MOAD: Modeling Observation-based Approximate Dependency

Abstract—**While dependency analysis is foundational to many applications of program analysis, the static nature of the existing technique presents challenges such as limited scalability and inability to cope with multi-lingual systems. We present a novel dependency analysis technique that aims to approximate program dependency from a relatively small number of perturbed executions. Our technique, called MOAD (Modeling Observation-based Approximate Dependency), reformulates program dependency as the likelihood of one program element being dependent on another based on a set of observations, instead of a Boolean relationship. MOAD generates a set of program variants by deleting parts of the source code, and executes them while observing the impacts of deletions on various program points. From these observations, MOAD infers a model of program dependency that captures the dependency relationship between the modification and observation points. While MOAD is a purely dynamic dependency analysis technique similar to Observational Slicing, it does not require iterative deletions for a single slicing criterion. Rather, MOAD makes a much smaller number of multiple, independent observations in parallel and infers dependency relationship for multiple program elements simultaneously, significantly reducing the cost of dynamic dependency analysis. We evaluate MOAD by instantiating program slices from the obtained probabilistic dependency model and comparing the slices to the results of Observational Slicing. The results show that, with a certain configuration, MOAD requires only 18.7% of the number of observations to construct the model; the size of the slice generated by MOAD is 16% larger than the slice from ORBS, on average.**

I. Introduction

Understanding dependencies between program elements is a fundamental task in software engineering [1], [2]. It provides a basis for many software engineering tasks including program comprehension [3], software testing [4], maintenance [5], [6], refactoring [7], security [8], and debugging [9]. The traditional static approach based on dependency graphs [10] has been widely adopted but suffers from issues such as its inability to handle multi-lingual systems (combining analyses for multiple languages is too complicated) and limited scalability (partial analysis between parts of a large system is not easy using static approaches that only work with full analysis of the entire system).

Observation Based Slicing (ORBS) [11]–[14] was designed to overcome these issues. ORBS applies speculative deletions iteratively to the program under analysis, and observes whether the latest applied deletion is viable (i.e., the code compiles after deletion) and is unrelated to the dependency (i.e., the variable of interest shows the same behaviour after deletion w.r.t. a test suite). When deletions are made at the line level, ORBS can be entirely language agnostic [11] and can analyse files for which the grammar is unavailable/unknown and languages with unconventional semantics, e.g. Picture Description Languages (PDLs) [15]. Despite its benefits, ORBS has one clear drawback: the cost of analysis. Being a purely dynamic approach, it iteratively attempts to validate its speculative deletion of program elements via compilations and test executions. As such, it can incur a significant cost.

This paper investigates whether it is possible to perform *approximate dependency* analysis dynamically *at a lower cost*. A precise dependency analysis can report whether program element A depends on program element B or not: the outcome is Boolean. An approximate dependency analysis instead reports the likelihood that A depends on B based on the observations it has been given: the outcome is a real number. While probabilistic program dependency analysis techniques have been proposed before [2] they require an initial static analysis which is then extended with probabilistic information based on test executions. We conjecture that a more general, approximate dependency analysis based on dynamic observations can still be useful in many program analysis contexts while being significantly less costly.

The approximate nature of MOAD stems from the fact that it *infers* a stochastic model of program dependencies. Unlike ORBS which performs iterative deletions to analyse program dependency with respect to a single program element (i.e., the slicing criterion), our technique, MOAD (Modelling Approximate Dependency), learns an approximate model of program dependencies for multiple program elements from a smaller number of dynamic observations. Intuitively, ORBS makes a single slice increasingly more accurate by iteratively applying deletions. MOAD, however, applies deletions that are (often) independent from each other, and makes observations for multiple program elements for each execution. This approach introduces the following benefits:

- MOAD requires many fewer observations than ORBS, as it *infers* the relationships between individual deletions and thus the dependency, instead of constructing the dependency by iteratively deleting until it gets an one-minimal slice [11].
- The output of MOAD can be used to construct multiple slices, whereas a single ORBS run produces a single slice.
- Moreover, since observations required by MOAD are independent from each other, MOAD is inherently parallel.

To evaluate our claims, we have implemented MOAD and performed dependency analysis against a benchmark suite of programs that have been widely used in the slicing literature.

We evaluate the viability and the accuracy of MOAD by instantiating concrete slices based on the outcome of MOAD: program element A is in the backward slice of program element B iff the reported likelihood of B depending on A is greater than a threshold value. A comparison to a baseline random slicing technique shows that MOAD is indeed learning the program dependencies; a comparison to ORBS slices shows that MOAD can produce slices that are only 16% larger than ORBS slices, using less than 20% of observations. We also investigate various ways to construct the observation sets, as well as the impact of different inference models. Note that dependency modeling is more general than program slicing (and has other use cases [2]). However, we leave the additional use cases for future work and focus here on slicing as a simple way to present the approach.

The technical contributions of this paper are as follows:

- We introduce the concept of approximate dependency analysis, which transforms the dependency relationship from Boolean to probabilistic space.
- We present MOAD, a technique that models approximate program dependency, and describe the essential components: how to generate observations, and how to infer models of program dependency.
- We conduct an empirical evaluation of MOAD via program slices instantiated from the learned models. The results show that MOAD requires only 18.7% of the number of observations to construct the model; the size of the slice generated by the MOAD is 16% larger than the slice from ORBS, on average.

The rest of this paper is organised as follows. Section II introduces the concept of approximate dependency analysis, and explains how it relates to the existing slicing technique ORBS. Section III introduces MOAD, a technique that aims to model approximate dependency, and how we can use it for slicing by instantiating program slices from the learned dependency models. Section IV presents the set-up of empirical evaluation, the results of which are reported in Section V. Section VI contains discussions of our findings and potential future work. Section VII presents the related work, and Section VIII concludes.

II. Approximating Program Dependency

Program dependencies are dependence relations that hold between elements of a program, e.g. statements, expressions, or variables. If the computation of one element directly or indirectly affects the computation of another element we consider them to be dependent. A plethora of techniques have been proposed to capture and model dependence information.

Often these techniques are static and use parsing and detailed analysis of program elements based on the semantics of the programming language in question. The outcome of the analysis is typically exact and captures the binary dependence relations, i.e. two program elements either have or do not have a dependence relation. While dynamic dependence analysis approaches have been proposed, they typically annotate an already extracted static model with probabilities based on concrete executions [2] (static-then-annotate) or do not build any explicit model at all [11].

The downside of the static and static-then-annotate approaches is that they cannot handle heterogeneous systems, some of whose components are either binary, or written in multiple languages and file formats. Even if it is possible in theory to combine analyses of multiple languages and formats, concrete tools actually support only a fixed and typically small selection. An additional practical problem is that they need to duplicate the early stages of multiple compiler tool chains.

Observation-based slicing [11] (ORBS) addresses these problems and allows language-agnostic dynamic slicing without detailed semantic knowledge by reusing the build toolchain. An implementation of ORBS is also trivial and requires very few lines of code. However, ORBS does not learn a general model of the program components and their relations. Given a target slicing criterion, it simply creates a slice through iterative build and execute cycles. The process has to be repeated if another slicing criterion is selected for analysis which can typically happen in fault diagnosis or debugging applications.

This paper proposes a middle ground that leverages the minimal requirements, general applicability, and easy implementation of lightweight dynamic analysis techniques such as ORBS, while modeling dependence relations approximately. By not requiring any detailed syntactic or semantic knowledge of the components or programming languages involved, we can support heterogeneous systems with a very general approach and tool. Since model building is based on a few, specific, dynamic executions they can only *approximate* the full dependency information, typically with frequencies or probabilities. However, explicitly building such probabilistic models may have an advantage over the model-free, dynamic approaches such as ORBS, which do not learn anything between invocations. We posit that there exists an interesting and possibly complementary trade-off between what we propose and the existing program analysis methods.

While the idea of approximate program dependency modeling is a general one we focus below on a simple instantiation of it that targets slicing. This allows us to study the potential benefits compared to a well-known and general technique. Below we thus introduce MOAD in the context of program slicing: it dynamically mutates multiple program elements and models their effect on multiple other program elements.

III. MOAD: Modeling Observation-based Approximate Dependency

A. Methodology and Terms

This section first overviews MOAD and introduces key terminology, before presenting the two phases of our algorithm. The first, the observation phase, extracts a set of observations regarding the execution of various program mutants. The second, the inference phase, subsequently learns a model that aims to capture dependences within the program. As a case study, we illustrate the use of this model to infer program slices [16], [17].

Our approach is dynamic in nature and thus, in addition to a program, P, it takes a set of test inputs, I. We identify within P a set of deletable *units* $U = \{u_1, ..., u_{|U|}\}$. For example, U might be composed of lines of text, program statements, or blocks of code within a program. We subsequently create sub-programs of P, P', by deleting one or more units from P. To support the inference process, we abstract each sub-program as a boolean vector, called *deletion*, which has one entry for each unit. In this vector deleted units are assigned the value TRUE and retained units are assigned the value FALSE.

Borrowing the notion from program slicing [16], [17], we assess the impact of deleting various units at a set of (slicing) criteria, $C = \{c_1, ..., c_{|C|}\}$. Each c_i includes a program location (e.g., a line number) and a variable of interest (e.g., the variable updated by an assignment statement). To determine the impact of deleting a given unit, we observe the sequence of values produced by each criterion. The result is a boolean vector, called *response*, that has one entry per criterion. The entry c_i has the value TRUE iff the sequence of values produced for c_i is unaffected by the deletion (with respect to the sequence produced by the program for c_i without deletion). The key assumption here is that, if the deletion of unit u_a brings about a change in the the trajectory for criterion c_b, then the criterion c_b likely depends on (some part of) unit u_a.

During the observation phase we observe many such dependency relations. The output of the first phase is a set of observations, which pair a *deletion* with its corresponding *response*. Then, in the inference phase, we use the set of observations to build an inference model M that aims to capture dependence within the program. Finally, we study the inferred model by using it to produce program slices. In the next section we compare these inferred slices with those produced using Observation-based Slicing (ORBS) [11], [12].

B. Observation Phase

The core of the first phase is a *deletion generation scheme*, which generates the set of deletions used to produce program mutants. In our initial experiment, we suggest an exhaustive way to consider all cases of deleting the same number of different units. '1-hot' deletion generation scheme generates $|U|$ deletions where each deletion removes exactly one unit. In other words, it generates a set of one-hot encoding vectors of length $|U|$. In the same way, we design 'n-hot' deletion generation scheme which consists of one-hot to n-hot encoding vectors of length $|U|$; it considers all possible combinations of deleting different, less than n units.

In the rest of the paper, we consider two deletion generation schemes: 1-hot and 2-hot. The number of deletion generated by n-hot increases in a factorial order. With only 200 deletable units, 3-hot creates more than 1 million different deletions; observing them all incurs high cost. In the latter part of this paper, we show an attempt to solve this problem by sampling method. In addition, our future work includes developing a new efficient deletion generation scheme.

Algorithm 1: Observation phase

```
input :  $P$: an annotated version of the input program
         $I$: test suite
         GenScheme: deletion generation scheme
                    (1-hot, 2-hot)
         $max\_obsv$: maximum number of observations
output:  $O$: a set of observations
1  $O \leftarrow \{\}$
2  $E \leftarrow$ Observe $(P, I)$ // retain expected output
3  $deletions \leftarrow$ GenScheme $(P, max\_obsv)$
4  while $\neg deletions$.Empty () do
5      $deletion \leftarrow deletions$.Remove ()
6      $P' \leftarrow$ Apply $(P, deletion)$
7      $X \leftarrow$ Observe $(P', I)$
8      $response \leftarrow$ Compare $(E, X))$
9      $O \leftarrow O \cup \{(deletion, response)\}$
10 return $O$
```

Algorithm 1 describes the observation phase. Its inputs include, P, the program under study, I, the input test suite, the deletion generation scheme, GENSCHEME, and a maximum number of observations to generate. The algorithm assumes that P has been annotated to output a sequence of values capturing each slicing criterion, similar to ORBS [11]. The algorithm first initializes O to an empty set, E to the expected output sequence for each criteria (collected from the execution of unmodified P and I), and the set of *deletions* generated using the given scheme. If there are more than max_obsv observations possible under a given generation scheme, GENSCHEME randomly selects up to max_obsv of the possible deletions. Subsequently, Lines 4-9 process each deletion by first using the function APPLY to generate sub-program P' composed of only the non-deleted units (omitting those those units whose value in $deletion$ is TRUE). Next, function OBSERVE executes P' using set of inputs I to produce the trajectories for each criterion as produced by the annotations in P. The final step compares the expected output E with the output of P', and produce the result vector *response*. Note that P' may fail to compile in which case its outputs will fail to match the expected output. The response is paired with the deletion and recorded in the set of observation to be returned by the algorithm.

To illustrate Algorithm 1, consider a program P with three statements S_1, S_2, and S_3 using the 1-hot generation scheme. With three statements, 1-hot produces the three deletion {TRUE, FALSE, FALSE}, {FALSE, TRUE, FALSE}, and {FALSE, FALSE, TRUE}, unless max_obsv is less than three. Applying the first deletion to P produces the two statement program $S_2; S_3$, which is then observed (compiled and executed) using input I.

C. Inference phase

The inference phase, described in Algorithm 2, infers the slices for each criterion. The algorithm uses the set of observations, O, output by the first phase to train an inference model, $M : C \rightarrow D = \text{Boolean}^{|U|}$, where D is the set of possible

Algorithm 2: Inference phase

```
input :  $P$: a input program
         $C$: a set of slicing criteria
         $O$: a set of observations
         $dsg\_mdl$: a design of model $(\mathbb{O}, \mathbb{L}, \mathbb{B})$
output:  $\mathcal{P}\int$: set of inferred slices; one for each slicing
         criterion $c_k \in C$
1 $M \leftarrow dsg\_mdl.\textsc{Train}(O)$
2 $\mathcal{P}\int \leftarrow \{\}$
3 for $c_k \in C$ do
4     $deletion \leftarrow M(c_k)$
5     $P_k \leftarrow \textsc{Apply}(P, deletion)$
6     $\mathcal{P}\int \leftarrow \mathcal{P}\int \cup \{P_k\}$
7 return $\mathcal{P}\int$
```

deletions. Then, for each slicing criterion, the M is used to infer a deletion, which is then applied to the original program to produce the slice.

The key assumption made in the inference process is that if deleting the unit u_m changes the trajectory of the slicing criterion c_k, then u_m is likely to be in the slice of c_k. While this connection is straightforward for the 1-hot data, this data tends to overestimate deletability. For example when either of two statements can be deleted, but not both [11]. In contrast, for n-hot when n is greater than one, it is possible that not all of the deleted units influence c_k.

The remainder of this subsection presents the three different inference models that we study in the next section. As a notational convenience, hereafter we use "0" and "1" to denote "FALSE" and TRUE" respectively.

Once Success ($\mathbb{O}$): The Once Success model explicitly follows the aforementioned assumption. Let there exist a subprogram P' whose deletion unit u_m has been deleted. If P' preserves the trajectory of the slicing criterion c_k, the model removes u_m from the slice of c_k. More formally, the Once Success model, $M_{\mathbb{O}}$, trained with observations O, infers the deletion of the slice of c_k as follows:

$$M_{\mathbb{O}}(c_k)[m] = \begin{cases} 1, & \text{if } \exists (d,r) \in O \text{ s.t. } d[m] = 1 \text{ and } r[k] = 1 \\ 0, & \text{otherwise} \end{cases}$$

where $d[m]$ represents the m-th element of deletion vector s and $r[k]$ represents the k-th element of response vector r. Thus, $d[m] = 1$ and $r[k] = 1$ represents that unit m has been deleted and the response for criterion k is unchanged.

Logistic ($\mathbb{L}$): The Logistic model regards the response element, $r[k]$ (for slicing criterion c_k) as a dependent variable and the elements of the deletion, s, as the independent variables. Because the variables are binary values, they are modeled using logistic regression,

$$r[k] \approx L(d, \beta_k),$$

where the elements of β_k are the coefficients of the regression. The sign of each coefficient is used to determine if the corresponding unit is removed to preserve the slicing criterion c_k. If $\beta_k[m]$, the m-th coefficient of β_k, has a positive value, u_m is more likely to be removed from the slice, while a negative value indicates that u_m is less likely to be removed from the slice. More formally, $M_{\mathbb{L}}$, the Logistic model, infers the deletion vector for the slice taken with respect to c_k as follows:

$$M_{\mathbb{L}}(c_k)[m] = \begin{cases} 0, & \text{if } \beta_k[m] \leq 0 \\ 1, & \text{if } \beta_k[m] > 0 \end{cases} .$$

Bayesian ($\mathbb{B}$): The final model we consider uses Bayesian inference. We assume that $P(c_k|u_m)$ denotes the conditional probability of preserving the trajectory of c_k when the unit u_m has been deleted. From the observations, O, we estimate $\hat{P}(c_k|u_m)$ as follows:

$$\begin{aligned} P(c_k|u_m) &= P(\text{preserves trajectory of } c_k | u_m \text{ has been deleted}) \\ &= P(r[k] = 1 | d[m] = 1) \\ &= \frac{P(r[k] = 1, d[m] = 1)}{P(d[m] = 1)} \\ \hat{P}(c_k|u_m) &= \frac{\#(r[k] = 1 \text{ and } d[m] = 1)/|O|}{\#(d[m] = 1)/|O|} \\ &= \frac{\#(r[k] = 1 \text{ and } d[m] = 1)}{\#(d[m] = 1)}, \end{aligned}$$

where $\#(\mathsf{cond})$ is a number of observations in O satisfying the condition cond. Formally, $M_{\mathbb{B}}$, the Bayesian model, infers the deletion of the slice of c_k as follows:

$$M_{\mathbb{B}}(c_k)[m] = \begin{cases} 0, & \text{if } \hat{P}(c_k|u_m) \leq \mu_{i \in \{1..|U|\}}(\hat{P}(c_k|u_i)) \\ 1, & \text{if } \hat{P}(c_k|u_m) > \mu_{i \in \{1..|U|\}}(\hat{P}(c_k|u_i)), \end{cases}$$

where $\mu_{i \in \{1..|U|\}}(\hat{P}(c_k|u_i))$ is an average value of the estimated probability.

IV. Experiment Setup

A. Research Questions

We evaluate MOAD by investigating the following four research questions.

RQ1. Viability: *Do the learned models capture program dependence information?*

To ascertain if our approach is viable we compare the learned models' ability to produce slices against that of a random slicer. If none of the models can outperform a random slicer then there is no reason to consider them further.

RQ2. Impact of the inference model: *Assuming that more than one model is viable, how does the performance of the viable models compare?*

To study RQ2 we consider the ability of each model to compute program slices. Because the models are trained with runtime information, they approximate dynamic slices. The most closely related dynamic slicing approach is observational slicing. As benchmarks we consider the slices produced by two observational slicing implementations W-ORBS [11] and T-ORBS [13], [14]. While these two are expected to produce more accurate slices than MOAD, they are also expected to take longer to do so.

When considering RQ2, *performance* is compared in terms of both slice precision, measured in lines of code, and slicing effort, measured as the number of observations required.

RQ3. Performance compared to ORBS: *For the best inference model, how well does MOAD perform compare to ORBS?*

Based on the results from RQ2, we compare the performance of ORBS and MOAD when using the best of the viable models. Parallel to RQ2, effort is measured in terms of the number of observations required. However, for precision, we take a more refined approach and count both missing and excess lines relative to the W-ORBS slice.

RQ4. Sampling effect: *How feasible is using only a sample of the* 2*-hot data?*

The primary goal of RQ4 is to determine if there is a sweet spot in the analysis that best balances precision and effort. Our expectation here is that a model trained with more observations will have higher precision, but require greater effort. To investigate RQ4, we consider ten different sample sizes when sampling from each program's set of 2-hot observations. The sizes include 10%, 20%, ..., 100% of the 2-hot observations. Since the performance might be biased by a particular sample, we repeat the sampling ten times.

B. Subjects

Subject	SLoC	$\lvert U\rvert$	$\lvert C\rvert$
mbe	64	45	16
mug	61	44	13
wc	46	33	17
prttok	410	388	98
prttok2	387	364	75
replace	508	465	253
sched	283	252	75
sched2	276	248	81
totinfo	314	227	210
tcas	152	110	62

TABLE I: The statistics of experiment subjects' properties.

Table I shows the program we study. For each it includes the number of non-comment-non-blank lines (SLoC), the number of units, and the number of criteria used. The first three subjects, mbe, mug, wc, are small, well known, programs that have well studied semantics. This makes them amenable to careful precise study. Furthermore, the first two raise specific challenges to dependence analysis and thus serve to highlight the pros and cons of our approximation technique. In addition, we study the Siemens suite [18], to see how our technique works on ordinary C code. The Siemens suite is used in lieu of larger programs because it is possible to exhaustively compute all the slices of each program (for all scalar slicing criteria). This removes any slice selection bias from the data.

C. Observation-based Slicing (ORBS)

We use Observation-Based Slicing (ORBS) [11] as a benchmark approach to evaluate the performance of MOAD. ORBS is a dynamic program slicing technique based on direct observation of program semantics (when executing the program on a chosen test suite). An ORBS slicer performs iterative, speculative deletion of parts of the code. Each deletion is made permanent if it preserves the trajectory of values computed at the slicing criterion.

The original ORBS implementation [11], slices source code at the line-of-text level. We refer to this algorithm as W-ORBS where the 'W' captures the use of a *deletion window*, in which W-ORBS considers the deletion of a sequence of consecutive lines of text. In addition to a performance advantages, the use of a deletion window enables W-ORBS to delete lines that can only be deleted together (e.g., the pair of brackets that enclose an empty block). Applied to line l_i, W-ORBS attempts to delete from one to k lines (i.e., from $\{l_i\}$ to $\{l_i, \ldots, l_{i+k-1}\}$). If it successfully deletes j lines (i.e., $\{l_i, \ldots, l_{i+j-1}\}$), the deletion continues with line l_{i+j}; if all k attempts fail, the deletion continues with line l_{i+1}. Thus after each successful deletion, W-ORBS moves onto the next target source code line (skipping over the deleted lines), while after each unsuccessful deletion it reverts the deletion before moving on to the next line of the file. W-ORBS performs multiple passes over the code until it cannot delete anything further, producing a 1-minimal line slice [11].

A recent variation of W-ORBS, T-ORBS [13], [14] works with a tree-based representation. T-ORBS performs a breadth-first tree traversal using a work list. For each node, n, it attempts to delete the subtree rooted at n. If the resulting program produces the same trajectory for the slicing criterion then the subtree is permanently deleted. Otherwise, its children are appended on the work list for later consideration. The T-ORBS implementation we used employs SrcML [19] to produce an XML tree from a program. It is important to note that we modified the original T-ORBS algorithm to attempt to delete only those syntactic elements that are considered by MOAD; in this case statements. Our motivation for modifying T-ORBS to attempt the deletion of only those syntactic elements that correspond to the units considered by MOAD is to provide an apples-to-apples comparison. Otherwise T-ORBS takes considerably longer as it has to consider numerous subtrees that represent only a subset of the statement. Doing so requires introducing a small amount of language information into an otherwise language agnostic algorithm but provides a better basis for comparison (and as a consequence also brings a dramatic speedup).

D. Configuration

In the initial experiments, the units considered by MOAD are programming language statements. We use SrcML (ver. 0.9.5) [19], an XML-based multi-language parsing tool, to identify statements. SrcML enables our approach to be applied to any programming language, including multi-lingual programs, that SrcML can process. Other than this dependence, our algorithm is language independent.

The set of slicing criteria considered consist of all scalar (`char`, `int`, `float`, etc.) assignments. We use Clang (ver.

3.8) [20] to insert logging statements for each slicing criterion. These statements are responsible for outputting the sequence of values computed for the slicing criteria. A slicer's goal is to preserve this sequence while removing unnecessary code.

As a benchmark, we apply both W-ORBS (with maximum window size of three) and (the modified) T-ORBS to our subjects.

The experiments were performed under Ubuntu 16.04, on an Intel(R) Core(TM) i7 CPU with 32GB of memory using gcc version 5.5.0.

E. Metrics

There are various metrics used to measure and to compare the performance of the slicing techniques.

- $WS_k, TS_k,$ and MS_k: The *slice* of the criterion c_k generated by W-ORBS, T-ORBS, and MOAD, respectively.
- miss: Given a reference slice (e.g., WS_k) and an inferred slice (e.g., MS_k) the number of units in the reference slice *missing* from the inferred slice.
- excess: Given a reference slice (e.g., WS_k) and an inferred slice (e.g., MS_k) the number of units in the inferred slice not in the reference slice.

By design T-ORBS and MOAD share the same set of deletable units. Thus, we can calculate miss and excess directly applying a set difference between the set of units remaining on the slice. Since W-ORBS modifies the source code at the line-of-text level, the same method is not applicable to compare W-ORBS and MOAD. Instead, we use a python difflib module to calculate miss and excess at the line level.

V. Results

A. Viability

To answer **RQ1**, we first create a random slicer. Our implementation randomly deletes each unit with a probability of 0.5. For every slicing criterion in every subject program, we run the random slicer ten times and check whether the slice generated preserves the trajectory of the slicing criterion. With 900 slicing criteria (see Table I) spread across the ten subject programs, the random slicer generates 9,000 slices in total. Only fifteen of the random slices compile, and none of them preserve the trajectory of the given slicing criterion. This result clearly indicates that it is very unlikely to produce a program slice by chance.

In contrast, Table II shows the ability of MOAD to produce viable slices that not only compile, but also capture the desired semantics. In the table, the second column shows the deletion generation scheme used to generate the observations. Then in the remaining columns we report, for each scheme, the success rate when using each of the three inference models: $\mathbb{O}$, $\mathbb{L}$, and $\mathbb{B}$. For the smaller programs mbe, mug, and wc, most slices preserve the trajectory successfully. For the Siemens suite 42% of the generated slices preserve the trajectory.

In the table, prttok shows a particularly low success rate. Investigating this, we found that the root cause was two lines of code, shown in the code snippet below, that can not both be deleted. It is interesting that deleting either Line 188 or 189 individually does not affect the trajectory. Thus the model learns to remove these lines. Consequently, when MOAD infers a slice it tends to unwantedly omit both lines. The result is that most trajectories change. This suggests the use of stronger statistical models (e.g., Rasmussen's Gaussian processes [21]), that can capture higher-level interaction effects between program elements.

Subject	Deletion Gen. Scheme	Success Rate $\mathbb{O}$	$\mathbb{L}$	$\mathbb{B}$
mbe	1-hot	1.00	1.00	1.00
	2-hot	1.00	1.00	1.00
mug	1-hot	1.00	1.00	1.00
	2-hot	1.00	1.00	1.00
wc	1-hot	1.00	1.00	1.00
	2-hot	0.88	0.76	1.00
prttok	1-hot	0.03	0.04	0.11
	2-hot	0.03	0.03	0.11
prttok2	1-hot	0.72	0.19	0.77
	2-hot	0.63	0.13	0.67
replace	1-hot	0.07	0.31	0.28
	2-hot	0.03	0.13	0.31
sched	1-hot	0.48	0.47	0.41
	2-hot	0.39	0.35	0.43
sched2	1-hot	0.30	0.26	0.28
	2-hot	0.17	0.26	0.28
tcas	1-hot	0.48	0.90	0.48
	2-hot	0.26	0.68	0.48

TABLE II: MOAD's success rate on the ten test subjects

```
164 static token numeric_case(...)
165 {
        ...
188     strcpy(token_ptr->token_string, token_str);
189     return (token_ptr);
190 }
```

Based on these results, we answer **RQ1** as follows:

RQ1. Viability: Inference models trained using dynamic observations can successfully learn program dependences.

B. Model Impact

Our initial look at the impact that a given model has focuses on the model's ability to remove units. Table III shows the average slice size, $\mu\left(WS_k\right)$, $\mu\left(TS_k\right)$, and $\mu\left(MS_k\right)$, over all slicing criteria for W-ORBS, T-ORBS, and the three models used with MOAD. To facilitate inter-program comparison, the average slice sizes are given as a percentage of the original program's size. The table also shows the number of observations involved. For W-ORBS and T-ORBS this count reflects the number of compilations and executions made while computing each slice, while for MOAD the number is the number of compilations and executions used in constructing the training data.

To gain some intuition for the relative slice sizes, we first compare MOAD's average slice size with that of W-ORBS and then T-ORBS before focusing in on the impact of the individual models. Due to the approximate nature of the inference, MOAD is expected to generate larger slices than W-ORBS or T-ORBS. To normalize the data across programs, we first consider the ratio of the average slice size generated using one of the six MOAD variants (2 deletion

Subject	W-ORBS		T-ORBS		1-hot						2-hot					
	$\lvert O_{\mathrm{W}}\rvert$	$\mu\left(WS_k\right)$	$\lvert O_{\mathrm{T}}\rvert$	$\mu\left(TS_k\right)$	$\lvert O_{\mathrm{M}}\rvert$	$\frac{\lvert O_{\mathrm{M}}\rvert}{\lvert O_{\mathrm{W}}\rvert}$	$\frac{\lvert O_{\mathrm{M}}\rvert}{\lvert O_{\mathrm{T}}\rvert}$	$\mu\left(MS_k\right)$ $\mathbb{O}$	$\mu\left(MS_k\right)$ $\mathbb{L}$	$\mu\left(MS_k\right)$ $\mathbb{B}$	$\lvert O_{\mathrm{M}}\rvert$	$\frac{\lvert O_{\mathrm{M}}\rvert}{\lvert O_{\mathrm{W}}\rvert}$	$\frac{\lvert O_{\mathrm{M}}\rvert}{\lvert O_{\mathrm{T}}\rvert}$	$\mu\left(MS_k\right)$ $\mathbb{O}$	$\mu\left(MS_k\right)$ $\mathbb{L}$	$\mu\left(MS_k\right)$ $\mathbb{B}$
mbe	6,546	35.1%	1,163	40.6%	47	0.72%	4.04%	46.0%	48.2%	**45.8%**	948	14.5%	81.5%	**43.4%**	48.4%	46.2%
mug	4,420	25.6%	969	33.7%	46	1.04%	4.75%	**41.7%**	51.0%	43.1%	904	20.5%	93.3%	**35.8%**	48.3%	45.5%
wc	3,843	23.7%	860	30.3%	35	0.91%	4.07%	**37.3%**	54.6%	46.9%	454	11.8%	52.8%	**31.6%**	44.9%	41.7%
prttok	262,374	43.3%	68,372	43.2%	390	0.15%	0.57%	**51.8%**	58.3%	59.9%	74,175	28.3%	108.5%	43.6%	**43.1%**	58.6%
prttok2	246,668	37.3%	41,324	38.8%	366	0.15%	0.89%	**42.6%**	66.6%	49.6%	65,393	26.5%	158.2%	**37.6%**	56.2%	48.8%
replace	1,214,230	45.6%	311,323	45.9%	467	0.04%	0.15%	**54.9%**	68.3%	60.9%	106,412	8.8%	34.2%	**44.1%**	50.9%	61.0%
sched	130,199	37.3%	33,911	37.9%	254	0.20%	0.75%	**59.5%**	69.1%	61.5%	31,181	23.9%	91.9%	**45.3%**	58.4%	62.9%
sched2	93,173	35.3%	31,200	33.7%	250	0.27%	0.80%	**49.2%**	64.6%	56.8%	30,227	32.4%	96.9%	**38.4%**	49.5%	56.3%
totinfo	451,715	33.9%	86,499	39.7%	229	0.05%	0.26%	**50.8%**	55.8%	53.5%	25,151	5.6%	29.1%	43.2%	**31.5%**	53.3%
tcas	40,883	37.3%	11,377	33.3%	112	0.27%	0.98%	**46.2%**	70.5%	47.5%	5,846	14.3%	51.4%	**39.4%**	53.1%	47.7%
Average:		35.4%		37.7%		0.37%	1.73%	48.0%	60.7%	52.5%		18.7%	79.8%	40.2%	48.4%	52.2%

TABLE III: $|O_{\mathrm{W}}|$, $|O_{\mathrm{T}}|$, and $|O_{\mathrm{M}}|$ represent the number of involved observations for each W-ORBS, T-ORBS, and MOAD, respectively. $\mu\left(WS_k\right)$, $\mu\left(TS_k\right)$, and $\mu\left(MS_k\right)$ represent the mean slice size, given as a percentage of the original program's size generated by each W-ORBS, T-ORBS and MOAD, respectively. Columns 9-11 and 15-17 show $\mu(MS_k)$ separately for each of the three models where the smallest mean for the three models is shown in bold.

Subject	Deletion Generation Scheme	MS_k vs WS_k (line)						MS_k vs TS_k (line)						MS_k vs TS_k (stmt)					
		$\mathbb{O}$		$\mathbb{L}$		$\mathbb{B}$		$\mathbb{O}$		$\mathbb{L}$		$\mathbb{B}$		$\mathbb{O}$		$\mathbb{L}$		$\mathbb{B}$	
		E(%)	M(%)	E(%)	M(%)	E(%)	M(%)	E(%)	M(%)	E(%)	M(%)	E(%)	M(%)	E(%)	M(%)	E(%)	M(%)	E(%)	M(%)
mbe	1-hot	2(3)	11(18)	2(3)	13(21)	2(3)	11(18)	3(5)	9(15)	3(6)	11(18)	3(6)	9(15)	1(3)	5(13)	1(3)	7(16)	1(3)	6(14)
	2-hot	2(3)	9(15)	2(3)	13(21)	2(3)	11(19)	2(4)	6(10)	3(5)	11(18)	3(6)	9(15)	1(3)	3(6)	1(3)	6(14)	1(3)	6(14)
mug	1-hot	8(14)	13(22)	7(12)	19(32)	8(14)	14(24)	0(1)	6(11)	2(4)	15(25)	0(0)	7(12)	0(0)	7(17)	0(0)	15(34)	0(0)	8(18)
	2-hot	8(14)	10(16)	8(13)	18(30)	8(13)	16(27)	0(1)	3(5)	0(1)	12(19)	0(1)	9(15)	0(1)	1(4)	0(0)	10(24)	0(0)	8(19)
wc	1-hot	5(11)	8(18)	4(9)	16(36)	4(10)	13(28)	1(2)	6(13)	2(4)	15(34)	1(3)	11(24)	0(1)	4(14)	0(1)	11(34)	0(1)	6(21)
	2-hot	5(12)	6(13)	5(12)	13(28)	4(10)	10(23)	1(3)	3(8)	2(6)	11(25)	1(3)	9(20)	1(4)	2(7)	1(4)	8(25)	0(2)	5(17)
prttok	1-hot	54(13)	63(15)	42(10)	81(19)	50(12)	92(22)	9(2)	45(11)	17(4)	81(19)	6(1)	75(18)	6(1)	42(10)	5(1)	71(18)	3(0)	72(18)
	2-hot	58(14)	34(8)	69(16)	46(11)	50(12)	87(21)	15(3)	18(4)	39(9)	42(10)	6(1)	70(17)	11(2)	15(3)	30(7)	34(9)	4(1)	66(17)
prttok2	1-hot	34(8)	39(10)	27(7)	128(33)	30(7)	61(15)	12(3)	28(7)	14(3)	125(32)	15(3)	57(14)	7(2)	33(9)	4(1)	95(26)	5(1)	50(13)
	2-hot	37(9)	23(6)	33(8)	94(24)	30(7)	58(15)	13(3)	10(2)	16(4)	87(22)	15(3)	54(14)	9(2)	7(2)	10(2)	59(16)	5(1)	48(13)
replace	1-hot	113(22)	110(21)	101(20)	171(33)	105(20)	132(25)	29(5)	81(16)	30(6)	156(30)	40(8)	122(24)	15(3)	63(13)	13(2)	111(23)	13(2)	91(19)
	2-hot	124(24)	66(13)	112(22)	94(18)	106(20)	133(26)	34(6)	32(6)	48(9)	86(16)	38(7)	121(23)	25(5)	17(3)	28(6)	52(11)	13(2)	91(19)
sched	1-hot	33(11)	74(26)	32(11)	102(36)	32(11)	80(28)	6(2)	65(23)	7(2)	96(34)	7(2)	72(25)	2(1)	67(26)	2(0)	83(33)	2(1)	72(28)
	2-hot	35(12)	37(13)	30(10)	72(25)	33(11)	85(30)	7(2)	26(9)	9(3)	69(24)	7(2)	77(27)	3(1)	27(10)	2(1)	60(23)	2(1)	76(30)
sched2	1-hot	41(15)	61(22)	39(14)	101(36)	41(15)	81(29)	5(1)	48(17)	7(2)	93(33)	6(2)	70(25)	2(1)	45(18)	2(1)	82(33)	2(1)	62(25)
	2-hot	44(16)	32(11)	43(15)	64(23)	41(15)	80(29)	7(2)	19(7)	9(3)	55(20)	6(2)	69(25)	4(1)	20(8)	5(2)	43(17)	2(1)	61(24)
totinfo	1-hot	37(11)	73(23)	31(10)	86(27)	35(11)	78(25)	3(1)	42(13)	7(2)	65(20)	5(1)	51(16)	1(0)	34(15)	1(0)	49(21)	1(0)	42(18)
	2-hot	37(12)	50(16)	76(24)	51(16)	35(11)	78(25)	4(1)	19(6)	53(17)	30(9)	4(1)	50(16)	2(1)	11(5)	48(21)	23(10)	1(0)	41(18)
tcas	1-hot	27(20)	35(26)	21(15)	68(50)	27(20)	37(27)	4(3)	29(21)	3(2)	67(49)	4(3)	31(23)	2(2)	23(21)	1(0)	48(43)	2(2)	24(22)
	2-hot	29(21)	25(19)	20(15)	41(30)	26(19)	37(27)	6(4)	19(14)	2(1)	39(29)	4(3)	32(23)	3(3)	15(13)	1(0)	25(23)	2(2)	24(22)

TABLE IV: Average value of excess, denoted by 'E', and miss, denoted 'M', in MS_k when compared to WS_k and TS_k. Columns 3-8 compare with WS_k at the line level, Columns 9-14 compare with TS_k at the line level, and finally Columns 15-20 compare with TS_k at the statement (unit) level. The values in the parentheses are the percentage of excess and miss compared to the number of lines or units in the original program.

generation scheme $\times$ 3 inference model) to the average W-ORBS slice size. The results find that MOAD produces slices that are on average 45% larger than those produced using W-ORBS. The most significant difference occurs with the program wc using the Logistic inference model trained with the 1-hot data. The size of the inferred slice is 2.3 times larger than that of the corresponding W-ORBS slice. On the other hand, there are inferred slices that are smaller than the corresponding W-ORBS slice. For example, the models trained by the configuration of (replace, Once Success, 2-hot), (prttok, Logistic, 2-hot), and (totinfo, Logistic, 2-hot) inferred slices that have 96%, 99%, and 93% of the W-ORBS slice size, respectively.

Next when compared with T-ORBS, the differences are smaller, which reflects T-ORBS producing slightly larger slices than W-ORBS on average. Over all programs MOAD produces slices that are 35.3% larger than T-ORBS. The most significant difference, occurs for (tcas, Logistic, 1-hot) where the MOAD slice is 2.1 times larger. Because the T-ORBS slices are slightly larger than the W-ORBS slices there are again examples where the MOAD slice is smaller than the corresponding T-ORBS slice. We investigate the differences between MOAD slices and both T-ORBS and W-ORBS slices further with RQ3.

Turning to the question of how performance of the three models compares, overall Once Success($\mathbb{O}$) generates smaller slices than the other two inference models (ANOVA, $p-value < 0.0001$). Comparing the rows of Table III, Once Success produces the smaller slice in 17 of the 20 rows. Considering the overall impact of the training data, 2-hot produces the smaller average slice in 24 of the 30 cases.

Because of its dominance, we focus the remainder of the analysis on the 2-hot data. First, considering the Once Success model trained using the 2-hot observations, the size of the average inferred slice is 16% larger the the W-ORBS average and 7% larger than the T-ORBS average. This is due to how the inference algorithm Once Success works. No matter how rarely it happens, if the deletion of a unit does not affect the trajectory, Once Success learns to delete it when inferring a slice. This implies that the number of deletions monotonically increases as the set of observations is increased. Thus, Once Success trained with 2-hot data ubiquitously generates smaller slices than when trained with the 1-hot data.

Considering the Logistic model ($\mathbb{L}$), it also tends to generate smaller slices using the 2-hot data, doing so for nine of the ten subjects. This dominance illustrates that the model learns more dependency relations from larger observations. Finally, the sizes of the slices generated by the Bayesian model ($\mathbb{B}$) show relatively minor variation as the size of the set of observations increases. That this is the most sophisticated of the models is thus evident. For example, the 1-hot data and 2-hot data each produce the smaller average for five of the subjects. Thus with more data, the estimated probability of preserving the slicing criterion may increase or decrease, depending on the observations. This results is echoed in the study of RQ4 in Section V-D.

Based on the data, we answer **RQ2** as follows:

RQ2. Impact of the inference model: Among three inference models, Once Success generates the smallest slices.

C. Comparison on the difference of the slice

Table IV compares MOAD slices to those produced by the two implementation of ORBS. For each subject, deletion generation scheme, and inference model, we calculate excess and miss (defined in Section IV-E), to facilitate a more sophisticated analysis of differences between model performance. The values in the table are the average counts computed over each program's slicing criteria; the values in parentheses are the corresponding percent of the number of lines or units in the original program. For both excess and miss, the smaller the number is, the more similar two slices are. Finally, note that it is only possible to compare with W-ORBS slices at the line level. In contrast, T-ORBS slices can be compared with MOAD slices at both the line level and the unit level.

Patterns evident in the data include that, when compared to W-ORBS, MOAD requires significantly fewer observations. For example, the number of 1-hot observations is orders of magnitude smaller than the numbers used by W-ORBS. Similarly, the 2-hot deletion generation scheme creates only 18.7% as many observations as used by W-ORBS. T-ORBS tends to use fewer observations when compared to W-ORBS. Thus, for example, The size of 1-hot data is 1.7% of the size of T-ORBS observations, while the number of 2-hot observations is 79.8% of the number of observations used by T-ORBS.

The patterns for 1-hot and 2-hot extend into the individual models. As the number of observations increases from 1-hot to 2-hot, the number of miss for Once Success ($\mathbb{O}$) significantly decreases, while excess slight increases. This tendency is repeated for the Logistic models ($\mathbb{L}$). However, miss barely changes for Bayesian models ($\mathbb{B}$) which further explains why the size of $\mathbb{B}$ slices shows little change in Table III.

Finally, turning to the three models, for all subjects and deletion generation schemes, $\mathbb{O}$ yields the smallest values of miss. For example, it accurately deletes 11 to 30 more lines when compare to the other two models. While there are many cases where excess of $\mathbb{O}$ is larger than excess of other models, the difference is modest. Thus, among the inference models, $\mathbb{O}$ tends to generate slice closest to those produced by W-ORBS and T-ORBS. Table V takes this into account as it shows the average difference of miss and excess between $\mathbb{O}$ and the other two models. Values are computed over all slicing criteria and subjects. It is clear from this data that for miss, $\mathbb{O}$ is consistently and notable better than the other two. Interestingly this dominance does not extend to excess where the values are smaller and more varied.

miss	MS_k vs WS_k (line) $\mathbb{L}-\mathbb{O}$	MS_k vs WS_k (line) $\mathbb{B}-\mathbb{O}$	MS_k vs TS_k (line) $\mathbb{L}-\mathbb{O}$	MS_k vs TS_k (line) $\mathbb{B}-\mathbb{O}$	MS_k vs TS_k (stmt) $\mathbb{L}-\mathbb{O}$	MS_k vs TS_k (stmt) $\mathbb{B}-\mathbb{O}$
1-hot	29.8	11.2	36.5	14.6	24.9	11.0
2-hot	21.4	30.3	35.2	23.6	25.5	18.2

excess	MS_k vs WS_k (line) $\mathbb{L}-\mathbb{O}$	MS_k vs WS_k (line) $\mathbb{B}-\mathbb{O}$	MS_k vs TS_k (line) $\mathbb{L}-\mathbb{O}$	MS_k vs TS_k (line) $\mathbb{B}-\mathbb{O}$	MS_k vs TS_k (stmt) $\mathbb{L}-\mathbb{O}$	MS_k vs TS_k (stmt) $\mathbb{B}-\mathbb{O}$
1-hot	-4.8	-2.0	2.0	1.5	-0.7	-0.7
2-hot	1.9	-4.4	6.4	1.7	2.8	-1.3

TABLE V: The difference in the average value of miss and excess between the three inference models. The upper table shows the data for miss while the lower table shows that for excess.

To gain confidence, we applied an ANOVA and then Tukey's HSD test [22] to the values of miss and excess[1]. The statistically significant results ($p < 0.0001$) show that miss for $\mathbb{O}$ is smaller than it is for $\mathbb{L}$ and $\mathbb{B}$. However, there is no significant differences between $\mathbb{L}$ and $\mathbb{B}$. The results for excess find $\mathbb{B}$ the smallest, followed by $\mathbb{O}$, and then $\mathbb{L}$.

Overall, the result shows that Once Success model trained with 2-hot observations can generate slices that are not only compact but also similar to ORBS slices. There exist some cases for which the Logistic model produces the smallest slice, such as the case of slicing totinfo with 2-hot data. However, such slices tend to have a high excess. Since the ORBS slice is not a unique slice of the slicing criteria, we explicitly check the output trajectory of such small slices. Our further investigation shows that they are unable to preserve the trajectory of the targeting slicing criteria, which implies that the dependency inference was not sufficiently precise.

From the overall trends observed in size and similarity comparisons, we answer our **RQ3** as follows:

RQ3. Performance compared to ORBS: Using Once Success trained with the 2-hot data, MOAD requires less than one fifth of the observations compare to W-ORBS. At the same time the inferred slice is only 16% larger than the W-ORBS slice.

D. Sampling effect

RQ4 considers the tradeoff between the amount of training data used and the quality of the inference. To gain an initial impression for the data, Figure 1 shows two examples, taken from the programs replace and totinfo. In each of the six plots, the x-axis shows the size of the sample, while the y-axis shows the ratio of the average slice size ($|MS_k|$) to the original program size ($|P|$): the values are averaged over all slicing criteria. Finally, the solid red (grey) and dashed blue horizontal

[1]The full details of Tukey's HSD results are available online at: https://hidden.for.dbr

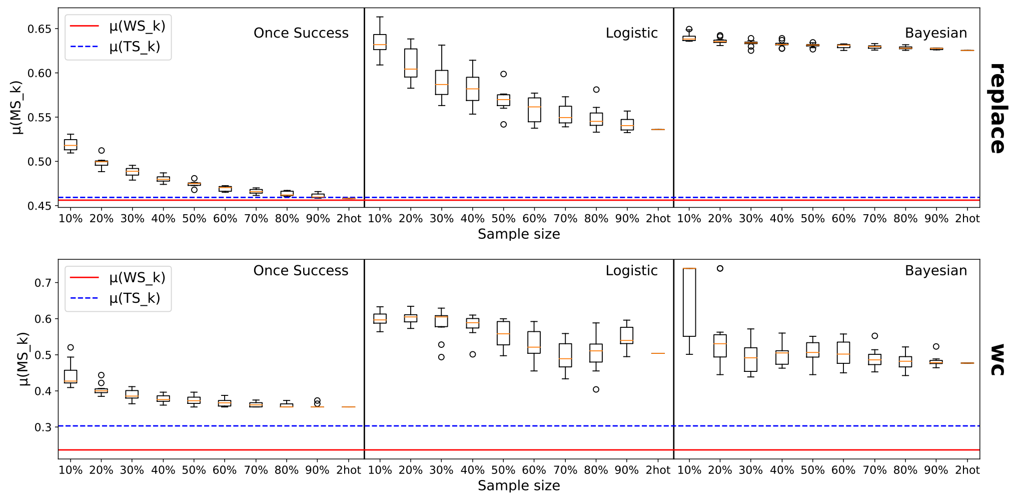

Fig. 1: The figure presents $\mu\left(MS_k\right)$ which represent the mean slice size, given as a percentage of the original program's size generated by MOAD using each size of the sample from 2-hot data. The boxplot shows the results of a trained model from 10 different random sampling. The red and blue line represents the ratio of the W-ORBS and T-ORBS slice size to the original program size averaging by all slicing criteria of all subjects.

lines represent the average slice size ratio produced by W-ORBS and T-ORBS, respectively. These six plots clearly show that as the sample size increases the size of slices approach the ORBS slice as the sample size increases.

Because the plots indicate a fair amount of variation, we consider each model separately. To begin with, the left two plots for Once Success model $\mathbb{O}$ suggest that $\mathbb{O}$ performs better when using 100% of the data. The reduction in size is substantial at small sample sizes, and it continues to decrease as the sample size increases. The average of the first three slice size differences ($20\% - 10\%$, $30\% - 20\%$, and $40\% - 30\%$) is 4.4 times larger than the average of the last three slice size differences ($70\% - 80\%$, $80\% - 90\%$, and $90\% - 100\%$). Note also that, when using only half of the 2-hot observations, $\mathbb{O}$ generates slices that are only 3.4% larger than when using all the data. Finally, the variance among different sampling is relatively small, which suggests that $\mathbb{O}$ is robust against the stochastic sampling.

The size of the slices also tends to decrease as the sample size increases for the Logistic model, $\mathbb{L}$, but the trend is not as strong as that of $\mathbb{O}$. The $\mathbb{L}$ model also shows higher variances across samplings, when compared to other inference models. Similarly, while the Bayesian model, $\mathbb{B}$, tends to generate smaller slices with more observations, the median size fluctuates and the difference of the slice size between samples is relatively small.

To gain additional statistical confidence, we have applied an ANOVA separately to each model[2]. In all three cases, the results are statistically significant ($p < 0.0001$). Applying Tukey's post-hoc test finds five equivalence classes of the resulting mean slice sizes. The most useful findings are that using samples of 40% to 90% of the 2-hot data can produce mean slice sizes that are not statistical different. The same is true of the range 50% to 100%, suggesting that using only half the data produces results no different than using all of the data. For $\mathbb{L}$ models, there is a similar band from 30% to 80%, and two narrower bands from 60% to 90% and 80% to 100%. These bands being narrows reflects these models being more sensitive to the specific data that they are trained on.

Finally, the Bayesian model, $\mathbb{B}$, shows the greatest stability with all values from 20% to 100% being in the same band, with only 10% showing inferior performance. This suggests that the models themselves are very robust against sampling variances. If it were possible to improve the size and the accuracy of slices produced by $\mathbb{B}$ models, the stability observed here may be a strong benefit of using $\mathbb{B}$ models.

We answer **RQ4** as follows.

RQ4. Sampling effect: The high rate of reduction and high variability of the inferred slice size for small sample sizes indicates there is some sampling effect especially with Once Success. At the other end the wide bands shows that is possible to train high performing models using only a fraction of the training data. For example Once Success infers slices an insignificant 3.4% larger when using only half of the observations while the Bayesian models has similar performance using as little as 20% of the training data.

VI. Discussions and Future Work

A. *Once Success ($\mathbb{O}$) vs.* Critical Slicing *vs. ORBS*

The Once Success model ($\mathbb{O}$) generates slices by deleting a line if the sub-program without the line preserves the trajectory. This is conceptually identical to *critical slicing* [23] and the slice of $<$ 1-hot, $\mathbb{O}$ $>$ is exactly the same as a *critical slice*. The result of Section V-C shows that the slices from *critical slicing* and ORBS are similar to each other for the programs studied in this paper. The next step is to investigate whether this tendency is retained on larger programs with more complicated dependency structures. Similarly, the slice from

[2]The full details of ANOVA results are available online at: https://hidden.for.dbr

ORBS with statement-level granularity might give a similar slice to the slice from $\mathbb{O}$.

B. Advanced and adaptive deletion generation scheme

The initial study reported here generates the deletions to observe before the observation starts. A more principled way would be to use Design of Experiments [24] to systematically select among the many possible deletions. This could also allow and utilise the sampling of more than 2 deletions per build and observe cycle, which could provide both speedup and faster learning. Yet another way that this could be approached is to generate the deletion just-in-time (adaptively) with respect to the observations that have been made, e.g. via active learning [25]. Using the observation data, an interim model can be learned that can propose the next deletion with the highest information gain.

C. Alternative inference models

Richer inference models could be used to model the relationships between units, and between criteria. Models such as Bayesian Networks [26] might be used to encode the conditional independencies between parts of programs and to infer the likelihood of program unit inclusion in slices based on the distributed joint distribution encoded therein. Markov Random Fields might also be tried to model the strength of association between program units as observed. Even if such probabilistic graphical models can be expected to perform well since they can capture the often graphlike dependency relations of software, more general statistical inference models, e.g. Gaussian processes [21], should be explored.

D. Observation-based forward slicing

There are two ways to slice the source code. A *backward slice* leaves the source code which influences the target slicing criteria. A *forward slice* leaves the source code that is influenced by the target slicing criteria. ORBS is a backward slicing method; it focuses on preserving the trajectory of only one slicing criterion. In contrast, multi-criteria slicing checks the trajectory of every variable, Thus it can infer the forward slice from lots of observations. One possible way is as follows: the forward slice, S_c, of the slicing criteria, c, is a set of statements which contains the other criterion whose trajectory changed when c has also been changed.

E. Parallelization Opportunities

Observational slicing brings many opportunities for parallelization. An excellent example is P-ORBS (Parallel ORBS) [27]. The implementation of P-ORBS is similar to that of W-ORBS except that it attempts to delete a set of windows sizes in parallel and then selects the largest deletion that produced the correct execution semantics. This enables the slicer to more quickly delete large blocks of code. While T-ORBS brings less inherent opportunity for parallelization, it is possible, for example, to consider the deletion of a set of siblings in the tree in parallel. In this light MOAD, by virtue of building training data from a set of executions has obvious parallelization opportunities. For example, looking at Algorithm 1, each iteration of the loop on Lines 4-9 is independent.

VII. Related Work

There have been multiple proposals to approximate dependencies by complementing a statically extracted graph with dynamic information [2], [28], [29]. A notable example is Baah et al. [2] who proposed to annotate traditional program dependence graphs (PDGs) with probabilities that capture dependence relations from dynamic execution of test cases. They extend the PDG with nodes having conditional probabilities that relate states of child nodes to the states of their parents. Similarly, Feng and Gupta [28] (using Bayesian Networks) and Gong et al [29] (using direct calculations of conditional probabilities) model the correctness of program statements (for fault localization) based on the control flow graph. Santelices and Harrold use probability models to predict the likely impact of changes in forward slicing [30] by augmenting static forward slices with relevance scores (labelling dependence edges in the interprocedural dependence graph with probabilities relating to coverage and propagation). While we also propose probabilistic modeling of dependencies of program elements we do not require an existing program dependence graph nor do we need detailed information about the state of the executing program. Our coarser modeling is thus more generally applicable while still being useful, at least for slicing. Future work can evaluate if the less precise models we build can also be used for fault localization and fault comprehension, as investigated by [2], [28], [29] and others, or what are the trade-offs between precision and usefulness of these models.

VIII. Conclusion

This paper introduces and studies a new technique aimed at modeling program dependence based on dynamic observation. The long term goal of this work is to enable the reformulation of dynamic program dependency analysis into a probabilistic space using statistical inference models. Doing so should lower analysis costs. Furthermore, the cost is all upfront: once the model is built, inferring results is very inexpensive. At the same time, the reformulation aims to retain the strengths of existing purely dynamic dependency analysis (no need for prior static analysis, language agnostic, scalable).

To illustrate the value of this new technique the paper studies MOAD, which uses the inference model to produce (observation based) dynamic slices. The results, presented in Section V, illustrate the value of the probabilistic model. Specifically, MOAD can produce slices that are only 16% larger than the 1-minimal W-ORBS slices on average, from models built using only 18.7% of observations required by W-ORBS. Furthermore, once the model is trained, the slicing cost is negligible.

References

[1] P. E. Livadas and P. K. Roy, "Program dependence analysis," in *Proceedings of the International Conference on Software Maintenance 1992*. Los Alamitos, California, USA: IEEE Computer Society Press, Nov. 1992, pp. 356–365.

[2] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[3] Zhifeng Yu and V. Rajlich, "Hidden dependencies in program comprehension and change propagation," in *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, May 2001, pp. 293–299.

[4] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 498–516, 1997.

[5] K. B. Gallagher, "Using program slicing in software maintenance," Ph.D. dissertation, University of Maryland, Baltimore, Maryland, December 1989.

[6] Á. Hajnal and I. Forgács, "A demand-driven approach to slicing legacy cobol systems," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 67–82, 2012.

[7] R. Ettinger and M. Verbaere, "Untangling: A slice extraction refactoring," in *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, ser. AOSD '04. New York, NY, USA: ACM, 2004, pp. 93–101. [Online]. Available: http://doi.acm.org/10.1145/976270.976283

[8] R. Karim, F. Tip, A. Sochurkova, and K. Sen, "Platform-independent dynamic taint analysis for javascript," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[9] S. Jiang, C. McMillan, and R. Santelices, "Do programmers do change impact analysis in debugging?" *Empirical Software Engineering*, vol. 22, no. 2, pp. 631–669, Apr 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9441-9

[10] S. Horwitz, T. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[11] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 109–120.

[12] ——, "ORBS and the limits of static slicing," in *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM 2015, September 2015, pp. 1–10.

[13] N. Gold, D. Binkley, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Generalized observational slicing for tree-represented modelling languages," in *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 547–558.

[14] D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo, "A comparison of tree- and line-oriented observational slicing," *Empirical Software Engineering*, Jan 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9675-9

[15] S. Yoo, D. Binkley, and R. Eastman, "Observational slicing based on visual semantics," *Journal of Systems and Software*, vol. 129, pp. 60–78, July 2017.

[16] M. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, Ann Arbor, MI, 1979.

[17] ——, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[18] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[19] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," in *Int'l Conf. Softw. Maintenance*. IEEE, 2013, pp. 516–519.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.

[21] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*. Springer, 2003, pp. 63–71.

[22] J. W. Tukey, *Exploratory Data Analysis: Limited Preliminary Ed.* Addison-Wesley Publishing Company, 1970.

[23] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '96. New York, NY, USA: ACM, 1996, pp. 121–134.

[24] K. Chaloner and I. Verdinelli, "Bayesian experimental design: A review," *Statistical Science*, pp. 273–304, 1995.

[25] M. Zuluaga, G. Sergent, A. Krause, and M. Püschel, "Active learning for multi-objective optimization," in *International Conference on Machine Learning*, 2013, pp. 462–470.

[26] F. V. Jensen, *Introduction to Bayesian Networks*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 1996.

[27] S. Islam and D. Binkley, "PORBS: A parallel observation-based slicer," in *24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–3.

[28] M. Feng and R. Gupta, "Learning universal probabilistic models for fault localization," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2010, pp. 81–88.

[29] D. Gong, X. Su, T. Wang, P. Ma, and W. Yu, "State dependency probabilistic model for fault localization," *Information and Software Technology*, vol. 57, pp. 430–445, 2015.

[30] R. Santelices and M. J. Harrold, "Probabilistic slicing for predictive impact analysis," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-10-10, 2010.