Weighted System Dependence Graph

Fang Deng
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: fdeng@ics.uci.edu

James A. Jones
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: jajones@ics.uci.edu

***Abstract*—In this paper, we present a weighted, hybrid program-dependence model that represents the relevance of highly related, dependent code to assist developer comprehension of the program for multiple software-engineering tasks. Programmers often need to understand the dependencies among program elements, which may exist across multiple modules. Although such dependencies can be gathered from traditional models, such as slices, the scalability of these approaches is often prohibitive for direct, practical use. To address this scalability issue, as well as to assist developer comprehension, we introduce a program model that includes static dependencies as well as information about any number of executions, which inform the weight and relevance of the dependencies. Additionally, classes of executions can be differentiated in such a way as to support multiple software-engineering tasks. We evaluate this weighted, hybrid model for a task that involves exploring the structural context while debugging. The results demonstrate that the new model more effectively reveals relevant failure-correlated code than the static-only model, thus enabling a more scalable exploration or post hoc analysis.**

***Keywords*-program models; static analysis; dynamic analysis; hybrid analysis; debugging; fault-localization**

I. Introduction

A number of software-engineering tasks can utilize information about the relationship and dependency of its program elements. For example, Weiser observed that “programmers use slices” [1], which is the traversal of dependencies among program instructions. Beyond individual inter-element dependencies, researchers have proposed program models that aggregate all such dependencies in order to better enable use by other automated analyses and applications.

In this work, we present a program model informed by both static dependencies and dynamic (i.e., runtime) behavior. The resulting *hybrid* model is weighted to account for the relative importance or relevance of each dependency. The weighting of the dependencies provides further support for software-engineering tasks, both manual and automated, that use them.

Static program-dependency models, such as the program-dependence graph [2] and the system-dependence graph [3], are used by many analysis techniques that support software-engineering tasks such as debugging and testing (e.g., [4, 5]) and detection of semantic clones[6]. These models are *static*, and as such present an over-approximation of the possible behaviors that the program may exhibit at runtime for *any* possible execution. Hence, all relationships and dependencies are given equal importance, although at runtime, many dependencies may never be realized. Program slices that are performed on this model often include much of the program [7].

To account for the inability of static models to distinguish relationships that occurred during execution and those that did not, researchers proposed dynamic models, such as the dynamic slices (e.g., [8]). Dynamic slices only represent the events that actually occurred during a specific execution, thus removing the instructions that were not relevant for that execution. However, despite the fact that many instructions may be removed from the slice, a dynamic slice may still be quite large because it represents all such instructions that *transitively* affected the slicing criterion point, and it presents all instructions with equal importance.

Leveraging the research in each of these domains — static *and* dynamic — we propose a *hybrid* model that is weighted by *all* (or any subset of) executions. Consequently, the weighted dependencies in this hybrid model give the client analysis more “relevance” information in a way that enables them to be more directed, and thus more efficient. For manually performed software-engineering tasks, these weighted dependencies can help to guide exploration of the program. For software-engineering tasks that are performed with automated analyses, these weighted dependencies can inform prioritizations of these relationships to influence the computation and result.

Three examples of software-engineering tasks that may benefit from such a model include (1) debugging a program to find a fault and understand its context, (2) examining the authorship and developer expertise of code and determining the potential for impact between developers, and (3) identifying cross-cutting features within a code base that may be considered for refactoring. In the debugging task, a developer may identify an erroneous state while at a breakpoint during an execution and need to know the most relevant computations that occurred that led to this state. In the task of assessing developer relationships, the functionality that most often executes together and that affect each other can be prioritized, and as such, relationships among developers can be more accurately assessed. In the task of refactoring of cross-cutting features, the cross-cutting functionality that is disparately located in the code files, yet dependent and highly execution-correlated, can be identified and presented to the developer. In this paper, we use the debugging-context task as the running example

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.50

IEEE computer society

and the task for which we evaluate, however we also provide motivation and ideas for other software-engineering tasks for which our model may be applicable.

The main contributions of this paper are:

- A hybrid, static/dynamic model of the program that represents dependencies that are weighted according to their incident instructions' execution correlation. The model utilizes any number of executions, and these executions may be differentiated into separate classes to further support specific applications. For example, to support debugging tasks, executions may be differentiated into executions of passing and failing test cases. The model may also be augmented with domain information for the task at hand.[1]
- An evaluation of the model, which was performed on two real-world software subjects. The evaluation assesses the benefits brought by the weighting of the dependencies for the debugging task, and it determines the benefit brought by the allowance for biasing classes of execution contribution. The results demonstrate that the new model more effectively reveals relevant failure-correlated code than the static-only model, thus enabling a more scalable exploration or post hoc analysis.

II. Motivating Example

To illustrate the concept of our new hybrid model and to motivate the need for such models beyond the traditional static and dynamic models, we present an example program and client task: debugging. Programmers often need to backtrack through dependencies to track down the source of an incorrect state [1]. In fact, techniques and tools have been proposed by researchers to help in this process. After identifying that programmers manually perform this function, Weiser created an automated technique called *slicing*, which enables a programmer to specify an instruction in the program and a variable of interest (i.e., the slicing criterion). Dynamic slicing techniques were also created to allow a programmer to narrow the search space of the influence on a slicing criterion to only those instructions that had an influence during one specific execution. Ko et al. [11] created a visualization and interface that allow programmers to easily and directly query such dynamic slices. These families of techniques are described in greater detail in Section III.

While each of these techniques reduces the search space for the instructions that may have influence on the slicing criterion, the reduced search space may still account for a large portion of the program. Moreover, the results are provided in a way that offers no order or ranking of the results, and as such, little guidance on how to direct a search. Thus, we are seeking to provide such information in our new hybrid model.

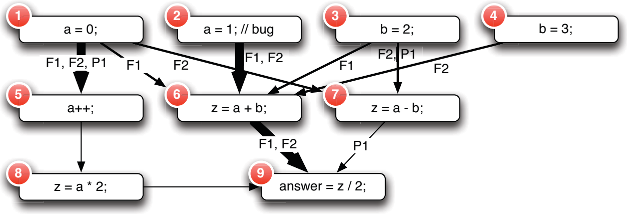


Fig. 1: Subgraph of an SDG, showing only data dependencies.

Figure 1 shows a subgraph of a system-dependence graph (SDG) for a program. To simplify the example, we show only data dependencies. Each of the instructions in this subgraph is presented with its source code and a label. For example, Node 1 has source code `a = 0;`. The program has a bug in Node 2: the instruction should read `a = 2;`.

Each of the dependency edges is labeled with the test cases that executed them — test cases *F1* and *F2* failed and test case *P1* passed. Edges that have no label are unexecuted by our test suite. Note that, because this is a subgraph, other unseen influences can affect dataflows (such as definition kills) — not all flows propagate beyond a reached node.

Consider that a programmer begins an investigation of an erroneous state on the instruction for Node 9. This location may have been identified through traditional means, such as a breakpoint, or through a state-of-the-art fault-localization technique. The programmer has inspected this line and determined that it is not the root problem, and wants to find the fault whose infection has propagated to here.

If she were to perform a static slice on Node 9 for variable `answer`, she would be returned all nodes in the subgraph shown, because every one of them is transitively reachable from Node 9. Alternatively, consider if she were to perform a dynamic slice. She could choose to slice on variable `answer` on Node 9 for either *F1* or *F2*. If she were to perform the dynamic slice on *F1*, the result would be the set {Node 1, Node 2, Node 3, Node 6, Node 9}. If she were to perform the dynamic slice on *F2*, the result would be the set {Node 2, Node 4, Node 6, Node 9}.

However, consider if we informed our slicing algorithm with the number of test cases that traversed each dependency. Additionally, given that our task is specifically debugging, we are most interested in observing the dependencies that were traversed by failing test cases. As such, we weight these edges more strongly based on their execution by these test cases — these edge weights are abstractly represented with line thickness in Figure 1. With such a weighted model, we could direct the programmer to prioritize the path ⟨Node 2→Node 6→Node 9⟩ above all others, as this dependency path is executed by both of our failing test cases.

By utilizing such a weighted dependence graph, a programmer can be aware of the degree to which a dependence is involved in failing test cases, and thus better guide the debugging traversal. We describe the model and the technique to generate it in detail in Section IV, but first provide some necessary background in the next section.

[1] This work extends our work that first presented the concept of weighted dependencies that was published at the International Conference on Automated Software Engineering (ASE) [9] and the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT) [10]. In contrast, this work provides (1) a whole-program model, (2) a more descriptive, sensitive, and configurable dependency-weighting model, (3) the incorporation of domain information, and (4) an evaluation of the model.

III. Background

To provide the necessary background to explain our approach, we overview some existing dependence models, organized into three categarizes: (1) static models, which use only program structure and no execution information; (2) dynamic models which use execution information from one test case; and (3) hybrid models which use both types of information.

A. Static Models

The *program-dependence graph* (*PDG*) and *system-dependence graph* (*SDG*) are static program models. They are used to determine which instructions in a program are related to the other instructions. The PDG is an intraprocedural model of a procedure that captures both control and data dependencies among instructions within the procedure. An instruction is said to be control dependent on another instruction if the execution result of the latter determines whether the former will be executed. For example, if instruction A may not be executed because of a predicate in instruction B, A is said to be control dependent on B. An instruction is said to be data dependent on another instruction if a value may propagate from the latter to the former. For example, if instruction C has a variable assignment (e.g., `y=10`) that over some path can reach, without redefinition, instruction D that has a reference of that variable (e.g., `print(y)`), D is said to be data dependent on C for the variable `y`. In a PDG, instructions of a procedure are represented as nodes, and control and data dependencies are represented as edges. Extending PDGs of individual procedures, the SDG is an interprocedural model of a program that enables efficient analysis of dependencies across procedure boundaries. In addition to the nodes and edges in PDGs, the SDG contains *summary* edges that account for procedure calling context. In this paper, we focus primarily on the control and data dependence edges. PDGs and SDGs can be used for computing static slices, which consist of a subset of the instructions in a program that may affect the value computed at some point of interest [12]. The construction of a static slice can be restated as a reachablity problem in the PDG or SDG.

A concern with the use of such static models is that they produce dependencies that may exist for any execution, and in fact, to be conservative, must over-approximate these dependencies. As a result, the graph is highly connected, which causes static slices to often include much of the program [7].

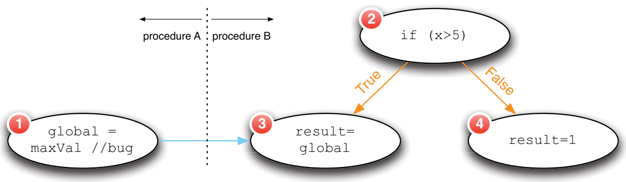


Fig. 2: Snippet from a system-dependence graph. Control dependencies are shown as orange edges, and data dependencies are shown as blue edges.

Figure 2 demonstrates both control and data dependence on a small portion of a program. In this example, a global variable, named `global`, is assigned an erroneous value in procedure A by instruction 1. This value is potentially used in procedure B by instruction 3 if the condition in instruction 2 evaluates to *true*. Because the definition of `global` may reach the use in instruction 3, an edge is drawn to represent that instruction 3 is data dependent on instruction 1. Because the execution of instructions 3 and 4 are each dependent on the outcome of the predicate in instruction 2, an edge is drawn from instruction 2 to instruction 3 with the label "true" to indicate that instruction 3 is control dependent upon instruction 2's evaluation of *true* (and likewise for the control dependence between instruction 2 and 4 for the *false* condition).

B. Dynamic Models

To address the imprecision caused by the conservative nature of static models to account for any possible execution, researchers proposed dynamic slicing [8], which presents the instructions that actually affected a point in the program for a specific execution. Like static slices, dynamic slices include all instructions in the transitive closure of dependencies. As such, they can also include much of the program, which may cause challenges in its use.

To address this challenge, Ko et al. developed an interactive tool to enable dynamic-slice querying [11]. With their tool, WhyLine, a programmer can ask "why did" and "why didn't" questions about the output of the program, and thus step backward, one dependency at a time, through the dynamic slice. By providing the ability to interact with and step through the slice, the usability of the dynamic is improved. However, the dynamic slice on which the interface is built remains potentially large, and only considers a single execution, thus precluding the potential to automatically distinguish relevant or useful dependencies.

C. Hybrid Models

Models that utilize both static and dynamic information have been deemed *hybrid* models. For example, Gupta and Soffa [13] proposed a hybrid slicing technique that allows static slices to be refined by dynamic information. Choi et al. [4] proposed a hybrid analysis that uses static information to improve the performance of dynamic slicing. While these models and analyses utilize both static and dynamic information to enable benefits that neither type of information afforded on its own, they do so by either utilizing a single execution or treating all executions as a union of one execution.

IV. Weighted System Dependence Graph

In this section, we propose a new model — Weighted System Dependence Graph (WSDG) — which bridges the static representation of the SDG with dynamic information from multiple executions. The new hybrid model reveals the degree of interest of each dependency in an SDG using commonly available and lightweight dynamic instrumentation. Although we envision a number of automated software-engineering tasks that can be benefited by such a program model (described in Section VIII), we continue to use the debugging task to describe the details of the technique to build the model.

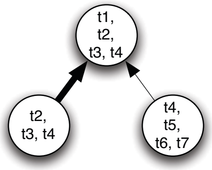


Fig. 3: Three instruction nodes labeled by the test cases that covered them. A dependency edge whose incident nodes have a higher set similarity of executions is assigned a greater weight.

A. Weighting of Dependencies

To describe how we weight the edges of an SDG, we define two terms: *co-execution* and *co-execution edge weight*. Two instructions, i_1 and i_2, where i_2 is dependent (either control- or data-dependent) on i_1, are co-executed if a test case *t1* executes both.

We chose to use instruction (i.e., statement) coverage to inform our weighting for the following reasons: (1) tools to instrument and produce statement coverage are readily available, and thus improve the chances that such a model can be more immediately integrated and used in research and practice; (2) the instrumentation is lightweight, and thus will not impose noticeable runtime overheads; and (3) it provides a baseline upon which future weighted, hybrid models can compare, in order to assess the trade-offs in efficiency and practicality. As such, two instructions i_1 and i_2 that are dependent and co-executed may or may not actualize the dependency between them. However, the likelihood is increased with each co-execution according to new test cases. With this intuition, we describe the co-execution edge weight.

The weight for a particular dependency edge, based on co-execution, is defined by the degree to which the same executions executed each of its incident nodes. We calculate the degree to which the same executions executed each of the dependent nodes using the Jaccard similarity coefficient [14]. The co-execution edge weight is calculated using the following formula

$$\text{co-execution edge weight}\,(E_{i_1}, E_{i_2}) = \text{Jaccard}\,(E_{i_1}, E_{i_2}) = \frac{|E_{i_1} \bigcap E_{i_2}|}{|E_{i_1} \bigcup E_{i_2}|} \qquad (1)$$

where i_1 and i_2 are two instructions with a dependency between them, and E_{i_1} and E_{i_2} are the sets of executions that executed i_1 and i_2, respectively.

For example, Figure 3 shows three SDG instruction nodes, each of which is labeled with the test cases that executed it. The edge between nodes $\langle t1, t2, t3, t4\rangle$ and $\langle t2, t3, t4\rangle$ has a weight of $\frac{3}{4}$, and the edge between nodes $\langle t1, t2, t3, t4\rangle$ and $\langle t4, t5, t6, t7\rangle$ has a weight of $\frac{1}{7}$. As such, the former edge has a higher co-execution edge weight, and the dependency that it represents is considered more likely to have been realized in execution.

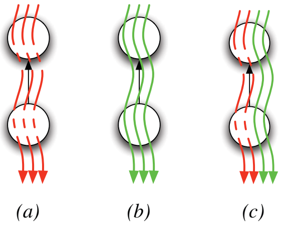


Fig. 4: Three possibilities of one dependency being traversed by different test cases. (a) A dependency traversed by three failing test cases. (b) A dependency traversed by three passing test cases. (c) A dependency traversed by two failing test cases and two passing.

B. Configurable Weighting Schema

With this basic formulation, we further extend the edge-weighting concept to target specific tasks by allowing for classes of executions to be differentially treated. Consider, for example, the debugging task presented in our motivating example in Section II. In this context, in order to understand why test cases failed, it is likely that the programmer who is exploring the program would prefer to be guided along paths that were executed primarily by failing test cases. Figure 4 exemplifies this issue: each subfigure depicts the same dependency executed by different combinations of passing and failing test cases (red for failing and green for passing). Should Figure 4b have the same weight as Figure 4a because it was executed by the same number of test cases, or should Figure 4a have more weight because it is implicated in more failures? Should Figure 4c be weighted more strongly than Figure 4a because more test cases executed that dependency, or should it be less because it is implicated in fewer failures?

In our approach, we allow for variable weighting schemes for different classes of executions. We independently calculate the co-execution edge weight for each class of executions. We then scale each class according to the client-analysis-defined contribution coefficients for each class. Finally, we sum the scaled co-execution weight contributions for each class to achieve the resulting weight.

V. Empirical Studies

To enable evaluation of our WSDG model, we targeted one application analysis: the debugging task described in the motivating example in Section II. To evaluate the effectiveness of our WSDG model, we conducted an experiment by which we independently varied factors of the technique to determine their effects on the ability to distinguish failure-correlated instructions in the program. In assessing the ability to distinguish such instructions, we sought to identify if the weighted dependencies gave higher weight to dependencies that lead to relevant instructions, which can be used to cluster those instructions to allow for greater scalability. Effectively, clustering related instructions provides a means by which users or client analyses can distinguish and prioritize appropriate actions. To this end, we posed three research questions:

RQ1: Do system-dependence graphs (without the augment of dynamic information) effectively cluster failure-correlated instructions?

RQ2: Does our dependence weighting in the WSDG improve the effectiveness of clustering failure-correlated instructions?

RQ3: Does independent weighting of dependencies to account for the influence of each passing and failing test cases affect the clustering of failure-correlated instructions, and if so, how?

A. Variables and Measures

The primary objective of our studies is to examine the degree to which failure-correlated code "clusters" around the faulty lines in the program that are causing failures. We define successful clustering as instructions that are highly correlated with failure have shorter distances over system-dependence graph edges than less failure-correlated instructions.

To address each of our three research questions, we define our **dependent variable** as the failure-correlation of the surrounding instructions in the SDG. As an assessment of failure-correlation, we utilize the Ochiai metric [15], which is a similarity metric that can assess the correlation of one dimension of data to another. The failure-correlation of an instruction i is computed with the following formula:

$$\text{failure-correlation}_{\text{Ochiai}}(i) = \frac{\text{failed}(i)}{\sqrt{\text{totalfailed} * (\text{failed}(i) + \text{passed}(i))}} \quad (2)$$

where *passed*(i) is the number of passed test cases in which instruction i is executed, *failed*(i) is the number of failed test cases in which i is executed, and *totalfailed* is the number of failed test cases in the test suite.

Our **independent variable** is defined by distance from the faulty instructions. To compute distance, we utilized the single-source-shortest-path algorithm (also known as *Dijkstra's algorithm*)[16]. Dijskstra's algorithm takes a graph and a source node as input. Each edge in the graph is assigned with a variable length. In our case, the distance of an edge is derived from its weight. The weight, as computed by Equation 1, is inverted: a stronger weight maps to a short distance and a weaker weight maps to a further distance. Dijkstra's algorithm finds the shortest path between the source and every other node in the graph. Because the faults may consist of multiple lines of code, we compute the shortest length from each instruction to any of the faulty lines.

B. Objects of Analysis

We used two popular C-language programs for coverage-based fault localization: Space and Gzip (version 1.0.7). The Gzip program is composed of 7928 lines of code and has a test suite that is composed of 214 test cases. The Space program is composed of 9564 lines of code and has a test suite that is composed of 13527 test cases. They were obtained from the "Subject Infrastructure Repository" (SIR) [17] along with faults and test cases. We use 20 faulty versions of Gzip and 34 faulty versions of Space, each containing a single fault. To achieve this quantity of faults for Gzip, we augmented the faults provided by the SIR by random mutation. The operators described by Offutt et al. [18] and the locations for mutation were randomly selected and applied.

C. Experimental Setup

For our experiments, we performed the following steps:

1) Execute the test suite, capturing coverage information.
2) Calculate the failure correlation of each instruction.
3) Compute the system-dependence graph of the program.
4) Compute dependency weights.
5) Calculate instruction distance from the faulty instructions.

We describe in detail the process for each step.

Step 1: Execute the test suite, capturing coverage information. We ran the test suite for our each of our subject programs, Space and Gzip, for every faulty version. We captured the coverage information (i.e., which instructions were executed by which test cases) by using the instrumenter built into the GCC compiler and the Gcov coverage reporting tool.

Step 2: Calculate the failure correlation of all instructions. We used the test results and coverage to compute the failure correlation for each instruction with the Ochiai metric, which is represented in Equation 2.

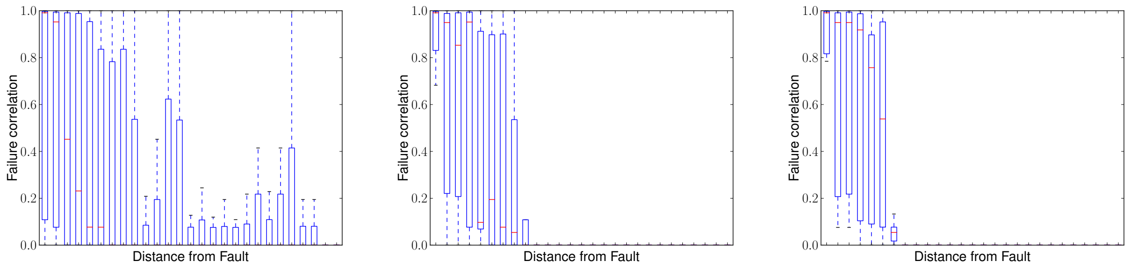
Step 3: Compute the system-dependence graph of the program. To compute the system-dependence graph, we used the CodeSurfer tool by GrammaTech [19]. When control or data flow can occur within a single line of code, CodeSurfer can provide multiple nodes for a line. We performed a merging of these same-line nodes to create a single node for each line. The resulting graph contains a node for each executable line.

Step 4: Compute dependency weights. We calculate the weight for each dependency edge given the technique described in Sections IV-A and IV-B, using the Equation 1. When evaluating the static-only SDG, we instead assign a constant weight for every dependency edge.

Step 5: Calculate instruction distance from the faulty instructions. Using the weights assigned in the previous step, a distance was assigned to each dependency edge. The edge distance is defined by simply inverting the weight. The weight of each edge is subtracted from the maximum weight across all nodes. As such, a large weight produces a small distance. Using these edge distances, Dijkstra's algorithm is utilized to determine the distance of every instruction from the faulty instructions.

D. Aggregated Results

Utilizing the process described in Section V-C, we assessed the failure correlation of the instructions at various distances from the faults in each of the faulty versions for each of our subject programs. Figures 5 (a), (b), and (c) present



(a) Static SDG

(b) WSDG

(c) WSDG, failing contribution > passing

Fig. 5: Boxplot for failure correlation across distances for all faulty versions of Gzip. The first box represents the closest 100 instructions to the faulty instructions across all 20 Gzip versions. Each subsequent box represents the next 100 instructions to the faulty instructions.

our results in boxplot format.[2] These boxplots represent the failure-correlation values of all instructions on the vertical axis and their distance to the fault on the horizontal axis. The instructions are organized into groups according to their distance from the fault and ordered along the horizontal axis. For an individual version, each group is comprised of 100 instructions: the closest 100 instructions to the fault are represented in the first box in the boxplots, the next closest 100 instructions are represented in the next box, and so on. Across all versions, these groups are aggregated. To exemplify this point, the first box for the Gzip program represents the closest 100 instructions across all 20 Gzip versions — the result is that the first box represents 2,000 instructions.

The boxplots in Figure 5 represent the results for the Gzip program. Figure 5a represents the failure correlation as a function of distance when utilizing a static-only SDG. In this figure, we observe that while the failure correlation does decrease as the distance is increased, a large degree of variability persists into the greater distances. Figure 5b represents the failure correlation as a function of distance when utilizing the basic, unbiased, weighting model for the WSDG from Section IV-A. In this figure, we observe that failure correlation decreases at a greater rate in the WSDG than the static-only SDG. Figure 5c represents the failure correlation as a function of distance when utilizing a larger weight for failing test cases than passing test cases to inform the WSDG. In this figure, we observe a higher median correlation persists for the closest 700 instructions, and at greater distances, produce only non-correlated instructions.

The boxplots in Figure 6 represent the results for the Space program. Figure 6a represents the failure correlation as a function of distance when utilizing a static-only SDG. In this figure, we observe that while the failure correlation does decrease as the distance is increased, a large degree of variability persists into the greater distances, even to a greater degree than that seen for the Gzip program. Figure 6b represents the failure correlation as a function of distance when utilizing the basic, unbiased, weighting model from Section IV-A to inform the WSDG. In this figure, we observe that failure correlation decreases at a greater rate in the WSDG than the static-only SDG. Figure 6c represents the failure correlation as a function of distance when utilizing a larger weight for failing test cases than passing test cases to inform the WSDG. In this figure, we observe that while results do not change drastically from the basic model in Figure 6b, they do decrease at a greater rate than the basic model.

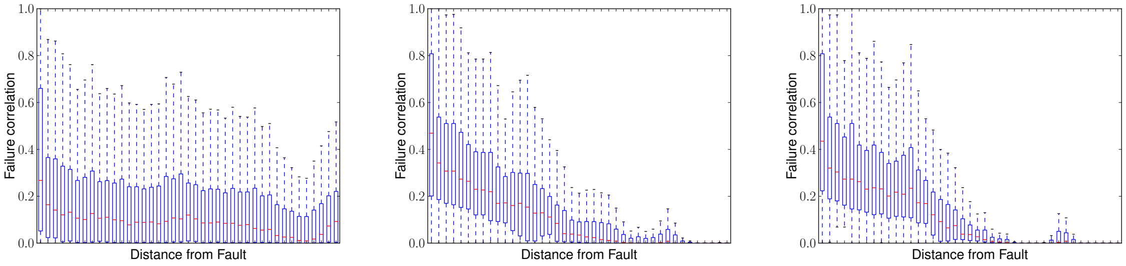
E. Individual Results

To view our results in a non-aggregated way (i.e., for individual versions) to see whether failure-correlated instructions truly cluster around the fault, and to give a more nuanced perspective of the model, we generated a visualization of the SDG and WSDG for specific faulty versions. Figure 7 shows the visualization for a faulty version of Gzip. In this visualization, the nodes are drawn as points with the color that was assigned to that instruction informed by TARANTULA using the OCHIAI [15] suspiciousness metric. The layout is informed by the static structure of the SDG as well as the dynamic information described in Sections IV-A and IV-B. The placement of nodes is determined by a force-directed graph-layout algorithm (e.g., [20]). The force between two nodes is derived from the weight of the edge that connects the two nodes (as defined by Equation 1). This visualization, along with a user study that evaluates it, are described in detail in a recent workshop paper [10].

To demonstrate how the choice of static-only SDG, basic WSDG, or failure-biased WSDG affects a particular program, we have generated three instances of the visualization for Gzip with one single fault. While some versions produced results that were more or less pronounced than others, we chose a faulty version that was representative of the overall results (represented in Figure 5). In these figures, we are representing the instructions that are faulty as a larger, square node.

Figure 7a represents the static-only graph. In this instance of the visualization, the unexecuted instructions (gray nodes) are highly interspersed with the executed instructions — these unexecuted instructions are indeed executable as we do not in-

[2] A boxplot is a standard statistical device for representing data sets. In these boxplots, each data set's distribution is represented by a box. The box's height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The line within the box represents the median. The vertical lines attached to the box indicate the tails of the distribution.



(a) Static SDG

(b) WSDG

(c) WSDG, failing contribution > passing

Fig. 6: Boxplot for failure correlation across distances for all faulty versions of Space. The first box represents the closest 100 instructions to the faulty instructions across all 34 Space versions. Each subsequent box represents the next 100 instructions to the faulty instructions.

clude any unexecutable instructions in our visualization (such as comments or blank lines). Some of the failure-correlated instructions (red nodes) do cluster around the fault (in the upper-left corner), but they are interspersed with unexecutable instructions. Other failure-correlated instructions do not cluster and are interspersed with many other instructions of varying degrees of correlation.

Figure 7b represents the visualization using the basic, unbiased, weighting model from Section IV-A. In this figure, we observe that the unexecuted instructions are unclustered, while the executed instructions are highly clustered. The failure-correlated instructions cluster to a greater degree than the static view, however they are still interspersed with the less-correlated instructions.

Figure 7c represents the visualization when utilizing a larger weight for failing test cases than passing test cases. In this figure, we observe that the unexecuted instructions again are unclustered, the failure-correlated instructions are highly clustered, and the less-correlated instructions are loosely clustered.

VI. Discussion and Analysis

In this section, we reflect upon the results presented in Sections V-D and V-E and address our research questions posed in Section V. Overall, we observe from the boxplots in Figure 5a and Figure 6a and the visualization in Figure 7b that the static-only SDG does perform a certain degree of clustering of the failure-correlated code. Although, we also observe that it is intermixed with other less failure-correlated code as well.

To RQ1, we assess that using the static-only SDG, the failure-correlated code loosely clusters around the fault, with a limited degree of effectiveness.

We also observe from the boxplots in Figure 5b and Figure 6b and the visualization in Figure 7b that the WSDG does cluster more effectively than the static-only SDG. This difference is most striking for the results shown for the Space program in Figures 6(a) and (b). In these, we observe how for the static SDG the median is relatively low close to the fault and only drops slightly while extending the context away from the fault, and the variance of the failure-correlated code remains wide throughout the entire program. Comparatively, for the WSDG, the correlation of the nearest instructions is greater (which is desired) and drops as the distance from the fault is increased (which is also desired), thus demonstrating a better clustering for the failure-correlated code. This difference is also observed in the boxplots for Gzip in Figures 5(a) and (b) and evidenced in the visualizations in Figures 7(a) and (b).

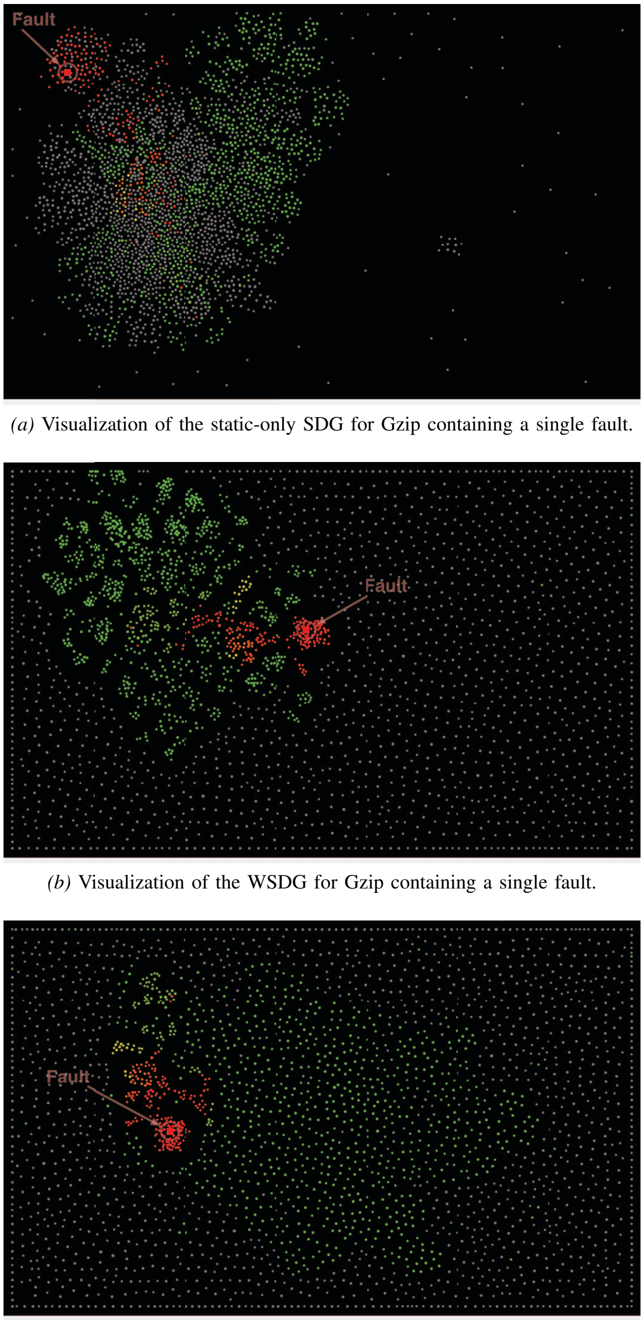
To RQ2, we assess that the WSDG does improve the ability for the failure-correlated code to cluster around the fault over the traditional SDG.

When comparing the results between the WSDGs with the failure-biased WSDGs, we observe that the failure-correlated code does cluster more effectively, although the degree of improvement is to a smaller degree than the improvement that we observed in going from a static SDG to the dynamically weighted one. In comparing Figures 5(b) and (c), we see fewer high-variance boxes in the basic WSDG than the failure-biased one. This is due to the presence of failure-correlated code in the further distances mixed with low-failure-correlated code at those same distances. In the failure-biased boxplot of Figure 5c, the median correlation value remains high while traversing away from the fault until it precipitously drops to low correlation values, thus demonstrating a relatively high degree of clustering. The results for the Space program (in Figures 6(b) and (c)) are less dramatic, but still observable at the medium to higher distances. Similarly, in the visualizations shown in Figures 7(b) and (c), the red, suspicious code clusters more effectively and is less intermixed with non-suspicious code in the failure-biased version.

To RQ3, we assess that the failure-biased WSDG does affect the ability for the failure-correlated code to cluster around the fault, and in fact slightly improves its effectiveness.

VII. Threats to Validity

Threats to internal validity arise when factors affect the dependent variables without the researchers' knowledge. It is possible that some implementation flaws could have affected the results. However, in addition to diligence in checking and testing our code, we have evidence for its correctness — namely, the consistency of the results that we find with the boxplot generation and the visualization. The results for



(a) Visualization of the static-only SDG for Gzip containing a single fault.

(b) Visualization of the WSDG for Gzip containing a single fault.

(c) Visualization of the WSDG for Gzip containing a single fault, giving a greater contribution to failing test cases than passing.

Fig. 7: Visualization for Gzip with a single fault. The fault is represented by a larger, square node. Please note: this visualization relies heavily upon color. If viewing a grayscale print, an electronic copy will be a useful reference.

each subject program is consistent between these two representations: we see a stark clustering and drop-off of failure correlation for Gzip and a more varied and gradual drop-off for Space, in both the boxplot and visualizations.

Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the technique on only two programs, and thus we are unable to definitively state that our findings will hold for programs in general. We attempted to address some of these uncertainties by performing our evaluation programs of real-world origin. For each program, we also experimented with many faults.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. In our case, we measure the relevant context for the fault by assessing the failure-correlation value measured by the Ochiai metric. The Ochiai metric assesses the degree of correlation of an instruction's execution with failure. The failure correlation of an instruction gives an indication of the relevance for an instruction to a debugging task, but such indications may not reflect true cognitive relevance for the developer. However, in future work, observational studies of many actual developers of varying degrees of expertise are needed to determine an effective way to assess context.

An additional threat to construct validity is that our algorithm for computing path distance does not account for feasibility of the paths. As such, some paths that link pairs of nodes may be regarded as a short path when the truly feasible path may actually have a longer path. We do not consider this to be a serious threat because the goal of our evaluation is not to gauge the precise length of any particular path. Instead, the goal is to assess the relative distance of groups of nodes. Such path-feasibility issues arise both for fault-correlated code and non-correlated code, and as such, the distant instructions in our evaluations are truly farther, whether over feasible or infeasible paths. That is, it is just as likely that non-correlated code can be reached over infeasible paths as correlated code. Our experimental result that the correlated code is nearer to the fault indicates that the model correctly relates those groups of instructions. Future WSDG-based analyses that prescribe specific paths will need to account for path feasibility.

VIII. Applications

In this paper, we primarily used the debugging task to demonstrate the benefit of our model as well as describe how it can be configured to support that task. Particularly, we showed how a static dependency model was less effective at grouping relevant code for debugging exploration. Moreover, we described how the executions can be classified by its test-case pass/fail status so as to treat each class differently.

In addition, we envision a myriad of applications that can utilize such a weighted, hybrid dependency model. To inspire the reader to consider such applications, we present two additional applications that we foresee.

The second application task that we envisage is one that identifies developer-to-developer relationships and potential for change impact. Change-impact analysis (such as [21, 22]) is a highly researched topic because it seeks to minimize inadvertent bugs by identifying locations in code that could impact other code. Recently, Begel et al. [23] developed a social-networked model of developers that leverages information about code authorship and relationships within the code. The current model is static, i.e., it does not consider whether statically dependent code ever, in practice, executes together. Using a hybrid static/dynamic model, such as the WSDG, we can imagine more accurate correlations between code and thus developers, which in turn may enable project managers to better understand whose code can affect whose, as well as enabling individual developers to be more aware of those with whom they should communicate.

The third application task that we envisage is one that can identify cross-cutting features — these are often referred to as *aspects*. Such features can be extracted and identified for purposes of testing, comprehension, and refactoring. We can imagine that an execution-influenced static model of the program can identify "clusters" of code that often co-execute. Additionally, we imagine cases where test cases can be classified according to the functionality that they test. Consider biasing user-interface tests with a strong contribution, and all others with a weak contribution. The result would likely cause disparately located user-interface code to cluster together so that the developer can more easily find, explore, and contextualize this code, and also potentially refactor it to make it more maintainable in the future.

While each of these applications would need to be evaluated separately, we see such areas as promising directions for future work. A first approximation of application tasks that can utilize such a hybrid model may be all such techniques that currently either utilize static or dynamic code dependencies, of which there are many.

IX. Related Work

Because the application task using our new model that we highlighted in this paper supports the exploration of relevant code for debugging, we focus much of our related work on providing context for debugging. Multiple researchers have developed techniques that provide context for debugging tasks. In this section, we briefly survey this research.

Slicing techniques (e.g., [8, 12]) have been developed that determine the instructions that propagate influence to or from a particular point in the program. These techniques can be used to provide context. Static-slicing techniques select all instructions that could provide influence over any test case, while dynamic-slicing techniques select instructions that provide influence for a single test case. Much in the same way that we utilize many test cases to inform a model that represents dependency information among program elements, union slicing [24] generates a union of the dynamic slices for multiple test cases. However, it does not provide any weight information to give a sense of a stronger or weaker dependency for the executions that informed the dynamic slices. Each of these techniques can produce slices that include a substantial portion of the program. We provide a weighted hybrid model that enables a guided walk over the dependencies.

Ko and Myers [11] developed an interface, called WHYLINE that builds on dynamic slicing and enables querying and traversal of them. WHYLINE utilizes a single execution trace while our approach uses the execution information from multiple test cases (up to the whole test suite). By considering multiple executions, we can guide developers to those locations that highly correlate to failure across all the test cases.

Masri [25] developed a fault-localization technique based on information-flow coverage profiles. An information flow is a chain that comprises dynamic data and control dependencies. Hsu [26] et al. developed a fault-localization technique called RAPID which can identify bug signatures to assist debugging. This technique examines failing test case traces and identifies common subsequences in these traces. Developers can examine these common subsequences to explore for the actual bugs. Jiang and Su [27] developed a fault-localization technique that provides potential faulty control-flow paths. Chilimbi et al. [28] proposed a fault-localization technique called HOLMES that utilizes intraprocedural acyclic path profiles. Each of these techniques provide multiple fault diagnoses, which are ranked. For each, the developer is provided with context within the fault diagnosis, which may or may not include the fault. The locations in these diagnoses are fixed — a particular location is either in a diagnosis or not. In contrast, our model does not dictate which fault-localization technique is used. Our model not only enables any of these localization techniques to be used, but allows for exploration within and around the diagnoses. Our studies [10] found that such techniques provide useful origins for exploration.

Cheng et al. [29] mined program-behavior graphs to identify the most discriminative subgraphs that contrast flow of correct and faulty executions. This approach provides context and informative relationships within the subgraph. We think that such a subgraph can benefit our approach for providing an origin for exploration. Our model provides the context outside of the subgraph as well as informing the strength of the relationships both within and around the subgraph.

Finally, Baah et al. propose a Probabilistic Program Dependence Graph (PPDG) [30], which enables the calculation of the probability of the correctness of an execution based on learned execution-state changes over previously learned correct executions. As such, the model can be used to identify error conditions and localize faults. The PPDG is informed by heavier-weight, execution-state-tracing instrumentation. The PPDG provides a more descriptive and thorough model of program execution than the WSDG, however that expressiveness comes at a cost. Although the runtime overhead of their state-tracing instrumentation is not revealed, the reported time to build the model is over five minutes for a program with only 512 lines of code. As such, the authors concede that PPDG is limited to smaller and non-deployed software [30, p. 544]. Comparatively, the WSDG requires only the one-time cost of building the static SDG, and the per-test-case runtime overhead of statement coverage, which is negligible (and already performed by many testing tools). For our subject programs that are over ten times as large, the static-SDG construction required roughly four seconds.

X. Conclusions

In this paper, we present a new hybrid model that provides program-dependence information with weights that indicate relevance. The model — the Weighted System Dependence Graph (WSDG) — is informed by static information and the dynamic information of any number of executions. When client applications are built upon our model, they can leverage these weights to inform paths of exploration for the developer (i.e., which instructions should be visited before others). In addition, the model can be customized for specific applications and analyses by categorizing execution information and differentially using those categories for dependency weighting. Task-specific meta-data, such as fault-localization results, can also be attached to nodes to support those specific tasks.

We illustrate the use of our model for the task of debugging in which failure-related code must be explored to find, understand, and fix the faults that cause failures. Further, we discussed additional software-engineering tasks that can utilize such a weighted, hybrid dependence model, and identified how the model can be customized to suit those tasks.

We present an empirical evaluation that demonstrated that our WSDG model provides relevant context for instructions of interest. An experiment was conducted that compared the WSDG with the static-only SDG. The results of our experiment demonstrated that the WSDG was able to (1) produce more relevant context (i.e., more effective), in (2) fewer instruction nodes (i.e., more scalable). Additionally, by applying task-specific customizations of the dependency weighting, the effectiveness can be even further improved.

In future work, we will extend our studies to include more subject programs and compare our model to related approaches. We will also investigate the use and effectiveness of the WSDG for additional software-engineering tasks. Finally, we will implement more accurate, yet more expensive and less commonly-available, instrumentation to determine the trade-offs that they incur.

XI. Acknowledgements

This material is based upon work supported by the National Science Foundation under award CCF-1116943, and by a Google Research Award.

References

[1] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, pp. 446–452, July 1982.

[2] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *SIGPLAN Not.*, vol. 19, no. 5, pp. 177–184, 1984.

[3] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 26–60, January 1990.

[4] J.-D. Choi, B. P. Miller, and R. H. B. Netzer, "Techniques for debugging parallel programs with flowback analysis," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 491–530, October 1991.

[5] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1993, pp. 384–396.

[6] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *International Conference on Softw. Eng.*, 2008, pp. 321–330.

[7] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering and Methodology*, vol. 16, April 2007.

[8] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, pp. 155–163, October 1988.

[9] F. Deng and J. A. Jones, "Inferred dependence coverage to support fault contextualization," in *International Conference on Automated Software Engineering*, 2011, pp. 512–515.

[10] F. Deng, N. Digiuseppe, and J. A. Jones, "Constellation visualization: Augmenting program dependence with dynamic information," in *International Workshop on Visualizing Software for Understanding and Analysis*, Sep 2011, pp. 1–8.

[11] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *International Conference on Software Engineering*, 2008, pp. 301–310.

[12] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 352–357, July 1984.

[13] R. Gupta and M. L. Soffa, "Hybrid slicing: an approach for refining static slices using dynamic information," in *Symposium on Foundations of Software Engineering*, 1995, pp. 29–40.

[14] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[15] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques*, 2007, pp. 89–98.

[16] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[17] H. Do, S. Elbaum, and G. Rothermel, "Infrastructure support for controlled experimentation with software testing and regression testing techniques," in *International Symposium on Empirical Software Engineering*, 2004, pp. 60–70.

[18] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, pp. 99–118, April 1996.

[19] *CodeSurfer*, GrammaTech, www.grammatech.com, Ithica, NY.

[20] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Softw. Pract. Exper.*, vol. 21, no. 11, pp. 1129–1164, 1991.

[21] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004, pp. 432–448.

[22] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *International Conference on Software Engineering*, 2005, pp. 432–441.

[23] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *International Conference on Software Engineering*, 2010, pp. 125–134.

[24] Á. Beszédes, C. Faragó, Z. M. Szabó, J. Csirik, and T. Gyimóthy, "Union slices for program maintenance," in *International Conference on Software Maintenance*, 2002, pp. 12–21.

[25] W. Masri, "Fault localization based on information flow coverage," *Softw. Test. Verif. Reliab.*, vol. 20, pp. 121–147, June 2010.

[26] H.-Y. Hsu, J. A. Jones, and A. Orso, "Rapid: Identifying bug signatures to support debugging activities," in *International Conference on Automated Software Engineering*, 2008, pp. 439–442.

[27] L. Jiang and Z. Su, "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *International Conference on Automated Software Engineering*, 2007, pp. 184–193.

[28] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *International Conference on Software Engineering*, 2009, pp. 34–44.

[29] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *International Symposium on Software Testing and Analysis*, 2009, pp. 141–152.

[30] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 528–545, July 2010.