Probabilistic Slicing for Predictive Impact Analysis

Raul Santelices and Mary Jean Harrold
College of Computing, Georgia Institute of Technology
E-mail: {raul|harrold}@cc.gatech.edu

ABSTRACT

Program slicing is a technique that determines which statements in a program affect or are affected by another statement in that program. Static forward slicing, in particular, can be used for impact analysis by identifying all potential effects of changes in software. This information helps developers design and test their changes. Unfortunately, static slicing is too imprecise—it often produces large sets of potentially affected statements, limiting its usefulness. To reduce the resulting set of statements, other forms of slicing have been proposed, such as dynamic slicing and thin slicing, but they can miss relevant statements. In this paper, we present a new technique, called *Probabilistic Slicing* (*p-slicing*), that augments a static forward slice with a relevance score for each statement by exploiting the observation that not all statements have the same probability of being affected by a change. P-slicing can be used, for example, to focus the attention of developers on the "most impacted" parts of the program first. It can also help testers, for example, by estimating the difficulty of "killing" a particular mutant in mutation testing and prioritizing test cases. We also present an empirical study that shows the effectiveness of p-slicing for predictive impact analysis and we discuss potential benefits for other tasks.

1. INTRODUCTION

Software is constantly modified during its life cycle, resulting in many challenges for developers because changes might not behave as expected or may introduce erroneous side effects. Whenever software must be modified to achieve some goal (e.g., fix errors, add new functionality, or improve the quality of the code), developers must assess the parts of the software potentially impacted by planned changes. This change-planning task is particularly delicate in large and complex software where developers do not fully understand the consequences that a change might have. Also, after software is modified, developers must identify the parts affected by changes so that they can retest those parts. To help accomplish these tasks, developers use *change impact analysis* (or, simply, *impact analysis*), which determines, for some level of granularity (e.g., statements, modules, features), which entities in the software can be affected by changes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright Georgia Institute of Technology. November, 2010..

Program slicing is a well-known technique that can be used for various software-quality tasks, including impact analysis. Program slicing was originally developed as a backward analysis of program code to aid in its comprehension and debugging [22]. Its counterpart, forward slicing, is used for change-related tasks, such as impact analysis [4], regression testing [15], and program integration [3, 18]. These techniques, collectively called *static* slicing, use the control and data dependencies in the code to identify the set of all statements that can affect or be affected by another statement. Unfortunately, static slicing is often too imprecise for practical use because it tends to produce very large sets of statements. To reduce the resulting set of statements, other forms of slicing have been developed, such as dynamic slicing [10] and thin slicing [20], but they are incomplete and can miss important parts of the code.

To address the limitations of previous slicing techniques for use in impact analysis, we developed, and present in this paper, a new technique, called *Probabilistic Slicing* (*p-slicing*), that augments a forward slice with a relevance score for each statement. P-slicing exploits the observation that not all statements have the same probability of being affected by a change. Intuitively, a forward slice only indicates "whether" a statement s is affected by a change C no matter how small that influence might be, whereas the relevance score indicates "how much" s is affected by C. The relevance score is computed by statically analyzing the probability that (1) an execution reaches s after executing C, (2) a sequence of data and control dependencies is exercised between C and s, and (3) a modification of the program state (an *infection* [21]) propagates from C to s through that sequence of dependencies. If these three events occur, then either the execution history of s (i.e., the number of occurrences of s during an execution) or the values computed or branching decisions taken at s are modified due to C, and, thus, s is impacted by C.

Our new technique exploits two important insights not fully considered in existing program-slicing research:

1. Our technique recognizes that some data dependencies are less likely to be covered than other data dependencies because the conditions to reach a use from a definition can be more complex and difficult to satisfy in some cases. To incorporate this factor, p-slicing uses an interprocedural reachability analysis to estimate the probability that a use is reached by a definition.
2. Our technique not only recognizes that data dependencies are more likely to propagate infections than control dependencies [8, 12, 20], but, unlike techniques that discard some or all control dependencies [20], our technique includes control dependencies from the beginning and gives them a lesser propagation probability than for data dependencies.

To perform forward slicing and compute the probabilities (relevance scores) for each statement, p-slicing annotates the interproce-

dural dependence graph [9] used for slicing so that each data or control dependence is labeled with the probability that the dependence will both be covered and propagate an infection after the source point of the dependence—the definition or branching statement—is reached. In this weighted graph, every path between a change location C and a statement s has a probability of propagating an infection from C to s that results from composing the probabilities of the individual edges (dependencies). To compute the probability that C affects s through any path between them, p-slicing performs a simplified data-flow analysis of the graph by leveraging the probabilities of individual dependencies to compute the transitive propagation probabilities from C to any other statement in the graph—the statements in the forward slice.

To the best of our knowledge, only one existing technique, called dynamic impact analysis [8], assigns probabilities to statements in a slice. However, this technique works only on the backward dynamic slice for a particular execution and is designed to assess which executed components contributed to the success or failure of that execution. Our new technique, instead, works on static slices that represent all possible executions (not just observed executions) to predict the impact of changes. Also, dynamic impact analysis considers only the dependence path (i.e., dependence sequence) with the highest propagation probability and does not consider the coverage probability of each component. In contrast, our technique composes the propagation probabilities for all dependence paths between two components and does consider the coverage probability of each component. (Individual paths might have a low impact probability but, if many paths exist, the cumulative impact probability of a change can be much higher.)

P-slicing has a number of applications related to changes in software. For example, it can be used to initially focus the attention of developers on the "most strongly impacted" parts of the program—the parts with the greatest relevance scores. For another example, it can help testers estimate the most vulnerable areas of the program that need to be carefully re-tested. For a third example, it gives testers an estimate of the difficulty of exercising the impact of a change on a particular statement or the program's output. For mutation testing [5], in which the cost of "killing" all mutants is usually prohibitive, testers can first target those mutants with the highest probability of being killed and discard as "likely equivalent" those mutants with a propagation probability below some threshold.

The main benefit of our work is the usefulness of p-slices over existing techniques for change-related tasks. P-slices are more useful than simple forward slices for two reasons: (1) they indicate not only which statements are affected by changes, but also how much each statement is affected, and (2) unlike dynamic techniques, which provide information on what has been observed, the relevance scores reflect all possible behaviors of the software, and thus, can be used for predictive purposes (e.g., identifying potential impacts that have not yet been observed). A second benefit of our introduction of p-slicing is that existing backward slicing techniques for program debugging and comprehension can potentially benefit from our probabilistic infection-propagation analysis.

In this paper, we present an empirical study of the effectiveness of p-slicing for predicting the actual impact that changes have within a slice when the changed programs are executed. The results indicate that p-slicing can drive the developer's attention to impacted statements more effectively than simply using a slice without probabilities. The results show, in particular, that for the 25% or so of the statements with the highest p-slice scores, the chances of finding statements that are actually impacted are the greatest. Thus, our results suggest that p-slicing should be used to predictively identify a large number of impacted statements quickly.

The main contributions of this paper are

- The concept of *p-slice* that distinguishes statements in forward slices by their probability of being impacted by changes.
- A new technique, *p-slicing*, that computes this probability for each statement in the program.
- An empirical study that shows, for the subjects we used, the effectiveness of p-slicing for predicting the actual impact that changes have at runtime.

2. FUNDAMENTAL CONCEPTS

This section first describes an example used to illustrate various concepts in this paper and then presents the core concepts for the rest of the paper.

2.1 Example

Figure 1 presents the example that we use throughout this paper. The first two columns in the figure list the program, consisting of four classes `M`, `B`, `P`, and `Q`, where `P` and `Q` are subclasses of `B` that override the method `B.foo`. The entry point of the program is the method `M.main`. In lines 1 and 2, the program creates instances of `P` and `Q` that are assigned to references `p` and `q`, respectively. Indirectly, in each case, the constructor of `B` is invoked with arguments `a` and `b`. At line 3, the program checks a condition that determines whether it continues to line 4 or exits. At line 4, one of the two references is assigned to `r`, depending on the value of `c`. At lines 5–9, conditions on the value of the field `x` of the instances of `P` and `Q` determine which call to `foo` is performed—the call at 7, 8, or 9. Finally, a loop at lines 10–11 decrements the field `x` of `q` until it is no longer greater than `c`.

Method `B.foo` performs an update of `x` depending on the condition at line 13. This method is overridden by `P` and `Q` at lines 17 and 18, respectively, so `B.foo` is never called directly. Instead, `P.foo` invokes `B.foo`.

The right of Figure 1 shows an almost complete *Program Dependence Graph* (PDG) [7] for this program. (We use an interprocedural form of the PDG [9, 19].) The nodes in this PDG are the statements of the program and two special nodes: START for the entry of the program and EN for the entry of `M.main`. In this PDG, we omit the constructor of `B` and parts of `B.foo`.

2.2 Control and Data Dependencies

A solid edge in the PDG represents the control dependence of the target node of the edge on the source node of the edge. Node n_1 is *control dependent* [7] on node n_2 if the decision taken at n_2 determines whether n_1 is necessarily executed. Control dependencies are created by the presence of branches and by other control decisions including virtual calls for which there is more one target method [19]. In Figure 1, an example of the former type is node 8, which is control dependent on node 6 taking the false branch. An example of the latter type is the call to `foo` at line 8, which decides at runtime whether `P.foo` (node 17) or `Q.foo` (node 18) is called. In the PDG, the special node START and its outgoing edges represent the decision of executing the program. The bottom right part of this figure shows, separately, the control-dependence edges for `B.foo`, including the interprocedural control dependences of nodes 13 and 16 on nodes 6, 8, and 9.

A dashed edge in the PDG represents a data dependence of the target node on the source node, indicating that a variable defined at the source node is used at the target node and that there is a definition-clear path in the program (i.e., a path that does not redefine the variable) between the source and target nodes. For example, in Figure 1, node 6 is data dependent on node 4 because 4

```
    class M {
      void main(int a, b, c) {
 1:   B p = new P(a); // ch1: a -> -a
 2:   B q = new Q(b); // ch2: b -> b+1
 3:   if (a+b+c < p.x) {
 4:     B r = (c%2==0)? p : q;
 5:     if (p.x > 0) {
 6:       if (r.x > 0)
 7:         p.foo(q.x);
          else
 8:         r.foo(q.x);
        }
      else
 9:     print r.foo(0);

10:   while (q.x > c)
11:     q.x--;
    }}}
```

```
    class B {
      int x;
12:   B(_x) { x = _x; }
      int foo(y) {
13:     if (x > y)
14:       x -= 100*y;
        else
15:       x++;
16:     return -y;
      }
      ...
    }
    class P extends B {
17:   int foo(y) { P.foo(y+1); }
      ...
    }
    class Q extends B {
18:   int foo(y) { return y*2; }
      ...
    }
```

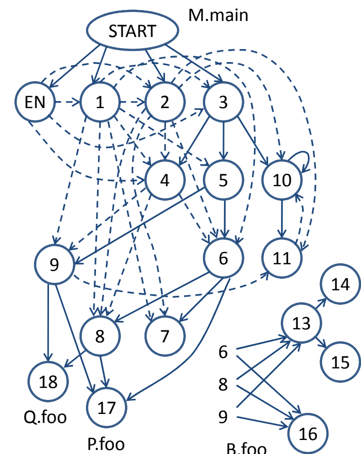

Figure 1: Example program consisting of classes M, B, P, and Q. M has two changes. On the right, a partial PDG for this program.

defines `r`, 6 uses `r`, and there is a path (4,5,6) that does not redefine `r` after 4. For space reasons, in this figure, we omitted the interprocedural data dependencies, and, instead, treated each definition and use from methods other than `M.main` as occurring inside the node in `M.main` where those methods are called. For example, we treat nodes 1 and 2 as definitions of the field `x`.

2.3 Slicing

Program slicing [22], also called static slicing, determines which statements of the program may affect or be affected (impacted) by another statement. A *static forward slice* (or, simply, forward slice) from node n is the set containing n and all nodes transitively affected by n along control and data dependencies, which corresponds to the set of all nodes reachable from n in the PDG. (To avoid unrealizable interprocedural paths, a solution by Horwitz, Reps, and Binkley [9] can be used.) For example, the forward slice from node 2 that indicates the potential impact of a change at that node (e.g., `ch2`) is the set {2,4,6,7,8,10,11,12,13,14,15,16,17,18}. This set contains all nodes in classes B, P, an Q because the constructor of B is implicitly invoked at 2 and because `r` might point to the same object as `q`. Also, the condition at node 6 controls the call to `P.foo` at 7 and, if `r` also points to the same object as `q` at 8, the call to `Q.foo` at 8 as well.

2.4 Change Impact

The impact of a change has been defined in the literature at various levels of granularity and precision [2,4,11,14]. Often, the presence of a statement in a static or dynamic forward slice is considered as indicative that the statement is affected or impacted. However, we are interested in the most precise definition of impact at the statement level, which implies a *semantic dependence* [13] of an impacted statement on a changed statement. In previous work, we used the concept of semantic dependence to define the impact of changes on other statements and changes [18]. In this paper, we use that definition.

Essentially, a statement s is impacted by a change C if applying C causes s to change its behavior for some execution. The behavior of a statement s in an execution is the *augmented* execution history [18] of s, which consists of the sequence of pairs (s^i, V_{s^i}) where s^i is the ith occurrence of s in that execution and V_{s^i} is the set of values computed and stored by s^i.

3. TECHNIQUE

P-slicing is a predictive technique that enriches the static forward slice from a statement C with a relevance score for each statement s in that slice. The score for statement s is an estimate of the probability that s is impacted by change C for any execution of the program. In this section, we first present an overview of p-slicing (Section 3.1) followed by the probabilistic model on which p-slicing is based (Section 3.2), the coverage and propagation probabilities of individual dependencies as required by the model (Sections 3.3 and 3.4), and the approximations made by p-slicing to compute these probabilities in practice (Section 3.5).

3.1 Overview

The probability that a change in a statement C impacts a statement s has two factors: (1) the coverage of a sequence of dependencies from C to s and (2) the propagation of an infection (i.e., state modification) from C to s along that sequence.

3.1.1 Coverage Probability

The motivation for considering the first factor in our technique is that not only it is necessary that a statement s be in the forward slice of a change C to be impacted by C, but s must actually be reached along some sequence of dependencies after C is executed. In general, different statements in a slice have different probabilities of being reached along some sequence of dependencies (i.e., path in the PDG). Thus, a statement that is more likely to be reached along a dependence sequence than another statement has a greater chance of being impacted by a change during an execution (if all other impact factors are the same for both statements).

For example, for change `ch1` in Figure 1, which modifies the field `x` of the object instantiated at line 1, the forward slice includes statements 3 and 5, both of which use the field `x` of that object. Both data dependencies look identical—they use the same field before it has a chance of being overwritten. However, p-slicing recognizes that statement 3 is more likely to be impacted by the change than statement 5 because statement 3 is always reached, whereas reaching statement 5 depends on a branching decision (at statement 3).

P-slicing performs a reachability analysis to compute the probability that each dependence in the slice is covered after the source of the dependence is reached. For a control dependence (s,s'), this analysis simply assigns to the dependence a probability $\frac{1}{N}$ of being covered, where N is the number of branches or decision outcomes that s can take. For example, the coverage probability for control dependence (3,10) in Figure 1 is 0.5 because N is 2 for statement 3.

For a data dependence (d,u), p-slicing performs a data-flow analysis that considers all definition-clear paths between d and u and propagates along each control-flow edge[1] the probabilities for all reachable uses [1] from that edge.[2] For example, the probability that the definition of `p.x` at 1 reaches its use at 3 is 1.0, whereas the probability that the same definition reaches its use at 5 is 0.5. Another example is the use of `p` at 7, which is reached by the definition of `p` at 1 after the conditions at 5 and 6 evaluate to true; its coverage probability is, therefore, $\frac{1}{2} \times \frac{1}{2}$, or 0.25. Note that the coverage probability for this dependence is less than for the other two examples simply because more conditions are required for the use to be reached.

P-slicing also accounts for aliasing as a factor in the probability that a data dependence is covered. For such a dependence to be covered, not only it is necessary that the use is reached along some path from the definition, but also, if the variable at the definition or the use can be aliased with more than one object, the dependence might not be covered. For example, the object pointed to by variable `r` in Figure 1 can be aliased with the objects pointed to by `p` and `q`. Thus, the coverage probability between the definition of `p.x` at line 1 and the use `r.x` at line 6 is not just 0.25, but it must be reduced because `r.x` might be aliased with `q.x` rather than `p.x`. Thus, p-slicing incorporates into the coverage probability of this data dependence the probability that `r.x` is aliased with `p.x` at line 6, which p-slice estimates as 0.5. Hence, the coverage probability for this dependence is 0.25×0.5, or 0.125.

3.1.2 Propagation Probability

The coverage of a dependence sequence from a change C to a statement s is a necessary but not sufficient condition for C to impact s. Thus, p-slicing not only computes the probability that s is reached from C along some dependence sequence, but also incorporates the probability that an infection at C propagates from each dependence to the next in a covered sequence.

It has been shown that control dependencies are, in general, less likely than data dependencies to propagate an infection from its source to its target [12, 20]. For example, if the condition at line 5 in Figure 1 changes to `p.x > -10`, then this change will have an impact on the rest of the program for only some values of the input `a`. However, the influence of a control dependence cannot be discarded altogether. Frequently, changes modify parts of the path taken by a program, such as the number of iterations in a loop. Thus p-slicing uses a measure for the infection-propagation probability of control dependencies that is less than for data dependencies but that still contributes to the overall propagation probability between two points in the dependence graph.

In general, it is not possible to compute with full accuracy the probability that a dependence will propagate an infection from its source to its target. For example, for the change of the condition at line 5 to `p.x > -10`, this probability depends on the input distribution of `a`. Therefore, p-slicing makes the simplifying assumption that, after a branching statement is infected, there is an equal chance that any of the N outcomes of that statement can be the result of evaluating the infected condition. Thus, the probability that an infected branching condition switches to another outcome (i.e., infects the target of each control dependence associated to the new outcome) is $\frac{N-1}{N}$. In our example, p-slicing assigns to each control dependence from statement 5 the infection-propagation probability 0.5 because N is 2 in this case.

For data dependencies, consider an infected operand in an expression used to define a variable. The infection of that operand will, in most cases, be transferred to the defined variable and then will reach its uses.[3] Therefore, p-slicing makes the simplifying assumption that the propagation probability for a covered data dependence is 1.0 (note that aliasing was handled in the coverage probability). For example, if `r` is infected at line 4 in Figure 1, then the propagation probability to its uses at 6, 8, and 9 is 1.0.

3.1.3 Impact Probability

Given the coverage and propagation probabilities for individual dependencies, p-slicing computes the product of these two probabilities to obtain the impact probability for each dependence—the probability that, after an infection reaches the source of the dependence, the target of the dependence is impacted (i.e., reached and infected). For example, a control dependence whose source has two branches has an impact probability of 0.5×0.5, or 0.25.

After the impact probabilities for dependencies have been computed, p-slicing performs another data-flow analysis, this time on the PDG, composing (multiplying) the impact probabilities of consecutive dependencies for all dependence paths between two statements s and s' in the slice to determine the probability that s' is impacted along some dependence sequence from s. Like the data-flow analysis from Section 3.1.1, this data-flow analysis obtains the impact probabilities from the change to not just one, but all statements in the forward slice.

For example, in Figure 1, consider change `ch2` and statement 6, which is affected by `q` (defined at 2) through a sequence of two data dependencies: (2,4) and (4,6).[4] The coverage probability for `q` being used at 4 is 0.25 because it requires the condition at 3 to be true and the inlined condition at 4 to be false. After that use is reached and `q` is assigned to `r`, the coverage probability for (4,6) is 0.5 because it only requires the condition at 5 to be true. Hence, the coverage probability that node 6 is reached from `ch2` via dependencies is 0.125. Note that this is only half the probability that an execution reaches 6 from 2 through any path (i.e., 0.25). Thus, unlike a simple forward slice that lists the affected statements, p-slicing identifies how a statement is reached within a slice and assigns probabilities accordingly. Meanwhile, the propagation probabilities for these two data dependencies is simply 1.0, so the impact probability of change `ch2` on statement 5 is 0.125.

Another example is the impact of `ch1` on statement 6, which involves both control and data dependencies and is affected through more than one dependence sequence. (As in the previous example, assume that `ch1` impacts both the reference `p` and the field `x`.) Statement 6 uses `r` and the field `x` of the object that `r` references, which is one of the instances created at lines 1 and 2. In this case, both `r` and `x` can be impacted by the dependence sequence q_1 = ((1,3),(3,5),(5,6)) consisting of one data dependence and two control dependencies. The impact probabilities for these dependencies are, respectively, 1.0, 0.25, and 0.25. Thus, the impact probability of `ch1` on statement 6 through this sequence is $1.0 \times 0.25 \times 0.25$, or 0.0625. There are, however, three other sequences connecting `ch1` and the variables used at statement 6: q_2 = ((1,3),(3,4),(4,6)), q_3 = ((1,4),(4,6)), and q_4 = ((1,6)), where (1,3), (1,4), (1,6), and (4,6) are data dependencies. The impact probabilities for those sequences are 0.125, 0.125, and 0.125, respectively. If `ch1` does not

[1] A control-flow edge represents the execution-successor relationship among statements in a program. Control-flow edges, along with the nodes representing the statements, form the control-flow graph (CFG) of the program.

[2] In fact, one data-flow analysis is sufficient to compute the coverage probabilities for all data dependencies in a slice or program.

[3] The other source of infection of a definition is a branching statement on which the definition depends. An infection in that statement propagates to the definition if it makes the definition execute when, otherwise, it would not execute (or vice versa).

[4] For the sake of the argument, assume for this example that `ch2` modifes both the field `x` and the object assigned to `q` at line 2.

impact statement 6 via q_1, `ch1` can still impact that statement via q_2, q_3, or q_4. Thus, p-slicing combines these probabilities using the data-flow algorithms induced by the system of equations presented in the next sections, which guarantees that the overall impact probability of `ch1` on statement 6 reflects the three sequences through which that impact can occur.

3.2 Probabilistic Model of a Slice

This section formally presents our probabilistic model used to compute the probabilities (relevance scores) for p-slicing.

Let C be the statement denoting the location where a change is planned or has already been performed. P-slicing computes the *p-slice* for C, which consists of the forward static slice from C and, associated with each statement s in the slice, a relevance score denoting the probability that C impacts s.[5]

The probability that a change C affects a statement s is given by

$$P(C \rightarrow s) = \begin{cases} 1 & \text{if } C = s; \\ P(\overset{d}{\rightarrow}) & \text{if } src(d){=}C \wedge tgt(d){=}s; \\ P\left(\bigvee_{d \in out(C)} \overset{d}{\rightarrow} \wedge tgt(d) \rightarrow s \right) & \text{otherwise.} \end{cases} \tag{1}$$

where $P(x)$ is the probability of event x, $s \rightarrow t$ is the coverage of s and t such that an infection propagates from s to t, $src(d)$ and $tgt(d)$ are, respectively, the source and target statements of dependence d, $\overset{d}{\rightarrow}$ is the coverage of dependence d (implicitly assuming that the source of d has been reached) such that an infection propagates from $src(d)$ to $tgt(d)$ through d, and $out(s)$ is the set of data and control dependencies d such that $src(d){=}s$.

Equation 1 has three cases for $P(C \rightarrow s)$. If s is the change location, then the probability is trivially 1. Else, if s is directly dependent on C (i.e., a dependence d exists from C to s), then $P(C \rightarrow s)$ is the probability that the dependence between them is both covered and an infection propagates through it. Otherwise, $P(C \rightarrow s)$ is the probability of the disjunction of $|out(d)|$ events, each corresponding to (1) the coverage of a dependence d whose source is C, (2) the propagation of an infection through d, and (3) that the target of d affects s. The last part of this event makes the computation of $P(C \rightarrow s)$ recursive.

In our model, the probability of $\overset{d}{\rightarrow}$ can be expressed as the joint probability of its two components

$$P(\overset{d}{\rightarrow}) = P(\overset{d}{\rightarrow}_{cov}) \times P(\overset{d}{\rightarrow}_{prop}) \tag{2}$$

where $\overset{d}{\rightarrow}_{cov}$ is the event that d is covered (assuming that $src(d)$ has been reached) and $\overset{d}{\rightarrow}_{prop}$ is the event that d propagates an infection from its source to its target (assuming that d has been covered).

3.3 Coverage Probability

In this section, we discuss the coverage probabilities of control and data dependencies, respectively, as computed by p-slicing.

3.3.1 Control Dependencies

Given a branching statement s that has N branches ($N > 1$), we denote the probability that s takes the ith branch by $P(s \text{ branches to } i)$. Therefore,

$$\sum_{i=1}^{N} P(s \text{ branches to } i) = 1$$

$P(s \text{ branches to } i)$ is the coverage probability for all control dependencies cd_i from s labeled with branching decision i. Without assuming anything about the condition evaluated at s, the probability for each branch i (and for any associated control dependence cd_i) can be reduced to

$$P(\overset{cd_i}{\rightarrow}_{cov}) = P(s \text{ branches to } i) = \frac{1}{N} \tag{3}$$

3.3.2 Data Dependencies

The coverage probability for data dependencies is a more complicated matter. Necessarily, a data dependence is covered if, after its definition has been reached (assumed to be already true), some definition-clear path to its use is then executed. However, for objects such as instance fields and array elements, aliasing must also be considered. Thus, an additional condition for the coverage of such data dependencies is that the memory location written at the definition must be aliased to the memory location written at the use and that no memory location written along the path between definition and use can be aliased to the defined memory location. Therefore, the coverage probability for a data dependence du (where d is the definition and u is the use) can be expressed as

$$P(\overset{du}{\rightarrow}_{cov}) = P\left(\bigvee_{p \in paths(du)} \begin{array}{c} p \text{ covered} \wedge alias(d,u) \wedge \\ \nexists s : s \in p \wedge s \neq d,u \wedge \\ alias(def(s),d) \end{array} \right) \tag{4}$$

where $paths(du)$ is the set of all definition-clear paths between definition d and use u, $alias(d,e)$ indicates whether the memory locations defined or used at d and e are the same, $s \in p$ represents a statement instance[6] in path p, and $def(s)$ is the definition (if any) occurring at statement instance s.

Equation 4 reflects all the factors involved in the coverage of a data dependence but it cannot be implemented directly because the number of paths between definition and use can be infinite. Instead, our technique uses a slightly simpler equation pair:

$$P(\overset{du}{\rightarrow}_{cov}) = P\left(\bigvee_{s \in succ(d)} \begin{array}{c} s \text{ succeeds } d \wedge \\ s \text{ reaches } u \end{array} \right) \times P(alias(d,u)) \tag{5}$$

$$P(s \text{ reaches } u) = \begin{cases} 1 & \text{if } s = u; \\ 0 & \text{if } u \notin ru(s) \\ P\left(\bigvee_{s' \in succ(s)} \begin{array}{c} s' \text{ succeeds } s \wedge \\ s' \text{ reaches } u \end{array} \right) & \text{if } u \in ru(s) \end{cases} \tag{6}$$

where $succ(s)$ is the set of all control-flow successor statements of the statement or definition s, "s succeeds d" is the event that the immediate successor of definition or statement d in an execution is s, "s reaches u" is the event that u is reached along some definition-clear path from s, $alias(d,u)$ is the event that the variable defined at d is the same one used at u, and $ru(s)$ is the set of reachable uses [1] from statement s.

[5] See Section 2.4 for the definition of impact.

[6] A statement instance is a particular occurrence of a statement in a path; a statement can occur many times in the same path.

Because the events in the disjunctions in equations 5 and 6 are exclusive—exactly one of the control-flow successors of a statement succeeds it—and because the "succeeds" and "reaches" events are independent, the probability of each of these disjunctions is

$$P\left(\bigvee_{\substack{s_i \in \\ succ(s)}} \begin{array}{c} s_i \text{ succeeds } s \wedge \\ s_i \text{ reaches } u \end{array}\right) = \sum_{i=1}^{|succ(s)|} \begin{array}{l} \prod_{j=1}^{i-1} (1 - P(s_j \text{ succeeds } s)) \\ \times P(s_i \text{ succeeds } s) \\ \times P(s_i \text{ reaches } u) \end{array} \tag{7}$$

To compute an estimate of the probability that two variables are aliased (i.e., the references or pointers used to access them refer to the same memory location), our technique considers the points-to sets—found by a pointer analysis [1]—of the two references and uses the probability that an element of one set is in both sets:

$$P(alias(a,b)) = \frac{|\ PointsToSet(a) \bigcap PointsToSet(b)\ |}{|\ PointsToSet(a) \bigcup PointsToSet(b)\ |} \tag{8}$$

where a and b are references and $PointsToSet(r)$ is the points-to set of reference (pointer) r. The $alias(.,.)$ terms in equations 4 and 5 translate directly into $alias(a,b)$, where a and b are the references to the variables involved in those definitions and uses.

3.4 Propagation Probability

Given a dependence d that is covered, the probability that d also propagates an infection if an infection arrives at the source of d is

$$P(\xrightarrow{d}_{prop}) = \begin{cases} P(rhs(src(d)) \text{ propinf } lhs(src(d))) & \text{if } d \text{ is a data dependence;} \\ P(cond(src(d)) \text{ switches branch}) & \text{if } d \text{ is a control dependence.} \end{cases} \tag{9}$$

where $rhs(s)$ is the right-hand side of an assignment s (i.e., a statement that defines a variable), $lhs(s)$ is the variable assigned at s, $cond(s)$ is the condition evaluated at a branching statement s, e propinf v indicates that an infection in some variable used by expression e causes e to infect variable v, and "c switches branch" means that an infection in some variable in condition c causes the corresponding statement to take a different branch.

The probability that a dependence propagates an infection depends on the semantics of the source statement of the dependence and the distribution of the values used in that statement. Because the effect of the first factor can be complex and the second factor is, in general, undecidable, our technique uses a simpler equation for this probability:

$$P(\xrightarrow{d}_{prop}) = \begin{cases} 1 & \text{if } d \text{ is a data dependence;} \\ \frac{N-1}{N} & \text{if } d \text{ is a control dependence.} \end{cases} \tag{10}$$

where N is the number of branches coming out of $src(d)$. This approach assumes that, for a data dependence, an infection in any operand propagates to the defined variable (which is almost always the case) and, for a control dependence, the probability of switching from one branch to another is the probability of randomly picking any of the other $N-1$ branches.

3.5 Practical Approximations

Two more issues must be addressed to make an implementation of these equation systems practical: the unknown degree of independence among events in the disjunction of Equation 1 and the complications caused by loops.

3.5.1 *Independence of Disjuncts*

Recall that Equation 1 defines the probability that a change at node C in the program dependence graph impacts a statement s directly or transitively along any dependence sequence. Although that equation defines the change-impact probabilities to all statements, the problem of computing the probability of the disjunction in Equation 1 must be solved. Unlike the coverage of control-flow successors of a node, which are exclusive events, the coverage and infection propagation through a data dependence emanating from a node is, in general, neither exclusive nor inclusive with the coverage and propagation for the other (control and data) dependencies from that node. Because computing the degree of dependence among events in that disjunction is, in general, too complex and expensive, our technique uses an approach that follows two criteria:

1. Every term in the disjunction should add to the total probability—the probability of the disjunction should be greater than the probability of each individual term. Intuitively, the more dependencies emanating from a node, the greater the chances are that an infection will propagate through any of those dependencies.
2. The independence of the terms (dependencies from a node) in the disjunction should decrease with the number of terms. The more dependencies emanating from a node, the more overlap there is in their coverage probabilities.

Based on these criteria, our technique uses the following formula for the probability of the disjunction of events whose degree of independence is unknown:

$$P\left(\bigvee_{e \in E} e\right) = \sum_{i=1}^{|E|} \begin{array}{l} \prod_{j=1}^{i-1} (1 - \frac{P(ProbSort(E,j))}{j}) \\ \times \frac{P(ProbSort(E,i))}{i} \end{array} \tag{11}$$

where E is a set of events whose independence from each other is unknown and $ProbSort(E,i)$ provides the ith event of set E in descending order of probability: e is in position i if there are exactly $i-1$ events in E with higher probabilities than e. Tied events are ordered randomly—their exact order does not affect the result.

Equation 11 satisfies our criteria because it guarantees that the result is not less than the probability of the individual terms, each term adds to the final probability, and the result does not grow uncontrolled thanks to the i divider that grows with the number of events.

3.5.2 *Loops*

When solving our equations iteratively (e.g., using a data-flow-style algorithm), the presence of loops in both the control-flow graph and the PDG causes an undesired effect: no matter how small the probability that a graph node s' is reached (or receives an infection) from a node s is for one iteration of a loop containing both s' and s, for a sufficiently large number of iterations of the loop, the probability will converge to 1. Not only this effect is unrealistic (in practice, loops do not iterate infinite times), but it also highlights that a prohibitively large number of iterations is needed to solve this system of equations. (On each iteration, the probability of an event is incremented by only a fraction.)

Our solution to the loop problem is to fix the number of times that the backedges[7] in a loop are used. Essentially, this is equivalent to unrolling each loop a certain number of times. However, because

[7] A backedge is an edge whose target has already been visited during a pre-order depth-first search when that visit reaches the source of the edge.

of the potentially prohibitive size of the unrolled control-flow or dependence graph, our technique first identifies all the backedges in the graph and then performs the following steps K times (a parameter of the technique):

1. Disable all backedges, making the graph acyclic.
2. Solve the equations for the acyclic graph in depth-first post order.
3. Enable the backedges, returning the graph to its original form.
4. Solve the equations for the graph in depth-first post-order.

In Steps 1 and 2, when backedges are disabled, the probabilities from one node to another incorporate only one new iteration of each loop. In Steps 3 and 4, when backedges are enabled (used only once each), the probabilities from nodes inside loops to nodes outside their containing loops (which in Steps 1 and 2 might have been unreachable) are updated. In all, the effect of performing these four steps K times is similar to unrolling the loops K times but without the extra memory costs. More importantly, using a limited value of K (e.g., 5 or 10) lets our technique obtain probabilities that take loops into account (i.e., an event inside a loop is more likely to occur than an identical event outside a loop) without suffering from the problem of infinite loops already described.

4. EMPIRICAL EVALUATION

The goal of our evaluation is to assess the developer effort that can be saved by using p-slices instead of simple forward slices. These savings can be determined by how accurately the relevance scores of p-slices reflect the real impacts of changes on software running in typical operational conditions (e.g., in the field after deployment) with respect to using simple slicing. The closer the p-slice scores approximate the observed impacts, the better p-slicing is for estimating the impacts that changes have.

4.1 Research Questions

To investigate the predictive accuracy of p-slicing with respect to simple forward slicing for identifying impacted statements, we addressed the following research questions:

RQ1: How effective is p-slicing for finding impacted statements?

RQ2: How effective is p-slicing for different cost intervals? (How fast can the developer find impacted statements using p-slicing?)

RQ3: What is the cost of computing p-slices?

4.2 Experimental Setup

We implemented p-slicing in Java as an extension of our DUA-FORENSICS dependence analysis and instrumentation tool [17] that uses Soot[8] to analyze Java-bytecode programs. We set the parameter K for the number of data-flow iterations (Section 3.5.2) to 5.

To run our experiments, we used a quad-core Intel machine with 12 GB RAM running Linux and Sun's JVM 6.

For the study, we used the subjects listed in Table 1. The table provides the name, description, size in Java lines of code (non-comment, non-blank), number of tests, and number of changes studied. The first subject, Schedule1, is part of the Siemens suite that we translated to Java and can be considered as representative of small software modules. The next two subjects were obtained from the SIR repository [6]. NanoXML is a lean XML parser designed for a small memory footprint and XML-security is the signature and encryption component of the Apache project.

To simulate typical operational conditions, we used the test suite provided with each subject and changes provided with the subjects.

[8] http://www.sable.mcgill.ca/soot

Table 1: Subjects, test suite sizes, and changes.

Subject	Description	Lines of Code	Tests	Changes
Schedule1	priority scheduler	290	2650	7
NanoXML	XML parser	3497	214	7
XML-security	encryption library	21613	92	7

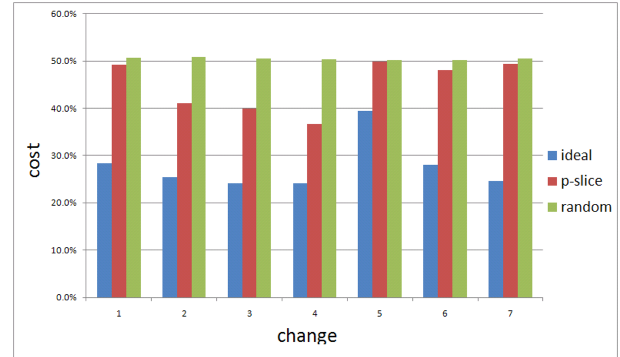

Figure 2: Examination costs for the changes in Schedule1.

4.3 Experiment Design

To measure the effort (cost) it takes to find the impacted statements within a slice, we considered a scenario in which the developer examines the statements in the forward slice in different orders (e.g., the order induced by the relevance scores of p-slicing). For each order, we computed the *examination cost* of identifying the actually impacted statements[9] as the average of the cost of reaching each of those statements when examining the slice in that order. The cost of reaching a statement s is the percentage of *statements* in the slice that must be examined before reaching s. For example, if a slice contains 1000 statements and, for some order, an impacted statement s is in the 50th position, then the cost of reaching s in that order is 5%. If, in that order, only three statements were actually impacted with costs 5%, 7.3%, and 18.9%, then the cost of using this order is 10.4% (i.e., the average of the three individual costs).

For each subject and change, we computed the p-slice and instrumented both the original and modified subjects to collect the augmented execution histories of the statements in the p-slice. We then ran the test suite for that subject and compared the collected augmented execution histories to identify the subset of statements from the p-slice that were impacted.[10]

The orders that we studied were:

1. Ideal: we ordered the statements by placing first those statements that were actually impacted. Of course, this order is not predictive but it helps identify the lowest possible cost that can be attained by any ordering strategy. We use this strategy to compare the costs of the other strategies by measuring how close to the best (ideal) case those strategies are for each subject.
2. P-slice: we ordered the statements by decreasing relevance score. When two or more statements have the same score, we used their *worst-case* cost,[11] which is consistent with the literature on fault-localization costs.
3. Random: we created 100 random orderings of the statements, and averaged their examination costs.

[9] The *actually impacted* statements are those that are impacted by a change in some execution.

[10] A statement is *impacted by a change during the execution* of the test suite if its augmented execution histories on the original and modified subject differ.

[11] For example, for a slice of size 100, if 10 statements are located at the same distance and they occupy positions 11–20 in this order, then the cost of each statement is 20%.

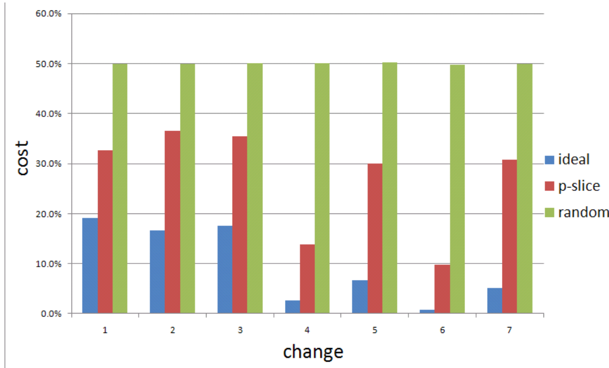

Figure 3: Examination costs for the changes in NanoXML.

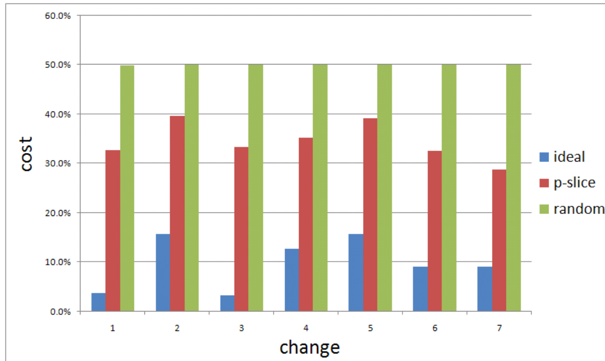

Figure 4: Examination costs for the changes in XML-security.

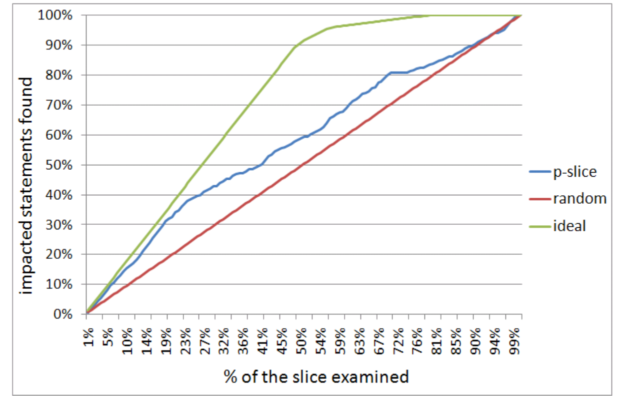

Figure 5: Impacted statements found versus cost for Schedule1.

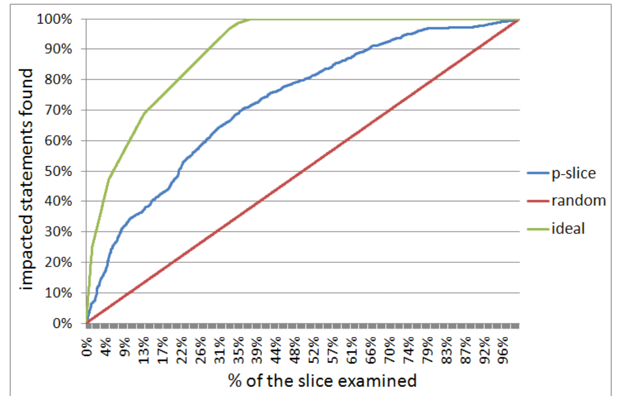

Figure 6: Impacted statements found versus cost for NanoXML.

4.4 Results and Analysis

4.4.1 RQ1: Effectiveness of p-slicing

Figures 2, 3, and 4 present the results for the costs of the order strategies applied to the changes in our subjects. On each graph, the horizontal axis represents the results for each change; changes are numbered from 1 to 7. The vertical axis represents, for each change and order strategy, the examination cost of the impacted statements as a percentage of the slice size. For each change, the first bar is the cost of *ideal*, the second bar is the cost of *p-slice*, and the third bar is the cost of *random*. The third column of Table 2 shows the average sizes of the slices per subject as percentages of the subject sizes (Lines of Code, second column).

Naturally, the ideal cost was always the least cost because no other ordering can have a lesser cost. The ideal cost was also considerably less than the other two in some cases. Also, as expected, the random order cost around 50% in all cases because, for all positions in that order, every impacted statement has the same chance of being located in that position. P-slicing, meanwhile, performed consistently better than the random order. For example, for Change 1 in NanoXML, p-slicing cost 32.6% of the slice, or just 13.5 percentage points more than the ideal (best possible) case. The random order, however, cost 30.8 points more than the ideal for this change. On average, for NanoXML, p-slicing cost 17.2 percentage points more than the ideal, whereas the random order cost 40.2 points more. This means that, on NanoXML, p-slicing was considerably better in identifying the statements that are actually impacted than randomly examining the slice.

For all changes in XML-security, p-slicing was also clearly better than the random order. On average, p-slicing cost 24.6 percentage points more than the ideal case, while the random order cost 40.1 points more.

For Schedule1, which is a much smaller subject, the ideal cost was much greater (as a percentage of the slice size) than for the other subjects because the slices have few statements and those statements are more closely related to each change—half of the statements or more in each slice were actually impacted. Hence, as expected, the average cost of using p-slicing was also greater than for the other subjects because a greater percentage of impacted statements in each slice must be reached. Also, p-slicing was not as superior to the random order as in the other two subjects, with a noticeable advantage for changes 2, 3, and 4 only. The average costs of p-slicing and the random order for this subject were 17.2 and 22.8 points over the ideal cost, respectively.

In all, the results show that p-slicing can achieve an important reduction in the effort of the developer for identifying the actually impacted statements within a slice. In particular, for the realistic subjects, NanoXML and XML-security, p-slicing resulted in an ordering that reduced, with respect to simple slicing (random order), the difference with the ideal case to almost one half.

4.4.2 RQ2: Effectiveness vs. cost intervals

Developers cannot always afford to inspect an entire slice, but they can benefit from an ordering that is cost-effective for their inspection budget. Thus, an important question is how soon impacts are found during their inspection. Figures 2–4 show the average inspection costs per change, but for RQ2 we need to visualize each cost interval separately. The graphs of Figures 5, 6, and 7 provide this information for our three subjects, respectively, as an average for all changes in each subject. In the graphs, the horizontal axes

Figure 7: Impacted statements found versus cost for XML-security.

represent the cost (i.e., the percentage of the slice examined) and the vertical axes represent the percentage of the impacted statements found. The curves in the graphs represent the percentage of impacted statements found for each point in the cost line for p-slicing and simple slicing (random order). The curve for the ideal ordering is also included to provide a perspective on the greatest cost-effectiveness that can be attained.

For example, the curves in Figure 6 show that, after examining 4% of the slice, p-slicing has covered about 20% of the impacted statements, whereas the random order has found one fifth of that number. The ideal case at that cost point is about 50%. After 4% of the slice has been visited, the slope of the p-slice curve remains greater than the slope of the random order until about 30%. After 30%, p-slicing remains on top of the random order but the difference decreases as the exploration reaches the end of the slice. The ideal curve, meanwhile, shows that the maximum effectiveness can be achieved, at best, at 38% of the slice.

For any predictive technique such as slicing (random order) or p-slicing, it is clear that 100% effectiveness will be reached only when about 100% of the slice has been examined—every statement in a slice, no matter how small its probability, has a chance of being impacted. In fact, the graphs confirm that there are impacted statements that received the lowest scores. However, the graphs also strongly confirm that the statements with the highest p-slice scores, especially those in the first 25% of the p-slice, are much more likely to be impacted than the rest of the statements. This is a strong indicator that p-slicing discriminates effectively the most relevant statements by locating 40–55% of the impacted statements in the top 25% of the p-slice. The random order, in contrast, locates only 25% of the impacted statements in that same initial cost range.

The ideal curves emphasize the benefit of using p-slicing over a random order. If the difference between the areas under the p-slice and random-order curves[12] is already considerable for the first 25% or 50% of the slice (depending on the subject), this difference is even more important because the best-case area is not the entire area under the 100% horizontal line, but the area under the ideal curve, which reaches 100% effectiveness only after 31% to 80% of the slice has been examined.

An interesting case is that of Schedule1. Even though, on average (see Figure 2), p-slicing does not provide a significant improvement, the ideal curve on Figure 5 reveals that p-slicing performs very close to the ideal case until about 20% of the slice and still closer to the ideal than to the random order until about 28%. This is another indication that p-slicing is very effective, considering its predictive nature, for the part of the slice with the greatest scores.

[12] The area under a curve reflects the accumulated effectiveness between 0% and the current point in the slice.

Table 2: Average analysis costs in seconds.

Subject	Lines of Code	Slice Size	Dependence Analysis	P-slicing Analysis
Schedule1	290	42.2%	12	1
NanoXML	3497	56.9%	16	52
XML-security	21613	64.1%	701	5796

In all, the curves for p-slicing in these graphs confirm the hypothesis of RQ2: p-slicing finds more impacted statements faster than the random ordering and it is especially fast in the first 20–25% of the slice. Thus, the results suggest that developers will experience the greatest cost-effectiveness during the inspection of the top 25% or so of the p-slice, if their budget permits.

4.4.3 *RQ3: Cost of computing p-slices*

Table 2 shows, for each subject, the average time in seconds, for all changes, required to perform the dependence and p-slicing analyses (last two columns) using our setup described in Section 4.2. The p-slicing analysis corresponds to the computation of the impact probabilities for the statements in the slice. To put these numbers in perspective, the second and third columns show the size of the subjects in lines of code and the average size of the slices relative to the subject size.

The cost of both analyses is proportional to the size of the subject. The dependence analysis, which includes the parsing, slicing, and instrumentation (to find the real impacts for our study), is small compared with the cost of p-slicing on NanoXML and XML-security. In the worst case, XML-security, this cost was slightly more than 10 minutes on average. On Schedule1, however, this cost was greater than p-slicing because of the small size of this subject and the comparatively higher cost of initializing our toolset. On NanoXML, p-slicing took less than a minute on average and less than two minutes in the worst case (not shown). On XML-security, however, p-slicing took between 1 and 2 hours per change. This result is not unexpected because p-slicing performs a data-flow analysis on the interprocedural dependence graph and such an analysis usually has a time complexity between linear and quadratic in practice. This tendency can represent a problem for p-slicing of larger subjects, although our current implementation does not include yet optimizations of data-flow analysis extensively studied in the literature, which can dramatically reduce the cost of this analysis and our technique.

4.5 Threats to Validity

The main internal threat to the validity of our study is the possibility of implementation errors in our p-slicing, instrumentation, monitoring, and reporting infrastructure. This threat is reduced by the maturity of DUA-FORENSICS, which has been in development for years. We also carefully tested, inspected, and manually validated the results of our p-slicing module for small examples and parts of the subjects used in the study.

The main external threat to our conclusions is the limited variety and nature of the subjects and changes we studied. These subjects, however, have been used extensively in previous research. Moreover, NanoXML and XML-Security are real-world programs. Thus, although we cannot generalize our conclusions, our study indicates that p-slicing can be cost-effective for predicting the real impacts that changes have.

5. RELATED WORK

Program slicing was introduced as a backward analysis for program comprehension and debugging [22]. Static forward slicing [9]

was then proposed for identifying the statements affected by other statements, which can used for change-impact analysis [4]. Unfortunately, static slices are often too big to be useful. Our work alleviates this problem by recognizing that not all statements are equally relevant in a slice and that a static analysis can estimate their relevance to improve the effectiveness of the forward slice.

Other forms of slicing have been proposed, such as dynamic slicing [10] and thin slicing [20], that produce smaller backward slices but can miss important statements for many applications. Our technique, in contrast, is designed for forward slicing and does not drop statements but scores them instead. It remains to be seen how p-slicing performs for backward slicing.

Also, in thin slicing [20], missing control dependencies can be added on demand as the user searches for the desired statement (e.g., the location of an error), but this expansion is guided by the user. Our technique, instead, starts with all statements in the static slice (which is a safe approach) and automatically estimates their relevance to help the user. Furthermore, our technique incorporates the differences in the coverage probabilities of dependencies.

Dynamic impact analysis techniques [2, 11, 14], which collect execution information to assess the impact of changes, have also been investigated. These techniques, however, work at a coarse granularity level (e.g., methods) and their results are subject to the executions observed. Our technique, in contrast, works at the statement level and analyzes the program statically to predict the impacts of changes for any execution, whether those impacts have been observed yet or not. In other words, our technique is predictive, whereas dynamic techniques are descriptive. Yet, in general, static and dynamic techniques complement each other; we intend to investigate that synergy in the future.

As already discussed in the Introduction, we are aware of only one other technique [8] that assigns probabilities to statements in a slice. This technique, however, works only on the backward dynamic slice to assess the possible contribution of components to the output in a particular execution and considers only the strongest dependence sequence from those executed. Also, the probabilities computed by this technique do not include the coverage probabilities of dependencies like our technique does.

6. CONCLUSION

In this paper, we presented Probabilistic Slicing (p-slicing), a novel technique that computes and assigns relevance scores to the statements in a static forward slice. These scores improve the information provided by a slice to developers by indicating which statements are more likely to be impacted by a change during any execution. Our technique exploits two main insights: (1) control dependencies contribute less than data dependencies to the propagation of infections (state modifications) but their contribution cannot be ignored and (2) the coverage probability of a dependence contributes directly to the probability that an infection is propagated through that dependence.

We also presented a study showing that, for a set of subjects and changes, p-slicing is more effective than pure slicing at predicting which statements are actually impacted during the execution of these subjects. This effectiveness is especially significant for those statements with the highest scores, indicating that many impacted statements are quickly isolated by p-slicing and that even a limited inspection of a forward slice by a developer can benefit considerably from using these scores.

We are currently investigating better ways to estimate coverage and propagation probabilities and more efficient implementations of our technique. In addition, we are exploring other applications of p-slicing, including test-suite augmentation [16] and debugging.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proc. of Int'l Conf. on Softw. Engg.*, pp. 432–441, May 2005.

[3] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. on Softw. Eng. and Methodology*, 4(1):3–35, Jan. 1995.

[4] S. A. Bohner and R. S. Arnold. *An introduction to software change impact analysis*. In Software Change Impact Analysis, Bohner & Arnold, Eds. IEEE Computer Society Press, pp. 1–26, June 1996.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

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

[7] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Systems*, 9(3):319-349, July 1987.

[8] T. Goradia. Dynamic impact analysis: a cost-effective technique to enforce error-propagation. In *ISSTA 93*, pp. 171–181, July 1993.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Prog. Lang. and Systems*, 12(1):26-60, Jan. 1990.

[10] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[11] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. of Int'l Conf. on Softw. Engg.*, pp. 308–318, May 2003.

[12] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Trans. Softw. Eng. Methodol.*, 19(2):1–33, 2009.

[13] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Softw. Eng.*, 16(9):965–979, 1990.

[14] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proc. of ACM Conf. on Obj. Oriented Prog. Syst., Lang., and Appl.*, pp. 432–448, Oct. 2004.

[15] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Methodology*, 6(2):173–210, Apr. 1997.

[16] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. of Int'l Conf. on Autom. Softw. Eng.*, pp. 218–227, Sept. 2008.

[17] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *Proc. of Int'l Conf. on Automated Softw. Eng.*, pp. 343–352, Nov. 2007.

[18] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proc. of Third IEEE Int'l Conf. on Softw. Testing, Verification and Validation*, pp. 429–438, Apr. 2010.

[19] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Trans. Softw. Eng. Method.*, 10(2):209–254, 2001.

[20] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proc. of Conf. on Prog. Lang. Design and Impl.*, pp. 112–122, June 2007.

[21] J. Voas. PIE:A Dynamic Failure-Based Technique. *IEEE Trans. on Softw. Eng.*, 18(8):717–727, Aug. 1992.

[22] M. Weiser. Program slicing. *IEEE Trans. on Softw. Eng.*, 10(4):352–357, July 1984.