrow MBrACDrE11$

$1 \rightarrow rr$

[2] We present the algorithm given in [15]; in [8] Larus introduces a modification to the SEQUITUR algorithm which we have not used.

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

     Grammar G
     Rule T
1.   for each token in S
2.     append token to end of production for T
3.     if duplicate digram appears
4.       if other occurrence is a rule g in G
5.         replace new digram with non-terminal of g.
6.       else
7.         form a new rule and replace duplicate
8.         digrams with the new non-terminal.
9.     if any rule in G is used only once
10.      remove the rule by substituting the production.
11.  return G
```

Figure 4. SEQUITUR **algorithm.**

After applying the new rule the algorithm continues appending tokens and seeking redundancies. In this case another redundancy, MB, is found, and a second rule is created:

$\mathcal{T} \rightarrow 2rACDrE11x2$

$1 \rightarrow rr$

$2 \rightarrow MB$

The grammar is complete when there are no more tokens to process; in this case the result is:

$\mathcal{T} \rightarrow 2rACDrE113G43CF4rx$

$1 \rightarrow rr$

$2 \rightarrow MB$

$3 \rightarrow x2$

$4 \rightarrow 1r$

Following processing, the resulting grammar for these traces contains terminals (function names, r, and x) and rules, that can contain both terminals and other rules.

The SEQUITUR algorithm runs in time $O(N)$, where N is the trace length. The size of the compressed trace is $O(N)$ 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 SEQUITUR takes advantage of.

The SEQUITUR grammar can be stored as it is constructed in the form of a *whole path DAG* [8]. Each node in the whole path DAG is a rule in the SEQUITUR grammar. A directed acyclic graph is constructed by connecting the members of each production rule to their rule or non-terminal node. The whole path DAG for the above SEQUITUR grammar is shown in Figure 5. Note that the outgoing edges from any node are ordered by the members of the production rule. The size of the whole path DAG is equal to the size of the SEQUITUR grammar plus the sum of the lengths of the production rules, since each DAG node represents a grammar rule and the sum of the lengths of the production rules equals the number of DAG edges.

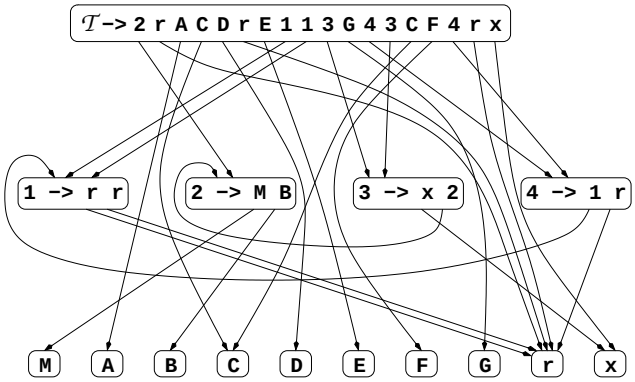

Figure 5. Whole path DAG.

4.2. Path impact algorithm

The `PathImpact` algorithm for estimating impact is shown in Figure 6. One way to visualize its operation is to consider beginning at a procedure node in the DAG, and ascending through the DAG performing recursive forward and backward in-order traversals at each node, stopping when any trace termination symbol is found. Traversing the DAG in this manner yields an impact set equivalent to that which would result by traversing the uncompressed traces.

Any changed procedure is a terminal, so `PathImpact` begins at that terminal, ascending upwards in the DAG searching forward for procedures that were called after the changed procedure, and searching backward for procedures that are returned into. This is implemented in the function `up` shown in Figure 7. The integer *act.stat* is used to signal the end of an execution trace and set the Boolean values of *fwd* and *bwd* accordingly. `Up` uses two functions, `forward` and `backward`, to search for terminals in the grammar that should be added to the impact set I; these functions are shown in Figures 8 and 9, respectively.

The integers *act.retn* and *act.skip* perform the necessary bookkeeping functions while searching forward, governing which functions are returned into while searching backward. `Forward` matches function names and returns, and passes the number of unmatched returns to `backward` in the structure named *act* and variable named *retn*.

`Backward` uses *act.retn* to determine how many procedures to include in the impact set. When `Backward` has added a number of procedures equal to *act.retn*, the search backwards in this trace can stop. `Backward` must also match function returns and names while searching backwards, since functions that return before the changed function is executed cannot be returned into. The variable *skip* controls this by incrementing when a return is encountered and decrementing when a procedure name is found. Procedures are added to I when *skip* is zero.

Since the `SEQUITUR` grammar forms a DAG, and a DAG has no cycles, the traversal performed by `PathImpact` must terminate. It is relatively easy to verify that `PathImpact` is correct for an uncompressed trace such as the one given in Figure 3. Since the `SEQUITUR` grammar exactly reproduces the uncompressed trace, the introduction of intermediate nodes in the DAG due to the formation of grammar rules does not affect the ability to retrieve the trace via the graph traversal. Since each trace records a sequence of actual calls and returns the calling context is always preserved. As long as the trace termination symbols are always appended to the end of each execution, and as long as they are honored while searching, `PathImpact` computes the predictive impact set outlined in Section 3.

```
algorithm PathImpact( G, c )
 input     Whole Path DAG G
 input     Changed Procedure c
 output    Impact Set, I

     struct Action contains:
          integer skip = initially 0
          integer retn = initially 0
          integer stat = initially 0
     Set I = set of potentially impacted procedures, initially empty.
     boolean fwd = initially true.
     boolean bwd = initially true.
     Action act.
 1.  if node c exists in G
 2.    add c to I
 3.    for each node p whose production contains c
 4.      up( p, c, fwd, bwd, act )
```

Figure 6. `PathImpact` algorithm.

```
algorithm up(p, c, fwd, bwd, act )
 input     Node p
 input     Node c
 input     boolean fwd
 input     boolean bwd
 input     Action act
 output    Action

     Node t
 1.  if( fwd == true )
 2.    t = c
 3.    while t has a successor in prod rule of p
 4.      t = successor of t
 5.      act = forward( t, act )
 6.      if( act.stat == -1 )
 7.        fwd = false
 8.        break
 9.  if( bwd == true )
10.    t = c
11.    while t has a predecessor in prod rule of p
12.      t = predecessor of t
13.      act = backward( t, act )
14.      if( act.stat == -1 )
15.        bwd = false
16.        break
17.  if( fwd == false AND bwd == false )
18.    act.stat = -1
19.    return act
20.  for each node q whose production contains p
21.    up( q, p, fwd, bwd, act )
```

Figure 7. `Up` function in `PathImpact`.

```
algorithm forward( g, act )
input     Node g
input     Action act
output    Action

     Node t
 1.  if( g is a procedure )
 2.    if( g is a program exit )
 3.      act.stat = -1
 4.      return act
 5.    if( g is a function return )
 6.      act.retn++
 7.      return act
 8.    add g to I
 9.    act.retn–
10.    return act
11.  t = first element of production rule of g
12.  act = forward( t, act )
13.  if( act.stat == DONE )
14.    return act
15.  while t has a successor in prod rule of g
16.    t = successor of t
17.    act = forward( t, act )
18.    if( act.stat == DONE )
19.      return act
20.  return act
```

Figure 8. `Forward` **function in** `PathImpact`.

Applying `PathImpact(E)` to the whole path DAG in Figure 5 for a proposed change in procedure *E*, the algorithm first adds *E* to the impact set, I. Next, since the only rule whose production contains *E* is the rule $\mathcal{T}$, `PathImpact` calls `up(`$\mathcal{T}$`,E,true,true,act)`. Procedure `up` searches forward to find procedures directly and transitively called by *E*. The immediate successor of *E* in rule $\mathcal{T}$ is *1*, therefore `up` calls `forward(1,act)`. `Forward` searches recursively for the end-of-trace symbol, *x*, and counts returns. The production rule for *1* results in two calls of `forward(r,act)`. The value of *act.retn* after these two calls is 2. Then control returns to `up` in rule $\mathcal{T}$, where the next successor of *E* is considered. Since the next successor is another *1*, another call of `forward(1,act)` and two more calls of `forward(r,act)` result, and the value of *act.retn* is 4. The next successor in the production rule for $\mathcal{T}$ is *3*. `Forward(3,act)` causes a call to `forward(x,act)`. Since *x* is the end-of-trace symbol, `forward` sets *act.stat* to -1 and returns. The value of *act.stat* causes `up` to break out of the forward searching loop, set the value of *fwd* to false, and begin searching backwards for procedures that *E* returns into.

`Backward` uses the value of *act.retn* and its own count of return symbols encountered, *act.skip*, to determine which procedures in the trace are actually returned into. The search starts by calling `backward(r,act)` since *r* is the immediate predecessor of *E* in the production for rule $\mathcal{T}$. Since *r* is a procedure return `Backward` increments the value of *act.skip*. The subsequent call, `backward(D,act)`, decrements *act.skip* rather than adding *D* to the impact set I. The next predecessor in the production for rule $\mathcal{T}$ is the procedure *C*. Execution of the call `backward(C,act)` adds procedure *C* to the impact set and decrements *act.retn*. Subsequent calls to `backward(2,act)` result in the addition of *A* and *M* to the impact set I and the decrementing of *act.retn* to 1. An execution that terminated abnormally would count fewer returns in *act.retn* and, consequently, cause `backward` to terminate sooner. In this example, since there is no next predecessor, the algorithm ends. The resulting impact set for `PathImpact(E)` is { M, A, C, E }.

```
algorithm backward( g, act )
input     Node g
input     Action act
output    Action

     Node t
 1.  if( g is a procedure )
 2.    if( g is a program exit )
 3.      act.stat = -1
 4.      return act
 5.    if( g is a procedure return )
 6.      act.skip++
 7.      return act
18.    if( act.skip > 0 )
19.      act.skip–
10.      return act
11.    if( act.retn > 0 )
12.      add g to I
13.      act.retn–
14.      return act
15.  t = last element of production rule of g
16.  act = backward( t, act )
17.  if( act.stat == DONE )
18.    return act
19.  while t has a predecessor in prod rule of g
20.    t = predecessor of t
21.    act = backward( t, act )
22.    if( act.stat == DONE )
23.      return act
24.  return act
```

Figure 9. `Backward` **function in** `PathImpact`.

Similarly, `PathImpact(G)` results in the impact set { M, B, G } and `PathImpact(F)` results in the set { M, B, C, F }. The worst-case running time of `PathImpact` is $O(N)$, where N is the uncompressed length of the concatenated traces. `PathImpact` does not require additional space above that required by the whole path DAG.

We are also interested in the ability to incrementally update a whole path DAG as a program and its operational profile evolve. Adding a trace to the whole path DAG can be easily accomplished due to the on-line nature of the `SEQUITUR` algorithm. Removing a trace simply involves deleting a section of the compressed trace and associated edges and adjusting the grammar. Adjusting traces in response to a deleted procedure requires removing traces containing that procedure from the whole path DAG, via simple modification of `PathImpact`. Beginning at the procedure that has been removed from the system, we search upwards, forwards, and backwards locating the traces where the pro-

cedure occurs and then remove each trace from the DAG. Addition of a procedure requires identification of traces containing procedures changed to call that procedure; these can then be deleted per the foregoing approach and collected again, and new versions inserted as above.

Finally, we return to a point raised in Section 3, where we mentioned the possibility of altering the instrumentation to explicitly identify returning procedures rather than returning a common symbol. Doing so can reduce the cost of searching the DAG, since we then need to search only in a forward direction. However, doing so would also approximately double the size of the traces considered, since each return would require a procedure name plus a special character to indicate a return. Also, pairs of returns would no longer match other pairs of returns unless they were associated with the same two procedures, in the same order, reducing possibilities for compression.

Therefore, a tradeoff exists between the cost of searching and the amount of compression obtainable. The `SEQUITUR` grammars generated in our experiments (see Section 5) were reduced to approximately one fifth of the average trace size of 1 MB. Larus reports reducing a 2GB trace to approximately 100 MB [8]. We would intuitively expect `SEQUITUR` to achieve better compression on larger traces. Due to the large number of `SEQUITUR` grammars generated in our experiments we have chosen to emphasize possible compression over a reduction in searching.

5. Empirical Validation

The research questions we wish to investigate are whether `PathImpact` computes an appropriate impact set relative to a specific operational profile or set of dynamic executions, and how that impact set compares to those computed by dependency-based impact analysis algorithms. To investigate these questions we performed an experiment comparing the impact sets calculated by `PathImpact` to those calculated by approaches based on transitive closure on the call graph and function-level static slicing.

5.1. Experiment materials

As an subject for our experiment 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. The developers of `Space` found 33 faults during the program's development. We found five additional faults [18]. We created 38 single fault versions, each containing one of the known faults. We excluded two versions that could not be analyzed by our tools.[3] The remaining 36 versions served as our experiment subjects, allowing us to model a process of performing predictive impact analysis on a function in which a fault is about to be corrected.

To provide input sets for use in determining dynamic behavior, we used test cases created for `space` for earlier experiments [18]. These test cases consisted of 10,000 randomly generated test cases and 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 7773 test cases including 2886 duplicates. To examine impact on single executions, we eliminated the duplicate test cases to obtain another set of 4887 individual test cases. We collected execution trace information for the individual test cases and for the test suites.

5.2. Experiment methodology

5.2.1 Measures

As discussed in Sections 1 and 3, the transitive closure and static dependency-based slicing techniques for impact analysis may over- or underestimate change impact, relative to a specific set of program behaviors as captured in a profile, a specific execution, or a test suite. Overestimates force maintainers to spend additional, unneeded time investigating potential impacts; underestimates may cause maintainers to omit, in their investigations, important potential impacts.

Thus, in this study, we examine these over- and underestimates. To do this, we compare the relative size and contents of the impact sets computed by transitive closure, `PathImpact` for single program executions, `PathImpact` for test suites, and function-level static slicing, for each of the faulty versions of `space`.

5.2.2 Impact analysis techniques

To calculate static function-level slices for each version we used the `Codesurfer` tool by *GrammaTech*.[4] In `Codesurfer` a function-level static forward slice uses all the dependencies identified by data and control dependence analysis in the changed procedure or procedures to identify dependent functions. We implemented the algorithm for impact analysis based on transitive closure, and we implemented the `PathImpact` algorithm.

5.2.3 Design

In this experiment, the three impact analysis techniques are our independent variables. Our dependent variables are the sets of functions identified as impacted by each technique.

We applied each impact analysis techniques to each version of `space`, calculating the number of functions that would be returned as potentially impacted by correction of

[3] On one version, the fault caused it to crash on nearly all executions, so no usable traces were produced. On a second version, the Codesurfer tool was unable to calculate accurate dependencies for a large array.

[4] Available from: GrammaTech, Inc. 317 North Aurora Street Ithaca, NY 14850, or http://www.grammatech.com.

the faulty function. For the transitive closure and function-level slice techniques this calculation was performed only once per version, since they are static and not dependent on test executions. For the `PathImpact` algorithm, we applied the algorithm once for each of the 4887 tests for each version. From the impact set resulting from each test, we calculated the differences between impact sets. The results for suites is based on the DAG created by concatenating all the executions from tests in a suite, including duplicates.

5.2.4 Threats to validity

We have studied only one program and set of changes, and we cannot claim that these are representative of programs and changes generally. Also, our tests do not represent operational profiles, and the test suites we use represent only one type of suite that could be found in practice. Additional studies of other programs, change distributions, and types of inputs are needed to address such threats to external validity. However, `space` is a "real world" program and its faults are actual faults, its test suites constitute one type of suite that could be used in practice.

A second concern involves our measures of impact. We would like to know how the impact sets returned by our algorithm correlate with true dynamic impact over the test suites. Such a measure would be difficult to obtain, however, given the general undecidability of precisely determining impact, and would most likely have to be accomplished at least partially through expensive and error-prone human calculations. However, our comparisons of the sets returned by various impact analysis techniques provide important perspective on their relative power.

Finally, we have considered only one static slicing implementation; other implementations may obtain different results. Also we have not yet compared our technique to dynamic slicing techniques, since no implementations were readily available and an appropriate implementation will require considerable effort to create. However, we intend to accomplish such a comparison in the future.

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

Figure 10 provides a view of the data we collected. Column one lists the version of `space`. Column 2 (**TC**) shows the size of the impact set resulting from transitive closure. The columns under the heading "**Individual Traces**" list measures of the impact set calculated by the `PathImpact` algorithm relative to the size and contents of the transitive closure set and static slicing set using individual program executions. The columns under the heading "**Test Suites**" show the same for executions of entire test suites. The column heading **PI/TC** is the average ratio of the size of the `PathImpact` set to the size of the transitive closure set. **PI-TC** is the average number of functions that are members of the `PathImpact` set, but not members of the transitive closure impact set. Likewise, **TC-PI** is the average number of functions that are members of the transitive closure impact set, but not members of the `PathImpact` set. **PI/FS** is the average ratio of the size of the `PathImpact` set to the size of the static slicing impact set. **PI-FS** is the average number of functions that are members of the `PathImpact` set, but are not members of the static slicing impact set. Similarly, **FS-PI** is the average number of functions that are members of the static slicing impact set, but not members of the `PathImpact` set. All of the average measures are obtained across either the sets of individual test executions, or the sets of test suites. We have calculated ratios so that the results can be compared across versions and test suites.

5.3.2 Analysis of impact sets

As Figure 10 shows, the transitive closure sets included far fewer procedures than those of the other two impact approaches. On examination of the changes it can be seen that this is due to the locations of the changed procedures in the call graph, most of which were relatively "deep" in the call graph (with the exception of version 30). The impact set calculated using `PathImpact` generally included (on average) many more procedures, and the set calculated using function-level static slicing contained the most.

Figure 11 contains a scatter-plot comparing sizes of the `PathImpact` sets to the sizes of the corresponding function-level static slice sets, for both test suite and individual test executions. Most of the data points lie on or below the $x = y$ line, with the exception of some of the cases involving versions 19, 20, and 21, which we discuss below. This graphically shows that `PathImpact` using single program executions calculates generally smaller sets than function-level static slicing with an increasing difference in the set sizes as the set size gets larger. `PathImpact` sets using test suite execution displayed a somewhat wider range of differences, with maximums approximately equal to the size of the static slicing impact sets.

We calculated paired *t*-test statistics between each of the four impact sets. In each of the comparisons the *t*-statistic was outside of the 95% confidence interval, the p-value was zero, and the null hypothesis (that the sets had the same mean) was rejected.

To directly compare the size of the **PI** sets with the size of the **TC** and **FS** sets across program versions we expressed the size of **PI** as a percent of **TC** or **FS** for each of the single execution impact sets and for the 50 test suites. Columns 3, 7, 10, and 14 in Figure 10 show the average relative size of **PI** for each version.

The **PI** sets were considerably larger than the **TC** sets, four and a half times larger in the case of single traces and nine and a half times larger for the test suites, and displayed

		Individual Traces							Test Suites							
Vers	**TC**	**PI/TC**	**PI-TC**	**TC-PI**	**PI**	**PI/FS**	**PI-FS**	**FS-PI**	**PI/TC**	**PI-TC**	**TC-PI**	**PI**	**PI/FS**	**PI-FS**	**FS-PI**	**FS**
1	3	14.8	41.6	0.0	44.6	0.45	1.0	55.4	33.0	96.0	0.0	99.0	1.00	1.0	1.0	99
2	3	16.6	46.8	0.0	49.8	0.44	1.0	64.2	36.5	106.4	0.0	109.4	0.97	1.0	4.6	113
3	2	22.4	42.8	0.0	44.8	0.37	1.0	76.2	56.5	111.0	0.0	113.0	0.94	1.0	8.0	120
4	3	6.0	14.9	0.0	13.0	0.18	1.0	82.1	6.3	16.0	6.3	19.0	0.19	1.0	81.0	99
5	3	4.6	10.9	0.0	13.9	0.18	1.0	100.1	7.0	18.0	0.0	21.0	0.18	1.0	93.0	113
6	2	8.0	14.0	0.0	16.0	0.13	1.0	105.0	15.5	28.6	0.0	30.6	0.26	1.0	90.4	120
7	4	1.8	5.4	2.2	7.3	0.40	0.5	11.2	4.1	13.3	1.0	16.3	0.90	1.0	2.7	18
8	3	1.7	3.2	1.2	5.0	0.72	0.5	2.5	2.7	5.0	0.0	8.0	1.14	1.0	0.0	7
9	6	2.9	14.0	2.6	17.4	0.51	0.7	17.3	5.4	27.7	1.0	32.7	0.96	1.0	2.3	34
10	4	4.1	14.4	2.1	16.4	0.48	0.6	18.2	7.0	25.0	1.0	28.0	0.82	1.0	7.0	34
11	5	3.2	13.0	2.0	16.0	0.47	0.4	18.4	6.4	28.1	1.0	32.1	0.94	1.0	2.9	34
12	4	3.4	12.6	3.0	13.6	0.52	0.5	12.9	3.6	13.6	3.0	14.6	0.56	0.7	12.1	26
13	14	1.6	13.4	5.0	22.4	0.55	0.7	19.3	2.9	29.0	2.0	41.0	1.00	2.0	2.0	41
14	6	4.5	21.3	0.2	27.1	0.40	1.0	41.9	10.9	59.7	0.0	65.7	0.96	1.0	3.3	68
15	6	3.2	14.0	0.6	19.4	0.31	1.0	44.6	7.7	40.4	0.0	46.4	0.74	1.0	17.6	63
16	7	0.9	3.7	4.4	6.3	0.57	0.5	5.3	1.6	5.0	1.0	11.0	1.00	1.0	1.0	11
17	6	3.7	17.2	1.0	22.2	0.52	0.6	21.4	6.9	36.2	1.0	41.2	0.96	1.0	2.8	43
18	4	3.8	12.6	1.0	15.6	0.56	0.5	12.9	4.2	13.6	1.0	16.6	0.59	0.7	12.1	28
19	1	4.0	3.0	0.0	4.0	1.33	1.0	0.0	1.3	3.0	0.0	4.0	1.33	1.0	0.0	3
20	1	7.0	6.0	0.0	7.0	1.17	1.0	0.0	6.9	5.9	0.0	6.9	1.14	1.0	0.1	6
21	1	7.0	6.0	0.0	7.0	1.17	1.0	0.0	6.9	5.9	0.0	6.9	1.14	1.0	0.1	6
22	5	1.7	4.8	1.4	8.4	0.93	1.8	2.4	1.9	5.5	1.0	9.5	1.05	2.0	1.5	9
23	1	10.7	9.7	0.0	10.8	0.60	1.0	8.2	16.4	15.4	0.0	16.4	0.91	1.0	2.6	18
24	14	1.6	13.2	5.0	22.3	0.52	0.6	21.3	3.0	29.0	1.0	42.0	0.98	1.0	2.0	43
25	6	1.1	5.0	4.3	6.8	0.21	0.2	25.5	4.8	25.8	3.0	28.8	0.90	1.0	4.2	32
26	18	2.0	26.3	8.4	35.9	0.40	0.9	55.0	5.0	72.9	1.0	89.9	1.00	1.0	1.1	90
27	7	0.9	3.7	4.4	6.3	0.57	0.6	5.3	1.6	5.0	1.0	11.0	1.00	1.0	1.0	11
28	17	1.9	20.5	4.7	32.8	0.42	1.0	47.2	3.9	50.0	0.0	67.0	0.85	1.0	13.0	79
30	16	0.4	0.0	10.2	5.8	0.07	0.0	73.2	0.8	0.0	3.5	12.5	0.16	0.0	66.5	79
31	12	2.1	17.0	4.1	24.9	0.23	1.0	85.1	7.7	81.7	1.0	92.7	0.85	1.0	17.3	109
32	3	2.6	5.9	1.2	7.7	0.48	0.8	9.1	5.3	13.8	1.0	15.8	0.99	1.0	1.2	16
33	3	3.0	7.0	1.0	9.0	1.00	1.0	1.0	3.0	7.0	1.0	9.0	1.00	1.0	1.0	9
35	5	1.3	4.5	3.2	6.4	0.35	0.5	12.1	3.3	12.5	1.0	16.5	0.91	1.0	2.5	18
36	1	7.0	6.0	0.0	7.0	0.12	1.0	50.0	49.0	48.0	0.0	49.0	0.88	3.0	10.0	56
37	13	1.3	10.8	6.4	17.4	0.51	0.6	17.2	2.6	22.0	1.0	34.0	1.00	1.0	1.0	34
38	13	1.4	11.2	6.3	17.9	0.52	0.6	16.8	2.6	22.0	1.0	34.0	1.00	1.0	1.0	34
Avg		4.56				0.51			9.51				0.87			
Std Dev		4.78				0.29			13.12				0.27			

Figure 10. Impact sets and their relationships calculated in the experiment.

a large standard deviation. Averaged across all the versions the **PI** impact sets were about one half (51%) of the size of the **FS** impact sets using single executions and 87% of their size using test suites. In the case of versions 19, 20, and 21, **PI** was larger than **FS** by one function. On inspection we found that in every case the additional function was the same. This extra function executes near the end of nearly every test and has no static dependencies (its purpose is simply to pause program execution after displaying output to the screen). Since the function nearly always executes it is nearly always included in **PI**, but since it has no static dependencies it is never included in **FS**. In practice, maintainers familiar with the system could cause it to be excluded from future impact consideration.

5.4. Discussion

Our goal was to compare the impact sets calculated by our four target approaches when performing predictive impact analysis relative to a specific set of program behaviors captured by a specific input set. We now comment on some of the implications of our results.

The sets shown in Figure 10 show that `PathImpact` caught a large number of procedures which would be missed by transitive closure, but did not include as many

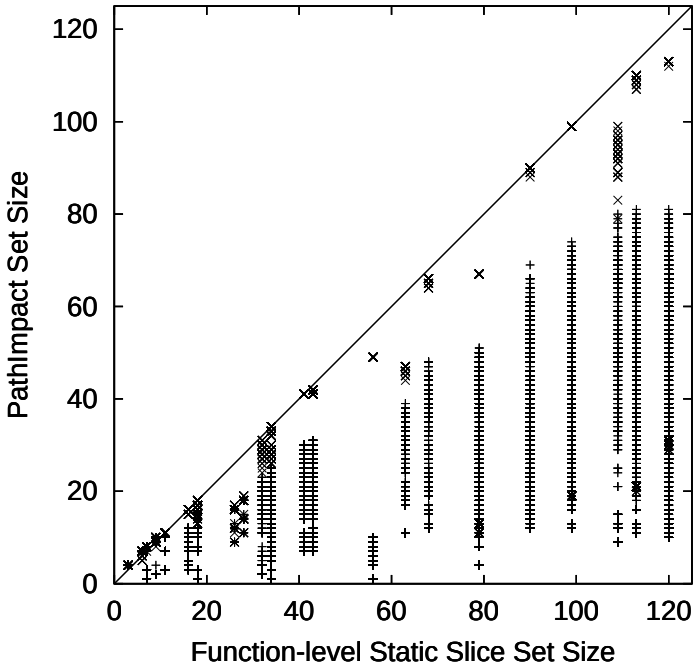

Figure 11. Impact set size comparison. Single traces are shown as "+", suites are shown as "×".

procedures as function-level static slicing. This is good on both counts since it is moderated by the fact that the included procedures *actually* executed, and thus could have propagated actual impact.

The impact sets observed for transitive closure were unreliable, having more to to with the location of the change in the call graph than the behavior of the program. This reinforces the notion that impact propagation is more dependent on the behavior of the program than the program's structure. `PathImpact` moderates the results of transitive closure and static slicing in cases where we have an interest in specific program behavior embodied in an operational profile or a specific set of tests.

Function-level static slicing is also, in some sense, a reflection of the program's structure. Static slices represent every possible behavior rather than actual behavior. In our experiments, this sometimes led to overestimation of change impact using static slicing. However, the test suites used, while showing considerable variation, often came remarkably close to agreement with the static slicing impact sets. Further, cases in which profile-based impact analysis overestimates impact are fewer, and the amount of overestimation is less, than with static slicing.

Of course, the extent to which the estimate from `PathImpact` matches actual impact is entirely dependent on the appropriateness of the operational profile or set of tests used. But it is these situations, where that appropriateness holds, that are our focus.

We have not presented execution time or slow-down statistics since our initial implementation processes stored traces with Java programs, rather than using a fully online implementation. A fully online implementation would build the DAG as the instrumentation executes and not save the uncompressed traces, and would be considerably more efficient than our prototype. Nonetheless, the simulation times we observed were relatively small. The time required to generate the DAG was less than one minute per test suite, and the time required to calculate an impact set was at most several seconds.

6 Conclusions and Future Work

The call graph of a program has severe limitations for predicting the impact of program changes. Static slicing is much more precise than transitive closure on call graphs, but may be expensive. It also may return unnecessarily large impact sets if the goal is to predict impact relative to a set of dynamic behaviors as present in a user profile or test suite.

We have thus introduced a new approach to function-level impact analysis, based on dynamic information obtained through simple program instrumentation. Our approach does not rely on availability of program source code and does not require static dependency calculations. In our approach dynamic traces are compressed using the `SEQUITUR` data compression algorithm [15], and a directed acyclic graph (DAG) is constructed following Larus [8]. We then use the `PathImpact` algorithm presented in this pa-

per to predict dynamic change impact. The results of our experiment show that `PathImpact` can provide potentially more useful predictions of change impact than function-level static slicing in situations where specific program behavior is the focus.

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. Future implementations will address performance issues such as program slowdown and impact calculation times. We also intend to investigate statistical measures of impact such as density functions for the predicted impact sets. 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 those based on static slicing.

Eventually, a thorough understanding of our technique will require an examination of the return on investment (ROI) in practice. By this examination, we hope to determine the extent to which the technique can save software maintainer's time and improve the quality of systems undergoing maintenance.

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] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33(3):60–66, Mar. 2000.

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

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

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

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

[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] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

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

[22] 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.

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