Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Observation-based approximate dependency modeling and its use for program slicing[☆]

Seongmin Lee[a,*], David Binkley[b], Robert Feldt[c], Nicolas Gold[d], Shin Yoo[a]

[a] KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea
[b] Loyola University Baltimore, 4501 North Charles Street, Baltimore, MD 21210, USA
[c] Chalmers University of Technology, Chalmersplatsen 4, 412 96 Göteborg, Sweden
[d] University College London, Gower St, London WC1E 6BT, UK

ARTICLE INFO

Article history:
Received 24 September 2020
Received in revised form 15 March 2021
Accepted 30 April 2021
Available online 3 May 2021

Keywords:
Dependency analysis
Program slicing
Model learning
MOAD

ABSTRACT

While dependency analysis is foundational to much program analysis, many techniques have limited scalability and handle only monolingual systems. We present a novel dependency analysis technique that aims to approximate program dependency from a relatively small number of perturbed executions. Our technique, MOAD (Modeling Observation-based Approximate Dependency), reformulates program dependency as the likelihood that one program element is dependent on another (instead of a Boolean relationship). MOAD generates program variants by deleting parts of the source code and executing them while observing the impact. MOAD thus infers a model of program dependency that captures the relationship between the modification and observation points. We evaluate MOAD using program slices obtained from the resulting probabilistic dependency models. Compared to the existing observation-based backward slicing technique, ORBS, MOAD requires only 18.6% of the observations, while the resulting slices are only 12% larger on average. Furthermore, we introduce the notion of the observation-based forward slices. Unlike ORBS, which inherently computes backward slices, MOAD's model's dependences can be traversed in either direction allowing us to easily compute forward slices. In comparison to the static forward slice, MOAD only misses deleting 0–6 lines (median 0), while excessively deleting 0–37 lines (median 8) from the slice.

1. Introduction

Understanding dependency between program elements is a fundamental task in software engineering (Livadas and Roy, 1992; Baah et al., 2010). It provides a basis for many software engineering tasks including program comprehension (Zhifeng Yu and Rajlich, 2001), software testing (Binkley, 1997), maintenance (Gallagher, 1989; Hajnal and Forgács, 2012), refactoring (Ettinger and Verbaere, 2004), security (Karim et al., 2019), and debugging (Jiang et al., 2017). The traditional static approach based on dependence graphs (Horwitz et al., 1990) has been widely adopted but suffers from issues such as its inability to easily handle multilingual systems (combining analyses for multiple languages can be quite complex) and limited scalability (partial analysis of a large system is not viable using static approaches that require whole-program analyses).

Observation Based Slicing (ORBS) (Binkley et al., 2014, 2015; Gold et al., 2017; Binkley et al., 2019) 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 *slicing criteria* (i.e., the variable of interest shows the same behavior after deletion with respect to a test suite). When deletions are made at the line-of-text level, ORBS is entirely language agnostic (Binkley et al., 2014; Gold et al., 2017; Binkley et al., 2019; Lee et al., 2020) and can analyze files for which the grammar is unavailable/unknown, and analyze languages with unconventional semantics such as Picture Description Languages (PDLs) (Yoo et al., 2017). Despite its benefits along with a lightweight implementation it needs, ORBS has one clear drawback: the cost of analysis. Being a purely dynamic approach, it iteratively attempts to validate its speculative deletion of every program element via compilations and test executions. As such, it can incur significant cost.

This paper investigates the feasibility of *approximate* dependency analysis *at a greatly reduced cost*. A precise dependency analysis aims to 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

[☆] Editor: Gabriele Bavota.
[*] Corresponding author.
E-mail addresses: bohrok@kaist.ac.kr (S. Lee), binkley@cs.loyola.edu (D. Binkley), robert.feldt@chalmers.se (R. Feldt), n.gold@ucl.ac.uk (N. Gold), shin.yoo@kaist.ac.kr (S. Yoo).

https://doi.org/10.1016/j.jss.2021.110988
0164-1212/© 2021 Elsevier Inc. All rights reserved.

depends on B: the outcome is a real number. While probabilistic program dependency analysis techniques have been proposed before (Baah et al., 2010) they require an initial static analysis which is then extended with probabilistic information based on test executions. We conjecture that a more general analysis, based solely on dynamic observations can still be useful in many program analysis contexts while being significantly less costly.

The approximate nature of our approach stems from the fact that it *infers* a stochastic model of program dependences. Unlike ORBS which performs iterative deletions to analyze program dependency with respect to a single program element (i.e., the slicing criterion), our technique, MOAD (Modeling Observation-based Approximate Dependency), learns an approximate model of program dependence from significantly fewer dynamic observations. Intuitively, ORBS makes a single slice increasingly more accurate by iterative deletion. In contrast, MOAD employs a set of deletions that can be treated independently. For each, it observes multiple program elements and can thus, ultimately, learn more from 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 uncovering dependence information by iteratively deleting until it arrives at a one-minimal slice (Binkley et al., 2014).
- The output of MOAD can be used to construct multiple backward and forward slices, whereas a single ORBS run produces a single backward slice.
- Moreover, since the observations required by MOAD are independent from each other, MOAD is inherently parallel.

To evaluate MOAD, we have implemented it 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 producing slices based on the MOAD produced model: 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 program dependences; a comparison to ORBS slices shows that MOAD can produce slices that are only 16% larger than ORBS slices, while using only 18.7% of the observations. We also investigate various ways to construct the observation sets, as well as the impact of different inference models.

In earlier work (Lee et al., 2019) we presented the basic technique and empirically evaluated its use for producing backward slices. This paper extends our previous work in three main ways. First, we provide a more extensive comparison of backward slices produced by MOAD and those produced by ORBS. Secondly, we compare MOAD-based forward slices to those of the CodeSurfer static analysis tool from Grammatech Inc. (2002). Finally, we also describe our framework in more depth and provide more extensive experiments to show the relative merits of different deletion schemes as well as inference models when inferring MOAD dependency models. Taken together this extends the use of the underlying modeling technique and provides a richer explanation both of its benefits as well as clarifying its limitations. As hinted at in Section 2, our study of slicing only scratches the surface. Our probabilistic dependency modeling is far more general. It can be applied to a wide range of problems such as Fault Diagnosis (Baah et al., 2010). However, we leave these additional use cases to future work and focus here on slicing as a representative example.

To summarize, the technical contributions of this paper are as follows:

- We introduce the concept of learning approximate dependency analysis, which transforms the dependency relationship from Boolean to probabilistic.
- We present MOAD, a technique that models approximate program dependency, and describe its essential steps: how to generate observations and how to infer models from the observations.
- We conduct an empirical evaluation of MOAD via backward program slices instantiated from the learned models.
- We show how to instantiate forward program slices from the models learned by MOAD and empirically compare these slices to those of a static analysis tool.

The rest of this paper is organized as follows. Section 2 introduces the concept of approximate dependency analysis, and explains how it relates to the existing slicing technique ORBS. Section 3 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 4 presents the set-up of empirical evaluation, the results of which are reported in Section 5. Section 6 contains discussions of our findings and potential future work. Section 7 presents the related work, and Section 8 concludes.

2. Approximating program dependency

Program dependency is dependence relations that hold between elements of a program (e.g., statements, expressions, or variables). If the computation of an element a directly or indirectly affects the computation of another element, b, we consider b to be dependent on a. A plethora of techniques have been proposed to capture and model dependence information.

Often these techniques are static and require parsing and detailed analysis of program elements based on the semantics of the programming language in question. The outcome of the analysis typically captures a binary dependence relation where two program elements either may or may 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 (Baah et al., 2010) (static-then-annotate) or do not build any explicit model at all (Binkley et al., 2014).

One downside of the static and static-then-annotate approaches is that they cannot easily 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 (Binkley et al., 2014) (ORBS) addresses these problems and allows language-agnostic dynamic slicing without detailed semantic knowledge, by reusing the existing build chain. 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 single backward slice through iterative build and execute cycles. The process has to be repeated if another slicing criterion is selected for analysis, as, for example, typically happens in fault diagnosis or debugging.

If we consider the notion of a "perfect" dependence model that accurately captures the dependences in the program, a learning approach can be seen as aiming to approximate that perfect model as closely as possible for minimal effort in construction.

Classical ORBS (Binkley et al., 2014) could be seen as a learning technique: in effect, it starts from a perspective of "total dependency" assuming that everything is dependent on everything else (like a dependence cluster (Binkley and Harman, 2005)) and attempts to discover and learn total independence (w.r.t. the criterion) through deletion and execution by permanently removing independent elements. However, learned dependence/independence is not explicitly modeled separately from the code (the code state itself represents the current "understanding" of dependence at any given point in the process — information about independent program elements is lost during the process since they no longer exist).

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 explicitly and 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 our model building is based on a few, specific, dynamic executions it can only *approximate* the complete dependency information. However, explicitly building such probabilistic models brings potential advantage over model-free, dynamic approaches such as ORBS, which do not learn anything between invocations. For one, we can learn about multiple dependences for each execution. We can also use one and the same model for multiple program analysis tasks. We posit that there exists an interesting and complementary trade-off between what we propose and existing program analysis methods.

Conceptually, the shift from traditional, precise program dependency models to approximate dependence modeling might seem large. However, we argue that this is mainly a matter of perspective because which technique is a better fit depends on the specific analysis task at hand.

In a precise analysis the outcome is a binary relation stating that a program element (B) depends on another (A). However, when we consider the full space of inputs of the program there can be a subset of inputs for which the value at B does not depend on A. The precise analysis thus answers the question "Is program element B, for at least one actual input, dependent on program element A?" In contrast, an approximate analysis answers the question "What is the probability that program element B is affected by program element A?" To the latter question the precise analysis can only give an approximate answer. The technique we propose here also gives an approximate answer, since it samples only a subset of inputs and is typically not learning an optimal model. However, the approximation can be expected to improve with additional observations and with better inference algorithms. Which question and which approximation to it is preferable depends on the downstream analysis task to which the model will be applied, and also on the semantics of the code being analyzed as well as the complexity of the input space.

While the idea of approximate program dependency modeling is a general one, we focus below on an instantiation that targets program slicing. This allows us to study the potential benefits compared to a well-known and general technique. In the context of the above model-view discussion, slicing is one view of the model, with a particular direction of dependence ascribed to it. Thus in the following we consider MOAD in the context of program slicing.

3. MOAD: Modeling observation-based approximate dependency

This section first overviews the key terminology used to describe MOAD, before describing its two phases: the observation phase and the inference phase. The output of the first phase is a set of observations. These observations form the input to the inference phase, which builds an inference model M that aims to capture the dependence within the program. As a case study, we illustrate M by inferring program slices (Weiser, 1979, 1984). In the next section we compare the inferred slices with those produced using ORBS (Binkley et al., 2014, 2015) and the static slices produced by Codesurfer (Grammatech Inc., 2002).

3.1. Terminology

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* $U = \{u_1, \ldots, u_{|U|}\}$. For example, U might be composed of lines of text (Line 1, Line 2, ..., Line n), program statements (assign-statement, if-statement, etc.), or blocks of code (a whole method or a class, etc.) within a program. We subsequently create sub-programs of P by deleting one or more units from P. To support the inference process, we represent a 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.

Program slicing (Weiser, 1979, 1984) produces a subset of a program that relates to the value that a specific variable at a specific location, what we call a *slicing criterion*, has. There are two types of program slices: a *backward slice* consists of program elements that affect the value in the criterion, and a *forward slice* consists of program elements that are affected by the value in the criterion.

Borrowing the notion from program slicing, we assess the impact of deleting various units at a set of (slicing) criteria, $C = \{c_1, \ldots, c_{|C|}\}$. Each criterion 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 (*trajectory*) produced by each criterion. To do this, we implement the original program to output the value in the criteria by annotating printing scripts after each criterion, similar to ORBS (Binkley et al., 2014). 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 original program). The key assumption here is that, if the deletion of unit u_a brings about a change in the trajectory for criterion c_b, then the criterion c_b likely depends on (some part of) unit u_a.

3.2. Observation phase

The core of the first phase is a *deletion generation scheme*, which generates the set of deletions used to produce program mutants. Our experiments consider the two n-hot deletion schemes: 1-hot and 2-hot. The first, 1-hot generates $|U|$ deletions where each deletion removes exactly one unit. In other words, it generates all of the one-hot encoding vectors of length $|U|$. The second, 2-hot subsumes 1-hot; it considers both all single units and all pairs of units. The relation between the number of observations in 1-hot and 2-hot is, assuming there are n deletable units in the program, 1-hot requires $O(n)$ observations, while 2-hot requires $O(n^2)$ observations. (Because 2-hot involves considerably more deletions than 1-hot, in Section 5.5 we also consider the impact of sampling the 2-hot data.)

Algorithm 1 describes the observation phase. Its input includes, P, the program under study, I, the input test suite, and a deletion generation scheme, GENSCHEME. The algorithm first initializes the output observation set, O, to the empty set, E to the expected output sequence for each criterion (collected from the execution of P and I), and the set of *deletions* generated using the given scheme. Subsequently, Lines 4–9 process each deletion

Algorithm 1: Observation phase

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

Algorithm 2: Inference phase

```
input  : $P$: an input program
         $C$: a set of slicing criteria
         $O$: a set of observations
         $dsg\_mdl$: a design of model (one of $\mathbb{O}, \mathbb{L}, \mathbb{B}$)
output: $\mathcal{IS}$: set of inferred slices; one for each slicing
         criterion $c_k \in C$
$M \leftarrow dsg\_mdl$.Build $(O)$
$\mathcal{IS} \leftarrow \{\}$
for $c_k \in C$ do
    $deletion \leftarrow M(c_k)$
    $P_k \leftarrow$ Apply $(P, deletion)$
    $\mathcal{IS} \leftarrow \mathcal{IS} \cup \{P_k\}$
return $\mathcal{IS}$
```

by first using the function Apply to generate sub-program P' composed of only the non-deleted units (omitting 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', to 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 observations to be returned by the algorithm. For readers familiar with ORBS, the main difference between Algorithm 1 and ORBS is that while ORBS is a cumulative process to produce a backward slice for a single slicing criterion, each observation in Algorithm 1 is independent, thus they can be done in parallel.

To illustrate Algorithm 1, consider a program P with three statements S_1, S_2, S_3, and two criteria C_1 and C_2 using the 1-hot generation scheme on the statement level units. With three statements, 1-hot produces the three deletions {True, False, False}, {False, True, False}, and {False, False, True}. 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). It produces a response vector of two boolean values, where each of them represents whether the trajectory of the corresponding criterion is preserved or not. The tuple of the deletion and the response becomes a single observation. This process is then repeated for each of the other deletions and resulting two-statement programs.

3.3. Inference phase

The inference phase, described in Algorithm 2, infers the backward slices for a set of criterion. The algorithm uses the set of observations, O, (output by the first phase) to build an inference model, $M : C \rightarrow D = \mathsf{Boolean}^{|U|}$, where D is the set of possible deletions. Then, for each slicing criterion, M is used to infer a set of deletions, which, when applied to the original program, produce a backward 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 backward slice of c_k. We call this assumption a "*trajectory change assumption*". While this connection is straightforward, this data tends to overestimate deletability for the 1-hot data. For example, when either of two statements can be deleted, but not both (Binkley et al., 2014), 1-hot data cannot provide the failing situation when both are deleted misleading the model to generate an invalid slice. In contrast, for n-hot with $n > 1$, it is possible that not all of the deleted units influence c_k. In this study, we designed three different inference models (*dsg_mdl*) that instantiate the trajectory change assumption: Once Success ($\mathbb{O}$), Logistic ($\mathbb{L}$), and Bayesian ($\mathbb{B}$). We describe each of the models in following sections in detail.

The inference step is also necessarily constrained by the resolution of the observations. For example, if the granularity of the unit is a function-level, it is infeasible for MOAD to remove a statement that is independent from the criteria if another statement in the same function affects the criteria from the slice. The observations described in Section 3.2 identify the criteria that are potentially influenced by each deletable unit. Note that all of the criteria are part of a deletable unit, but some deletable units (e.g., the break statement in C) include no criteria.

The asymmetry between the granularity of the deletion and the granularity of the criterion of the observation impacts the slices that MOAD can produce. When computing a backward slice, the possible slicing criteria match the criteria used in the observations, and the slices are composed of the deletable units from the observations. However, when we consider forward slicing in Section 5.6, these two are reversed. Thus when computing a forward slice MOAD can accept as a slicing criteria any deletable unit, but it outputs as the slice a subset of the criteria from the Observation Phase. Because this inversion can be a source of confusion, when we compare the forward slices generated by MOAD to those from other slicers, we consider the lines of text involved in the criteria and in the deletable units (statements).

The remainder of this subsection details three different inference models studied in the next section. As a notational convenience, hereafter we use "0" and "1" to denote "False" and True", respectively.

3.3.1. Once success ($\mathbb{O}$)

The Once Success model explicitly follows the aforementioned assumption (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 backward slice of c_k). Assume subprogram P' is obtained from program P by removing deletion unit u_m. 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, $M_{\mathbb{O}}$, built with observations O, infers 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 mth element of deletion vector s and $r[k]$ represents the kth 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.

3.3.2. Logistic ($\mathbb{L}$)

Our second model, "Logistic", statistically implements our assumption by performing a logistic regression. In logistic regression, the coefficient represents how an independent variable helps make the dependent variable True (1) or False (0). To be more specific, if the sign of the coefficient is positive, the dependent variable has a higher chance of being true when the independent variable increases. Conversely, if the coefficient is negative, the dependent variable has a lower chance to be true when the independent variable increases (Szumilas, 2010).

The Logistic model regards the response element, $r[k]$ (for slicing criterion c_k) as a dependent variable and the elements of the deletion, d, as the independent variables.

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

The elements of β_k represent the regression coefficients, and we interpret their signs to decide whether each unit should be in the slice or not. If $\beta_k[m]$, the mth coefficient of β_k, has a positive value, it means that when we deleted u_m, it gave, on average, a higher chance to change the trajectory. On the other hand, if we delete a unit that does not affect the criterion, then the probability of trajectory change can only and will, on average, go down. Thus, the corresponding coefficient will be negative. Notice that, for the same reason, units that have a considerably smaller effect on the criterion compared to other affecting units could also have a negative coefficient. For simplicity, we use zero as the threshold for coefficients and decide to include elements with positive coefficients while excluding those with zero or negative coefficients. Future work could optimize the threshold value or even sample slices based on the sign and size of the coefficients. More formally, $M_{\mathbb{L}}$, the Logistic model, infers the deletion vector for the slice taken with respect to c_k as follows:

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

3.3.3. Bayesian ($\mathbb{B}$)

The final model we consider uses Bayesian inference. While the Once Success model strictly reflects the trajectory change assumption, it may overestimate the independence as we described before. In the Bayesian model, we represent the dependence in the conditional probability and generate a slice by applying a certain threshold. 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. There would be a difference in how much it depends on other units for each criterion. Thus, use an average of the conditional probability for each criterion as a threshold instead of a fixed value. Formally, $M_{\mathbb{B}}$, the Bayesian model, infers 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.

3.4. Observation-based forward slicing

An important advantage that MOAD has over ORBS, apart from the improved efficiency, is that it enables observation based *forward* slicing. In contrast to backward slicing, which aims to find program elements that affect the slicing criterion, forward slicing identifies all program elements that are *affected by* the slicing criterion (Horwitz et al., 1990).

Given a Program Dependence Graph (PDG), both backward and forward slicing can be thought of as finding the transitive closure of the program dependence: the transitive closure following dependence direction is the forward slice, while the one following the reverse dependence direction is the backward slice. ORBS approximates the backward slice by iteratively attempting to delete program elements. The iterative approach works because, essentially, all dependence converges to the slicing criterion when performing backward slicing: it suffices to observe only the slicing criterion, as the impact of any deletion that is relevant to the backward slice will be observed at the criterion.

ORBS, however, cannot easily compute forward slices. In contrast to backward slices, the dependence in forward slices fans out from the slicing criterion to multiple program elements in the forward slice. Consequently, the only way of obtaining a forward slice of a given criterion c using ORBS would be to compute backward slices from all the other program elements, and include any element that has c in its backward slice in the forward slice of c. Given that analysis cost is already a weakness of ORBS, computing a forward slice from multiple backward slices would not be an attractive approach.

On the other hand, MOAD can efficiently compute forward slices since, by definition, it approximates the dependence model itself. The main difference between ORBS and MOAD when generating a forward slice is how they utilize information from an observation. While ORBS only recognizes a single relation regarding a deleted unit affecting a single criterion, MOAD captures multiple relations, specifically which criteria are affected by deleted units.

4. Experiment setup

4.1. Research questions

We evaluate MOAD by investigating the following six research questions. The first RQ concerns whether MOAD has the capability to approximate program dependency. With RQ2 to RQ4, we evaluate MOAD by comparing its backward slices to those generated by ORBS and the static slicer of CodeSurfer. With RQ5, we evaluate how the amount of observation affects MOAD's accuracy. Finally, with RQ6, we compare the forward slices generated by MOAD with those generated by CodeSurfer.

RQ1. Viability: *Do the learned models correctly capture program dependence information?* MOAD is the first approach that stochastically models program dependence without requiring knowledge obtained from expensive static analysis; there is no guarantee that MOAD would have learned the program dependence from scratch. To investigate if our approach is viable, we compare the ability of our learned models to produce slices against that of a random slicer, which produces a random subset of a program as a slice without any prior knowledge. If none of the models can outperform a random slicer, there is no reason to further consider them.

RQ2. Impact of Model Type: *Which inference model performs the best?* Given our use of program slicing in our evaluation, to answer RQ2, we compare the size of slices generated using each of the three inference models.

RQ3. Comparison to ORBS: *How does the best performing inference model compare to ORBS?* MOAD is a purely dynamic approach that builds inference models using dynamic information only. The closest related baseline approach is observational slicing. We consider two implementations, W-ORBS (Binkley et al., 2014) and T-ORBS (Gold et al., 2017; Binkley et al., 2019): while both are expected to produce more accurate slices when compared to MOAD, they are also expected to take longer. In this research question, we evaluate both the quality and the efficiency of MOAD in comparison to W-ORBS and T-ORBS.

RQ4. How does the best performing inference model compare with a static slicer? We perform a quantitative analysis of backward slices generated by MOAD, and those generated by the widely studied static backward slicer, CodeSurfer. RQ4 is addressed by a complete enumeration study for every line in the three C benchmark programs, mbe, mug, and wc, investigating why each line is in one slice but not in the other, or vice versa. In this way, we analyze the root causes of the differences between the two.

RQ5. Sampling effect: *What is the impact of using sampled observations?* We investigate whether we can further reduce the analysis cost of MOAD by using sampled subsets of the observations. The hypothesis is that models built with more observations will produce better slices at higher analysis cost. We consider ten different sampling rates when sampling from the 2-hot observations of each subject program, from 10% to 100% of all available observations, and build inference models using the sampled observations. To cater for the randomness in sampling, we repeat the process ten times for each sampling rate. Subsequently, we compare the size of slices that are generated by models built with different amounts of the sampled observations.

RQ6. How do MOAD's forward slices compare to static forward slices? While originally designed with backward slicing in mind, we consider the related operation of forward slicing to provide some insight into how well MOAD generalizes to other program dependences analyses. To better understand the forward slicing capability of MOAD, we qualitatively compare its forward slices with those computed by CodeSurfer.

4.2. Baseline

This section describes two baseline approaches: ORBS, a purely dynamic slicing approach, and CodeSurfer, a dependence analysis tool that can produce static slices.

4.2.1. Observation-Based Slicing (ORBS)

We use Observation-Based Slicing (ORBS) (Binkley et al., 2014) 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 (Binkley et al., 2014), 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 (it is impossible to delete any single line from the slice) (Binkley et al., 2014).

A recent variation of W-ORBS, T-ORBS (Gold et al., 2017; Binkley et al., 2019) 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 (Collard et al., 2013) 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 is to provide an apples-to-apples comparison between the output of the two slicers. Doing so requires introducing a small amount of language-specific information into an otherwise language-agnostic algorithm but provides a better basis for comparison. This also brings a dramatic speed-up because T-ORBS can spend considerable time as it considers the numerous sub-trees that represent the constituent parts of a given statement.

4.2.2. CodeSurfer

We use CodeSurfer to compute the static program slicing baselines. CodeSurfer is a static analysis tool produced by GrammaTech (Grammatech Inc., 2002). It computes static slices by first converting a program into a System Dependence Graph (SDG) (Horwitz et al., 1990) and then solving a reachability problem over this graph. The resulting set is often referred to as a dependence-closure slice, which among other things implies that it is not necessarily an executable program (Binkley, 1993).

Relevant to the experiments with MOAD, a CodeSurfer slice is returned at the vertex level, which represent a finer level of granularity than that used by MOAD. For example, consider the statement `x = x + a++`. CodeSurfer is capable of identifying that `a++` but not this entire statement is part of a slice.

Fortunately, CodeSurfer maintains a mapping connecting the original text of the program to the vertices and thus it is possible to map a CodeSurfer slice into a set of line numbers by exploiting this mapping. This mapping can also be used to map a line number into the set of vertices that overlap with the source code found on the given line. Thus in the experiment we identify the line associated with each slicing criteria, map that line to a set of vertices from which we compute a slice and then finally map the vertices back to a set of line numbers for comparison with MOAD. For example, Fig. 1 shows an excerpt from the output of CodeSurfer slicing script.

4.3. Configuration

In the initial experiments, the units considered by MOAD are programming language statements. We use SrcML (version 0.9.5) (Collard et al., 2013), 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 inherits ORBS' ability to be applied to any language and in multi-lingual analysis.

The set of slicing criteria considered consists of all arithmetic (`char`, `int`, `float`, etc.) assignments. We use Clang (version 3.8) (Lattner and Adve, 2004) to insert logging statements for each slicing criterion. These statements are responsible for outputting the sequence of values computed for the criteria. The goal of a slicer is to preserve this sequence while removing unnecessary code.

```
slice taken with respect to variable c on line 15 -> (9 12 15)
slice taken with respect to variable x on line 16 -> (9 13 16)
slice taken with respect to variable i on line 17
-> (9 11 14 17 24 28 30 32 36 50 52)
```

Fig. 1. Example line-level mapping of three CodeSurfer slices from the program mug.

Table 1
The statistics of experiment subjects' properties. Each of prttok, prttok2, replace, sched, sched2, totinfo, tcas represents printtokens, printtokens2, replace, schedule, schedule2, totinfo, tcas in the Siemens programs.

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

As a baseline, we apply both W-ORBS (with maximum window size of three) and (the modified) T-ORBS to our subjects. CodeSurfer we apply version 2.2p0 in its default configuration. 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.

4.4. Subjects

Table 1 shows the programs we study. It includes the number of non-comment-non-blank lines (SLoC), the number of units statements, and the number of criteria used. The first three subjects, mbe, mug, wc (word count), 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 (Do et al., 2005), 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 analysis. Both subject program sets (the three small programs and the *Siemens* suite) are widely used in the program dependence literatures (Horwitz et al., 2009; Santelices and Harrold, 2010; Binkley et al., 2014, 2015; Lee et al., 2020). Thus, their use would make it easy to compare our results with those of other existing literature techniques. Several previous papers have found the observation-based approach is viable for language-agnostic program dependence analysis (Binkley et al., 2014; Gold et al., 2017; Binkley et al., 2019; Lee et al., 2020). Leveraging this prior work, we use only C programs to evaluate MOAD in this study, while still being applicable to programs involving other programming languages. Considering the relative modeling performance of MOAD applied to C as opposed to other languages is outside the scope of this study.

4.5. Metrics

We use the following metrics to compare slicer performance. In the definitions we denote that slices computed by W-ORBS, T-ORBS, CodeSurfer, and MOAD, for the slicing criteria c, as WS_c, TS_c, CS_c, and MS_c, respectively.

- Success Rate: For a MOAD model, the *success rate* is the number of inferred slices that successfully compile and preserve the trajectory of the target slicing criterion divided by the number of slicing criteria.
- $|O_{\text{method}}|$: The number of *observations* used by each slicing method (including W-ORBS, T-ORBS and MOAD), when generating slices. An observation consists of a compilation and the subsequent execution assuming the compilation is successful.
- $\mu\ (WS_c)$, $\mu\ (TS_c)$, $\mu\ (CS_c)$, and $\mu\ (MS_c)$: The mean slice size, given as a percentage of the original program's size, generated by each W-ORBS, T-ORBS, CodeSurfer, and MOAD respectively.
- miss: Given a reference slice (e.g., WS_c) and an inferred slice (e.g., MS_c), the number of deletable units that are *expected to have been deleted (i.e., that were missed)* relative to the reference slice. In other words, miss is the number of deletable units in the inferred slice that are not in the reference slice.
- excess: Given a reference slice (e.g., WS_c) and an inferred slice (e.g., MS_c) the number of deletable units that are *excessively removed* from the inferred slice relative to the reference slice. In other words, excess is the number of units in the reference slice that are not in the inferred slice.

By design T-ORBS and MOAD operate on the same set of deletable units. Thus, we can calculate miss and excess directly from the slices produced by these two. However, the same is not true of W-ORBS and CodeSurfer. Thus when comparing them with MOAD we map each MOAD slice to the line-of-text level.

For MOAD we build a *statement-line* mapping and use it to convert a statement-level slice of MOAD to one in the line-level. SrcML with an option `position` provides the line number and the column number of the program elements as attributes in corresponding XML elements. We utilize such information to build a statement-line mapping. For CodeSurfer exploit its mapping from its internal dependence graph representation to lines of source text.

Two other granularity issues arise when comparing MOAD and CodeSurfer slices. First, when MOAD deletes a statement that contains one or more other statements (e.g., when it deletes an if statement), it necessarily deletes all the contained statements. Thus it cannot observe the effect of deleting solely the containing statement. For this reason we ignore containing statements such as if statements when comparing MOAD and CodeSurfer. Second, CodeSurfer slicing criteria are constructed using vertices from its internal graph representation. This is often be a much finer level of granularity than a complete line of text (e.g., consider the line of text *to++ = *from++). To compute a line-level slice with respect to a set of lines L, we use the union of all the vertices that represent code found on the lines of L.

Table 2
Success rate of MOAD on the ten test subjects.

Subject	Deletion gen. scheme	Success rate			Subject	Deletion gen. scheme	Success rate		
		$\mathbb{O}$	$\mathbb{L}$	$\mathbb{B}$			$\mathbb{O}$	$\mathbb{L}$	$\mathbb{B}$
mbe	1-hot	100%	100%	100%	replace	1-hot	7%	31%	28%
	2-hot	100%	100%	100%		2-hot	3%	13%	31%
mug	1-hot	100%	100%	100%	sched	1-hot	48%	47%	41%
	2-hot	100%	100%	100%		2-hot	39%	35%	43%
wc	1-hot	100%	100%	100%	sched2	1-hot	30%	26%	28%
	2-hot	88%	76%	100%		2-hot	17%	26%	28%
prttok	1-hot	3%	4%	11%	totinfo	1-hot	52%	50%	62%
	2-hot	3%	3%	11%		2-hot	32%	10%	65%
prttok2	1-hot	72%	19%	77%	tcas	1-hot	48%	90%	48%
	2-hot	63%	13%	67%		2-hot	26%	68%	48%

Table 3
$\mu\,(WS_c)$, $\mu\,(TS_c)$, and $\mu\,(MS_c)$ denote the mean slice size, given as a percentage of the original program's size, generated by each of W-ORBS, T-ORBS and MOAD, respectively. Columns 4–6 and 7–9 show $\mu(MS_c)$ separately for each of the three models where the smallest mean for the three is shown in bold. Numbers in the parentheses is the ratio of $\mu(MS_c)$ to $\mu\,(WS_c)$.

Subject	$\mu\,(WS_c)$	$\mu\,(TS_c)$	$\mu\,(MS_c)$, 1-hot			$\mu\,(MS_c)$, 2-hot		
			$\mathbb{O}$	$\mathbb{L}$	$\mathbb{B}$	$\mathbb{O}$	$\mathbb{L}$	$\mathbb{B}$
mbe	35.1%	40.6%	**44.4% (1.26)**	46.6% (1.33)	44.8% (1.28)	**41.5% (1.18)**	46.2% (1.32)	44.8% (1.28)
mug	25.6%	33.7%	**40.1% (1.57)**	49.9% (1.95)	40.9% (1.60)	**34.2% (1.33)**	46.7% (1.82)	42.4% (1.66)
wc	23.7%	30.3%	**36.1% (1.52)**	52.2% (2.21)	42.3% (1.79)	**30.3% (1.28)**	42.8% (1.81)	38.7% (1.64)
prttok	43.3%	43.2%	**50.4% (1.16)**	57.3% (1.32)	58.5% (1.35)	42.2% (0.98)	**42.0% (0.97)**	57.1% (1.32)
prttok2	37.3%	38.8%	**41.5% (1.11)**	61.6% (1.65)	47.1% (1.26)	**36.6% (0.98)**	50.8% (1.36)	46.6% (1.25)
replace	45.6%	45.9%	**52.7% (1.16)**	64.2% (1.41)	58.7% (1.29)	**41.8% (0.92)**	48.5% (1.06)	58.8% (1.29)
sched	37.3%	37.9%	**59.1% (1.58)**	66.4% (1.78)	61.0% (1.64)	**44.8% (1.20)**	57.1% (1.53)	62.2% (1.67)
sched2	35.3%	33.7%	**47.5% (1.35)**	61.9% (1.76)	54.1% (1.54)	**37.2% (1.05)**	46.6% (1.32)	53.6% (1.52)
totinfo	33.9%	39.7%	**49.3% (1.45)**	53.9% (1.59)	52.0% (1.54)	41.8% (1.23)	**30.7% (0.91)**	51.8% (1.53)
tcas	37.3%	33.3%	**45.7% (1.23)**	66.7% (1.79)	46.6% (1.25)	**38.8% (1.04)**	50.0% (1.34)	46.6% (1.25)
Average	35.4%	37.7%	46.7% (1.34)	58.1% (1.68)	50.6% (1.45)	38.9% (1.12)	46.1% (1.34)	50.2% (1.44)

5. Results

5.1. Viability (RQ1)

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 1) spread across the ten subject programs, the random slicer generates 9000 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 slice by chance.

In contrast, Table 2 shows MOAD's ability 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 the success rate for each of the three inference models, $\mathbb{O}$, $\mathbb{L}$, and $\mathbb{B}$, as the percentage of 'slices' that preserve the desired trajectory. 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 terms of Success Rate, the best performing approach is $\mathbb{B}$. It outperforms the other two in 6 of 10 cases for the 1-hot deletion generation scheme, and in 9 of the 10 with the 2-hot deletion generation scheme. Considering $\mathbb{O}$ and $\mathbb{L}$ it is interesting to note that, on average, $\mathbb{O}$ a performs worst with the 1-hot data, while $\mathbb{L}$ performs worst with the 2-hot data.

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 snippet below, where there is a data dependence from Line 188 to Line 189.

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

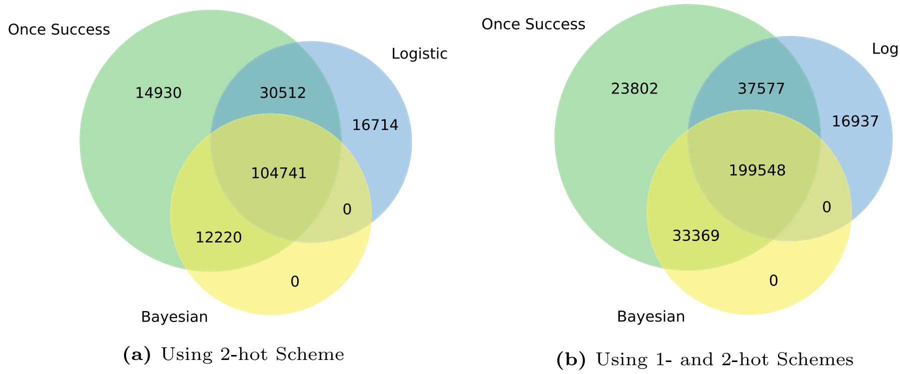
What is unusual about these two lines is that for many slices that do not depend on the value of `token_ptr` it is possible to individually delete either Line 188 or 189 without affecting the trajectory, but not both. Thus the model learns to remove each line. Consequently, when MOAD infers a slice it tends to unwantedly omit both lines. The result is that most trajectories change. A similar situation happens in replace, too. The last step of the replace program is to replace the target text with the input text, and it is triggered by a boolean variable which represents whether the input parsing has been succeeded. There are two assignment statements for that boolean variable, and deleting either one of them does not bother executing the last step. However, deleting both of the assignment statements makes the program exit before running the replacement step making lots of slices fail to preserve the trajectory. This suggests the use of stronger statistical models (e.g., Rasmussen's Gaussian processes Rasmussen, 2003), that can capture higher-level interaction effects between program elements.

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

> **RQ1. Viability:** Inference models built with dynamic observations can successfully learn program dependence.

5.2. Impact of model type (RQ2)

We evaluate the three inference models based on their ability to remove units. Table 3 shows the average slice size, $\mu\,(MS_c)$, over all slicing criteria for three models used with MOAD. To



(a) Using 2-hot Scheme

(b) Using 1- and 2-hot Schemes

Fig. 2. Venn diagrams of statements deleted by 𝕆, 𝕃, and 𝔹 under two, different deletion schemes.

facilitate inter-program comparison, the average slice sizes are given as a percentage of the original program's size.

According to the result, Once Success(𝕆) generates smaller slices than the other two inference models (ANOVA, *p-value* $<$ 0.0001). Comparing the rows of Table 3, Once Success produces the smaller slice in 18 of the 20 rows. 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.

Considering the overall impact of the training data, 2-hot produces the smaller average slice in 25 of the 30 cases. We further analyze the tendency with respect to each inference model. To the Once Success model (𝕆), the more attempts on deletion in data implies the more chances of the model to observe the independence between units and criteria. This implies that the size of the slice monotonically decreases as the set of observations is increased. Thus, Once Success built with 2-hot data ubiquitously generates smaller slices than when built with the 1-hot data. Considering the Logistic model (𝕃), it also tends to generate smaller slices using the 2-hot data, doing so for all ten subjects. This dominance illustrates that the model learns more dependency relations from the larger set of observations. Finally, the sizes of the slices generated by the Bayesian model (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 unit may increase or decrease, depending on the observations. This result is echoed in the study of RQ5 in Section 5.5.

We also investigate whether there are more qualitative differences between slices produced by 𝕆, 𝕃, and 𝔹. For this, we analyzed the subsumption relationship between sets of statements deleted by different models, and summarize the differences. Fig. 2 shows Venn diagrams of statements deleted by different models from all studied subjects and slicing criteria (Fig. 2a shows deleted statements using 2-hot scheme, while Fig. 2b shows all statements deleted using 1- and 2-hot schemes combined). By definition, all statements deleted by 𝔹 will be deleted by 𝕆, resulting in the set 𝔹 − 𝕆 being empty. This can be clearly seen in both diagrams.

Comparison of 𝕃−𝕆 and 𝕆−𝕃 is more interesting. Fig. 2 shows that 𝕆 and 𝕃 are better at deleting different sets of statements. We hypothesize that the frequency of execution (and, therefore, exposure) of different statements may interact with the relative aggressiveness of different models. Consider a statement that is frequently and commonly executed by many test cases. In some of these executions, the slicing criterion may depend on it; in other executions, it may not. 𝕆 will aggressively delete such statements, because a single trajectory-preserving execution is all it requires for deletion. 𝕃 will be more cautious, as it will consider both types of executions: ones that exhibit the dependence relationship to the slicing criterion, and others that do not.

In contrast, consider a statement that is deeply nested. It will be executed only under certain conditions: when such a statement is executed, it is likely because its execution is specifically required to affect the subsequent computation, including the slicing criterion. Consequently, deleting such a statement is likely to fail to preserve the trajectory, preventing 𝕆 from easily deleting them. Interestingly, if the deletion of a statement only infrequently affects a trajectory, 𝕃 is likely to put relatively smaller weights to the corresponding features of the logistic regression model, 𝕃.

We checked this hypothesis with a simple analysis involving average statement nesting depth in 𝕃 − 𝕆 and vice versa. We found the averages to be 5.54 for 𝕃 − 𝕆, but only 4.99 for 𝕆 − 𝕃. While the initial findings are promising, we note that this may not be the only factor that affects slice differences. Thus, this warrants further study and deeper analysis which is outside the scope of this paper.

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

> **RQ2. Impact of the inference model:** Among the three inference models, Once Success generates the smallest slices. Considering the impact of the training data, 2-hot deletion generation scheme produces smaller slices than 1-hot.

5.3. Comparison with ORBS (RQ3)

In this section, we compare the MOAD and the two implementations of ORBS regarding three aspects: efficiency, slice size, and accuracy.

5.3.1. Efficiency

Our initial look at the efficiency considers the number of observations required to generate slices. W-ORBS turns out to require fewer observations if it starts at the end of the code and works its way towards the beginning (Binkley et al., 2014). Therefore we used this version in our efficiency comparison. Table 4 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. ORBS examines every statement at least once for one slice. MOAD makes many observations, but after that, we can produce many slices from that observations. For a fair comparison, we compute the ratio between the number of resulting slices and the number of observations. The 5th, 6th, 8th, and 9th columns in Table 4 show the ratio. Patterns evident in the data indicate that, when compared to W-ORBS, MOAD requires

Table 4
$|O_{\text{W-ORBS}}|$, $|O_{\text{T-ORBS}}|$, $|O_{\text{1-hot}}|$, $|O_{\text{2-hot}}|$ denote the number of observations used by each of W-ORBS, T-ORBS, and MOAD with 1-hot and 2-hot deletion generation scheme, respectively.

Subject	$\|O_{\text{W-ORBS}}\|$	$\|O_{\text{T-ORBS}}\|$	$\|O_{\text{1-hot}}\|$	$\frac{\|O_{\text{1-hot}}\|}{\|O_{\text{W-ORBS}}\|}$	$\frac{\|O_{\text{1-hot}}\|}{\|O_{\text{T-ORBS}}\|}$	$\|O_{\text{2-hot}}\|$	$\frac{\|O_{\text{2-hot}}\|}{\|O_{\text{W-ORBS}}\|}$	$\frac{\|O_{\text{2-hot}}\|}{\|O_{\text{T-ORBS}}\|}$
mbe	6 546	1 163	46	0.70%	3.96%	948	14.5%	81.4%
mug	4 420	969	45	1.02%	4.64%	904	20.4%	93.2%
wc	3 843	860	34	0.88%	3.95%	454	11.8%	52.7%
prttok	262,374	68,372	389	0.15%	0.57%	74,175	28.3%	108.5%
prttok2	246,668	41,324	365	0.15%	0.88%	65,393	26.5%	158.2%
replace	1,214,230	311,323	466	0.04%	0.15%	106,412	8.8%	34.2%
sched	130,199	33,911	253	0.19%	0.75%	31,181	23.9%	91.9%
sched2	93,173	31,200	249	0.27%	0.80%	30,227	32.4%	96.9%
totinfo	451,715	86,499	228	0.05%	0.26%	25,151	5.6%	29.1%
tcas	40,883	11,377	111	0.27%	0.98%	5,846	14.3%	51.4%
Average:				0.37%	1.69%		18.6%	79.7%

significantly fewer observations. For example, the number of 1-hot observations is orders of magnitude smaller than the number used by W-ORBS. Similarly, the 2-hot deletion generation scheme involves only 18.6% as many observations as used by W-ORBS. T-ORBS tends to use fewer observations than W-ORBS, and thus, the values are closer. Overall, the number of 1-hot observation is 1.7% of the number of T-ORBS observations, while the number of 2-hot observations is 79.7% of the number of observations used by T-ORBS.

For a more complete efficiency comparison, we also consider efficiency in the worst-case scenario of using MOAD; if one wants only a single slice from a criterion in the program. In such a case, ORBS could very well be more efficient than MOAD which needs to analyze the whole program dependence. To investigate this, we computed the ratio between how many observations MOAD needs and the average number of observations needed to generate a single slice by ORBS. However, our results show that in this the worst-case scenario, MOAD with a 1-hot deletion generation scheme actually needs only 13.9% and 58.7% of the observations needed by W-ORBS and T-ORBS, respectively. For the 2-hot deletion generation scheme, the number of observations used by MOAD is the same as the number needed for W-ORBS and T-ORBS to generate approximately 14 slices and 59 slices, respectively.

5.3.2. Slice size

Due to the approximate nature of the inference, MOAD is expected to generate larger slices than W-ORBS or T-ORBS. Table 3 shows the average slice size, $\mu\,(WS_c)$ and $\mu\,(TS_c)$, over all slicing criteria from W-ORBS and T-ORBS, along with $\mu\,(MS_c)$. Compared to W-ORBS, the results find that MOAD produces slices that are, on average, 40% larger than those produced by W-ORBS. Considering the Once Success model built using the 2-hot observations, which is the setting for generating the smallest slice, the inferred slice's size is 12% larger than the W-ORBS slices on average. The most significant difference occurs with the program wc when using the Logistic inference model built 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, using two-hot data, the Once Success model inferred slices that have 98%, 98%, and 92% of W-ORBS slice size for replace, prttok, prttok2, and the Logistic model inferred slices that have 97% and 91% of W-ORBS slice size for prttok and totinfo, 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 30% larger than T-ORBS. With the Once Success model with 2-hot observations, the inferred slices are 4% larger than T-ORBS slices. 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.

5.3.3. Accuracy

While ORBS guarantees to provide a 1-minimal slice that preserves the trajectory of the slicing criterion, MOAD approximates the slice given the slicing criteria. Therefore, we set ORBS slices as ground truth and compare MOAD slices to those. For each subject, deletion generation scheme, and inference model, we calculate excess and miss (defined in Section 4.5), to facilitate a more detailed comparison of studied models. The values shown in Table 5 are averages over all slicing criteria in each of the subject programs; 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.

As the number of observations increases from 1-hot to 2-hot, the value of miss for Once Success ($\mathbb{O}$) significantly decreases, while excess slightly increases. This tendency is repeated for the Logistic models ($\mathbb{L}$). However, miss barely changes for the Bayesian models ($\mathbb{B}$) which helps explain why the size of $\mathbb{B}$ slices shows little change in Table 3.

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 compared 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 slices more similar to those produced by W-ORBS and T-ORBS. Table 6 illustrates this by showing 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, $\mathbb{O}$ performs consistently and notably better than the other two models. Interestingly this dominance does not extend to excess where the values are smaller and more varied.

To gain confidence, we applied an ANOVA and then Tukey's HSD test (Tukey, 1970) 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 the Once Success model built with the 2-hot observations generates slices that are not only

[1] The full details of Tukey's HSD results are available online at: https://coinse.github.io/MOAD_Rdata_webpage/.

Table 5
Average value of excess, denoted 'E', and miss, denoted 'M', in MS_c when compared to WS_c and TS_c. Columns 3–8 compare with WS_c at the line level, Columns 9–14 compare with TS_c at the line level, and finally Columns 15–20 compare with TS_c at the statement (unit) level. The values in parentheses reflect the percentage of excess and miss compared to the number of lines or units in the original program.

Subject	Deletion generation scheme	MS_C vs. WS_C (line)						MS_C vs. TS_C (line)						MS_C vs. TS_C (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 6
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.

miss	MS_c vs. WS_c (line)		MS_c vs. TS_c (line)		MS_c vs. TS_c (stmt)	
	$\mathbb{L}-\mathbb{O}$	$\mathbb{B}-\mathbb{O}$	$\mathbb{L}-\mathbb{O}$	$\mathbb{B}-\mathbb{O}$	$\mathbb{L}-\mathbb{O}$	$\mathbb{B}-\mathbb{O}$
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_c vs. WS_c (line)		MS_c vs. TS_c (line)		MS_c vs. TS_c (stmt)	
	$\mathbb{L}-\mathbb{O}$	$\mathbb{B}-\mathbb{O}$	$\mathbb{L}-\mathbb{O}$	$\mathbb{B}-\mathbb{O}$	$\mathbb{L}-\mathbb{O}$	$\mathbb{B}-\mathbb{O}$
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 7
Average value of miss and excess in MOAD backward slice when compared to CodeSurfer backward slice. $|L|$ shows the size of the number of lines in comparison. The values in parentheses reflect the percentage of excess and miss compared to $|L|$.

Subject	$\|L\|$	μ (MS_c)	μ (CS_c)	miss (%)	excess (%)	Subject	$\|L\|$	μ (MS_c)	μ (CS_c)	miss (%)	excess (%)
mbe	13	41.3%	45.2%	0 (2)	1 (6)	replace	275	35.5%	45.4%	24 (9)	51 (19)
mug	15	45.9%	31.1%	2 (16)	0 (1)	sched	163	42.3%	45.7%	24 (15)	30 (18)
wc	17	33.6%	20.6%	3 (17)	1 (4)	sched2	148	35.7%	46.8%	11 (7)	27 (19)
prttok	278	30.9%	54.8%	13 (5)	79 (29)	totinfo	157	36.7%	41.5%	12 (7)	19 (12)
prttok2	171	35.4%	40.4%	18 (10)	27 (16)	tcas	73	37.1%	38.9%	8 (11)	9 (12)
						Average		37.4%	41.0%	11 (10)	24 (14)

compact but also the most similar to ORBS slices. However, there are some cases where the Logistic model produces the smallest slice, such as when slicing totinfo with the 2-hot data. However, such slices tend to have a high excess. Since an ORBS slice for a given slicing criteria is not necessary unique, we explicitly check the output trajectory of such small slices. Further investigation revealed that they are unable to preserve the trajectory of the targeting slicing criteria, which implies that the dependency inference was not sufficiently precise.

As we described in Section 4.2.1, T-ORBS used in this study is a modified version of T-ORBS in Binkley et al. (2019) to stop its traversing at the statement-level. The modified version of T-ORBS shows a dramatic speed-up compared to the original T-ORBS. Thus, our answer to RQ3 only considers the comparison with W-ORBS. From the overall trends observed in size and similarity comparisons, we answer **RQ3** as follows:

> **RQ3. Performance compared to ORBS:** Using Once Success built with the 2-hot data, MOAD can require less than one fifth of the observations compare to W-ORBS. At the same time the inferred slice is only 12% larger than the W-ORBS slice. The result further presents that Once Success also produces slices of high accuracy: it achieved much smaller miss, while there is no big difference in excess compared to other inference models.

5.4. In-depth comparison to static backward slicing (RQ4)

To answer **RQ4**, we configure MOAD to use the 2-hot deletion generation scheme and the Once Success inference algorithm because the results from RQ2 show that this configuration is the most effective and accurate configuration. As described in Section 4.5, we convert the resulting slices to the line-level.

Table 7 shows the miss and excess values comparing the line-level backward slices from MOAD and CodeSurfer averaged over the set of criteria. $|L|$ shows the size of the number of lines in comparison. Note that $|L|$ is not the same with SLoC in

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int p(int j);
4  int q(int k);
5  int f1(int k);
6  int f2(int k);
7  int f3(int j);
8  int main(int argc, char *argv[])
9  {
10   int j;
11   int k;
12   j = (int) strtol(argv[1], NULL, 10);
13   k = (int) strtol(argv[2], NULL, 10);
14
15   while (p(j))
16   {
17       if (q(k))
18       {
19           k = f1(k);
20       }
21       else
22       {
23           k = f2(k);
24           j = f3(j);
25       }
26   }
27   printf("%d\n", j);
28 }
```

Listing 1: mbe

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  bool p(int i);
5  bool q(int c);
6  int h(int i);
7  int f();
8  int g();
9  int main(int argc, char *argv[])
10 {
11   int i;
12   int c;
13   int x;
14   i = atoi(argv[1]);
15   c = atoi(argv[2]);
16   x = atoi(argv[3]);
17   while (p(i))
18   {
19       if (q(c))
20       {
21           x = f();
22           c = g();
23       }
24       i = h(i);
25   }
26   printf("%d\n", x);
27 }
```

Listing 2: mug

```
1  #include <stdio.h>
2  int isletter(char c);
3  int main(int argc, char *argv[])
4  {
5    int characters;
6    int lines;
7    int inword;
8    int words;
9    char c;
10   characters = 0;
11   inword = 0;
12   words = 0;
13   while (scanf("%c", &c) == 1)
14   {
15       characters = characters + 1;
16       if (c == '\n')
17       {
18           lines = lines + 1;
19       }
20       if (isletter(c))
21       {
22           if (inword == 0)
23           {
24               words = words + 1;
25               inword = 1;
26           }
27       }
28       else
29       {
30           inword = 0;
31       }
32   }
33 }
34 int isletter(char c)
35 {
36   printf("%c ", c);
37   if (((c >= 'A') && (c <= 'Z'))
38       || ((c >= 'a') && (c <= 'z')))
39   {
40       return 1;
41   }
42   else
43   {
44       return 0;
45   }
46 }
```

Listing 3: wc

Fig. 3. mbe, mug, and wc source code.

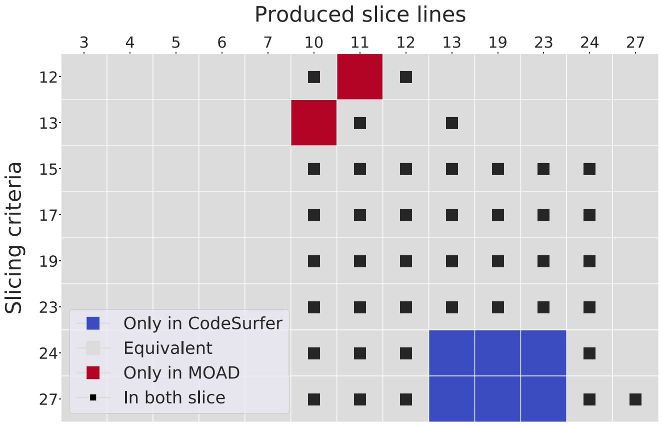


Fig. 4. Difference of the backward slices of MOAD and CodeSurfer of *mbe*.

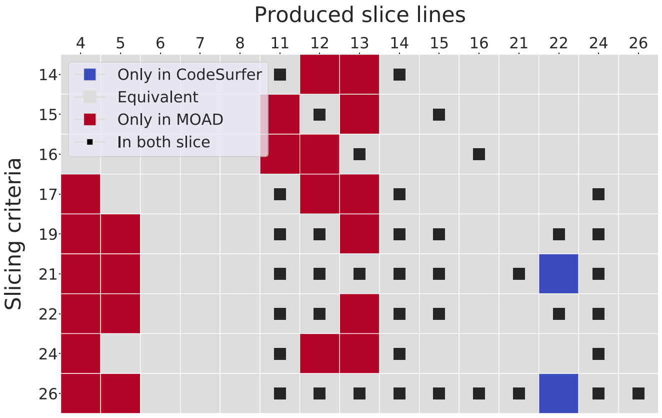


Fig. 5. Difference of the backward slices of MOAD and CodeSurfer of *mug*.

Table 1 since candidate lines are the lines that correspond to non-containing statements. $\mu\,(MS_c)$ and $\mu\,(CS_c)$, in this case, use $|L|$ as a denominator of the ratio. From the table, when slicing mbe there is no line that MOAD fails to delete that CodeSurfer deletes and thus miss is zero. On average, 2 and 3 lines are missed by MOAD in mug and wc and 8–24 lines are missed by MOAD for the programs in Siemens suite. In terms of excess lines, on average only 0 to 1 lines are errantly deleted by MOAD in the backward slices of mbe, mug, and wc when compared with slices generated by CodeSurfer. For the Siemens suite, MOAD excessively deletes 9–79 lines, which is a notable higher percentage. In the big picture there is a larger proportion of missed lines with the bigger programs because they include more opportunities for static analysis false-positives. We consider this issue in greater detail during the discussion of **RQ6**.

Fig. 3 shows the source code for the three benchmark C programs mbe, mug, and wc. For these three, Figs. 4, 5, and 6 visualize the differences between the slices produced by MOAD and those produced by CodeSurfer. In each figure

- The *x*-axis shows the line numbers of the lines whose source code is affected by the slicing criteria.
- The *y*-axis shows the program line numbers of the lines that make up the slicing criteria.
- Each row captures the backward slice taken with respect to the criterion found on the *y*-axis.
- A blue cell, 🟦, denotes an excess statement (one that is in the CodeSurfer slice but not the MOAD slice).
- A red cell, 🟥, denotes a missed statement (one that is in the MOAD slice but not the CodeSurfer slice).
- An empty gray cell, ⬜, denotes a statement in neither slice.

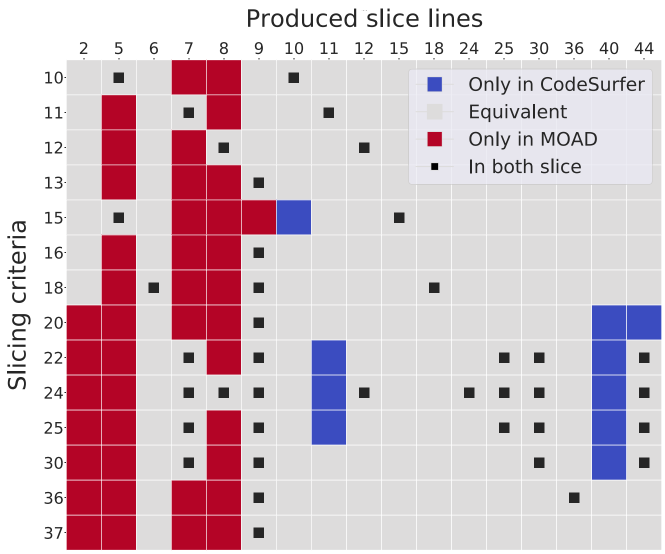


Fig. 6. Difference of the backward slices of MOAD and CodeSurfer of *wc*.

- A gray cell with a black square, ▪, denotes a statement in both slices.

The remainder of this section explores the differences between the slices produced by CodeSurfer and MOAD with an eye toward understanding their root causes. We addressed every mismatch cases and these differences can be partitioned into the following five categories. After explaining each, we illustrate it with examples taken from the three benchmark programs. The examples described by line numbers in the program represent all red or blue cells in the columns of referred lines in the figures.

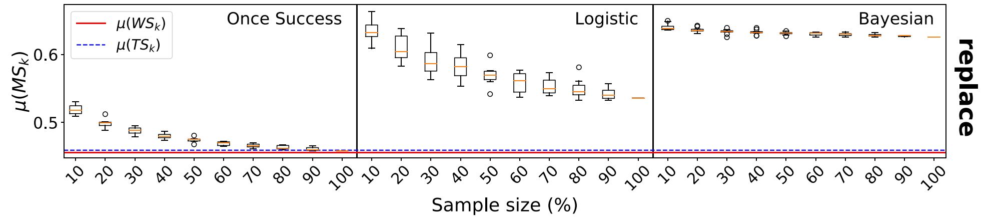
- 🟥 **[Keeping Declarations]**:
  Using the Once Success inference model, MOAD often cannot delete variable declarations because there is insufficient observation of their deletion. This occurs because the 2-hot deletion generation scheme deletes no more than two statements at a time. Therefore, if a variable has more than one use, Once Success will never observe the successful deletion of the variable declaration. Doing so requires deleting at least three statements.
  MOAD is also unable to delete a function prototype if its deletion produces a compilation error. In contrast, CodeSurfer computes dependence-closure slices using a graph representation that omits function prototypes; thus, it will never include one in a slice (Binkley, 1993).
  Examples of this category include declaration statements on Line 10 and 11 from mbe (Fig. 4); function declarations on Lines 4 and 5, and declaration statements on Line 11, 12, and 13 from mug (Fig. 5); and finally the function declaration on Line 2 and declaration statements on Lines 5, 7, and 8 from wc (Fig. 6).
- 🟥 **[Mutation Granularity]**:
  The only program mutation operator used by MOAD is statement deletion. This limits the granularity at which it operates. For example, consider the statement iterations++ nested within the loop while (scanf("%c", &c) == 1). Key to this example is that the iteration count does not depend on the value of c, which CodeSurfer correctly identifies. However, MOAD treats the loop statement as atomic, and thus is forced to retain c and consequently its declaration.
  The only example of this category is the declaration statement on Line 9 from wc (Fig. 6).
- 🟦 **[Missing Initialization]**:
  There are cases where the deletion of a variable initialization has no impact (e.g., in Java integer variables are implicitly initialized to zero). In such cases, MOAD will delete an initialization statement, while CodeSurfer cannot (assuming that the initialized value is used in a subsequent computation).
  Examples of this category include assignment statements on Lines 10 and 11 from wc (Fig. 6).
- 🟦 **[Missing Return]**:
  While some languages mandate the presence of a return statement, languages such as C do not. When a return statement is deleted from a program in a permitting language, the function simply returns the last value placed in the return location (e.g., the last value placed in the eax register). If this value happens to cause the same behavior (e.g., the return value is used as a boolean where it is expected to be true in normal execution), then MOAD can often delete the return statement without causing the observable semantics of the program to change.
  Examples of this category include return statements on Lines 40 and 44 from wc (Fig. 6).
- 🟦 **[Limits of Static Analysis]**:
  While the static analysis of wc is straight forward, both mug and mbe were designed to reveal the limitations of static analysis. A previous study (Binkley et al., 2015) showed that the observation-based slicing on which MOAD is based, can successfully untangle the complex data dependence found in these examples, and thus overcome the limitations exhibited by static analysis. MOAD too learns to untangle these complex data dependences and, in doing so, produces a more precise slice.
  Examples of this category include assignment statements to the variable k on Lines 13, 19, 23 from mbe (Fig. 4); and an assignment statement on Line 22 from mug (Fig. 5).

Our in-depth comparison between MOAD's backward slice and static backward slice could answer **RQ4** as follows:

> **RQ4. In-depth Comparison to Static Backward Slicing:** On average, MOAD misses 0–3 lines in mbe, mug, wc, and 8–24 lines in Siemens suite programs compared to CodeSurfer slices. We could identify two root causes of where MOAD missed deleting the statement from the slice compared to the static backward slice: **Keeping Declarations**, due to insufficient observation of the deletions, and **Mutation Granularity**, where statement deletion is not enough to observe the independence.
> On average, MOAD excessively deletes 0–1 lines in mbe, mug, wc, and 9–79 lines in Siemens suite programs compared to CodeSurfer slices. We could identify three root causes of MOAD excessively deleting statements from the slice compared to the static backward slice: **Missing Initialization** and **Missing Return**, where the programming language's implicit semantics makes no change, and **Limits of Static Analysis**, when static analysis fails to identify the true dependence in the program.

5.5. Sampling effect (RQ5)

RQ5 considers the tradeoff between the amount of training data used and the quality of the inference. To gain an initial impression for the data, Fig. 7 shows two example programs, replace and wc. In each of the resulting six plots, the *x*-axis shows the sample size, while the *y*-axis shows the ratio of the average slice size (averaged over all slicing criteria in the program) to



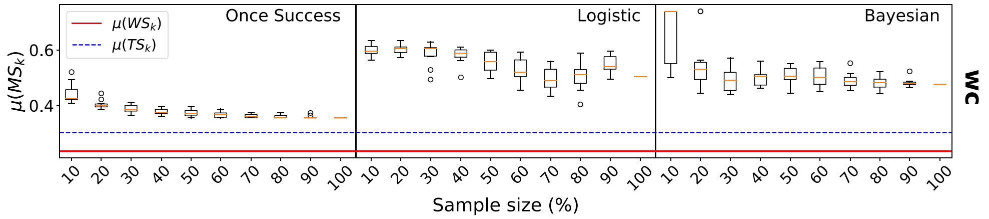
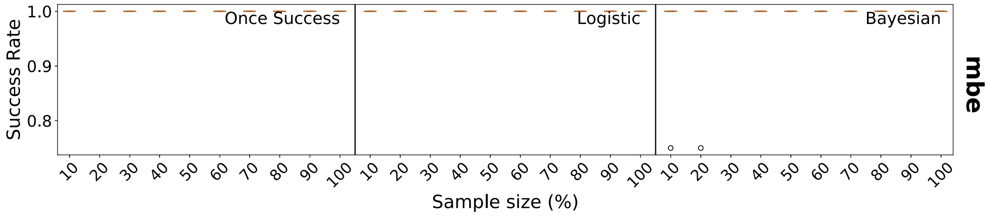


Fig. 7. Averaged over all slicing criteria for replace (top) and wc (bottom), the figure presents the mean slice size of MOAD, $\mu(MS_c)$, given as a percentage of the size of the original program for a range of 2-hot sample sizes (shown on the x-axis). Each boxplot summarizes the performance of a model built using ten different random samples. The red and blue lines represent the means for W-ORBS and T-ORBS slice size, averaged over all slicing criteria in the program.



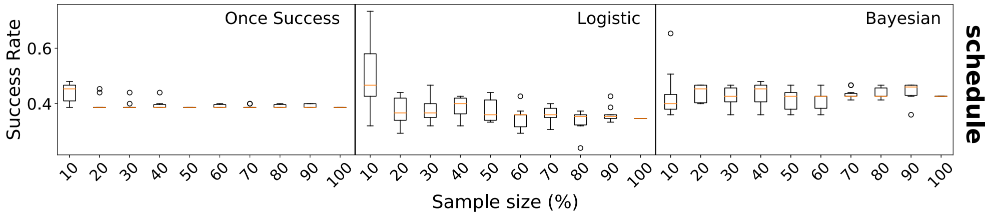


Fig. 8. The figure presents MOAD's mean success rate averaged over all slicing criteria for mbe (top) and sched (bottom). Each boxplot summarizes the performance of a model built using ten different random samples.

the original program size. Finally, the solid red (gray) and blue (dashed) horizontal lines represent the average slice size ratio produced by W-ORBS and T-ORBS, respectively. Overall, the six plots support the notion that as sample size increases, the slice size of MOAD decreases and in some cases approaches that of the two ORBS slicers.

Because the three different models show a fair amount of variation, we consider each model separately. To begin with, for the Once Success model $\mathbb{O}$ (the left two plots) there is a substantial initial drop off that lessens, but continues to fall, as the sample size increases. For example, the average of the first three slice-size differences (i.e., 20%–10%, 30%–20%, and 40%–30%) is 4.4 times larger than that of the last three slice size differences (i.e., 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 individual samples (the height of the boxes) is relatively small, which suggests that $\mathbb{O}$ is robust against the stochastic sampling.

For the Logistic model, $\mathbb{L}$, the size of the slices also tends to decrease as the sample size increases, but the trend is not as strong as with $\mathbb{O}$. The $\mathbb{L}$ model also shows higher variance across samplings (taller boxes), when compared to other inference models. Finally, while the Bayesian model, $\mathbb{B}$, tends to generate smaller slices with more observations, the trend is not strong and the median size fluctuates. Furthermore the difference in slice size between samples is relatively small (especially for replace).

To gain additional confidence, we applied ANOVA separately to all the data for each model.[2] In all three cases, the results are statistically significant ($p < 0.0001$) thus we applied Tukey's post-hoc test. For the $\mathbb{O}$ models Tukey's post-hoc test finds five equivalence classes of mean slice size. The most useful findings are that using samples of 40% to 90% of the 2-hot data produces mean slice sizes that are not statistically different. The same is true of the range from 50% to 100%, suggesting that using only half of the data produces results essentially indistinguishable from using all of the data.

The $\mathbb{L}$ models also have five equivalence classes. Notable among these, there is a band from 30% to 80%, and two narrower bands from 60% to 90% and 80% to 100%. Narrower bands reflects the models being more sensitive to the amount of data that they are built on.

Finally, the Bayesian models, $\mathbb{B}$, show the greatest stability with all values from 20% to 100% being in the same band. Thus only when using a 10% sample (the only other band) does the model show inferior performance. This suggests that these models themselves are very robust against sampling variation. If it

[2] The full details of ANOVA results are available online at: https://coinse.github.io/MOAD_Rdata_webpage.

were possible to improve the size and the accuracy of slices produced by $\mathbb{B}$, the stability observed here would be a strong reason for choosing these models.

We also investigate how the success rate changes as the number of observations increases. Fig. 8 shows their tendency for two example programs, mbe, and sched. Similar to Fig. 7, the x-axis shows the sample size, while the y-axis shows the success rate averaged over all slicing criteria in the program. As we discussed in Section 5.1, MOAD can easily generate trajectory-preserving slices for the small programs mbe, mug, and wc. This pattern persists even when using only a fraction of the observations from the 2-hot deletion generation scheme. For the Siemens suite programs, the success rate tends to decrease as the sample size grows for the Once Success model and the Logistic model. In line with Fig. 7, as the sample size increases more statements are deleted, which tends to cause the remaining code to be harder to compile and less likely to preserve the trajectory. The robustness of the slice size from the Bayesian model keeps the success rate steady despite the sample size growth.

RQ5 considers the impact of sampling effect. In particular, how viable is a model built using only a subset of the 2-hot data? We answer **RQ5** as follows.

RQ5. Sampling effect: The high initial rate of reduction and higher variability of the inferred slice size for small sample sizes indicates there is some sampling effect especially with Once Success. On the other hand, the wide bands of indistinguishable performance show that it is possible to build high performing models using only a fraction of the training data. For example, Once Success infers slices only 3.4% larger when using as little as half of the training data, while the Bayesian model has similar performance using only 20% of the training data. In summary it is viable to use only a subset of the data when building the models. The success rate related to the sampling effect tends to follow the slice size: it often decreases as more statements are deleted.

5.6. In-depth comparison to static forward slicing (RQ6)

RQ6 investigates how well MOAD can produce forward slices, despite its having been designed with backward slices in mind. As described in Section 3.4 this might be seen as a bit of a surprise because doing so with ORBS, which was also designed with backward slicing in mind, turns out to be quite difficult. However, that MOAD is easier to 'reverse' does not say anything about the quality of the resulting slices. To address this question we compare the forward slices of MOAD with those produced by CodeSurfer.

Similar to **RQ4**, we compare the slices at the line-level. Recall that for a MOAD forward slice we invert the sense of the criteria and the set of statements affecting the criteria. Thus the slicer starts with a set of affecting statements and produces as the slice a set of criteria. Given one or more line numbers from a program, MOAD computes a forward slice by first mapping those lines to one or more affecting statements within the program using the *statement-line* mapping described in Section 4.5. It then infers a set of criteria. Finally MOAD maps those criteria to the line numbers of the slice using the same mapping.

For CodeSurfer the criteria of the forward slice are the same as those of a backward slice, a set of dependence-graph vertices. Given one or more line numbers from a program, CodeSurfer computes a forward slice by first identifying the set of vertices representing code on the given line numbers. It then traverses dependence edges in the dependence graph *forward* (the origin of the name *forward slicing*) to identify the set of vertices that make up the slice. Finally, it maps this vertex set to a set of line numbers, which forms the final slice.

Table 8 shows the values of miss and excess comparing the forward slices of MOAD and CodeSurfer averaged over the set of criteria for each program. It is important to note that in this comparison we limit the set of lines potentially in a slice to those that contain one or more variables because these lines are those for which there exist a slicing criterion and thus they are the only lines that MOAD can report as being part of, or absent from, a forward slice. This set of lines is referred to as L and is used for a denominator calculating $\mu\,(MS_c)$ and $\mu\,(CS_c)$ in Table 8. Note that this set is different from the set named L in Table 7. For six of ten subjects MOAD has missed no lines that should have been deleted when compared to CodeSurfer. For the remaining four subject only 1–6 lines are missed. A single line has been excessively deleted by MOAD compared to CodeSurfer for mbe, and there is no excess lines deleted for mug and wc. For the program in Siemens suite, 7–37 lines have been excessively deleted by MOAD compared to forward slices generated by CodeSurfer.

Fig. 9 shows the difference between the forward slices of MOAD and CodeSurfer for prttok, visualized in the same way as in Section 5.4. We can identify structural or semantic relationships between the lines that have similar patterns in their forward slices (i.e., rows in the figure). For example, Lines 129–136 of prttok are all from the same case in a switch statement. As a second example, Lines 185, 203, and 273 are lines that call the function `unget_char`. Similar patterns can be observed in graphs of the other programs we well.

To further investigate the efficiency of MOAD, we statistically analyze the number of excessively deleted lines and the ratio of such lines to the size of CodeSurfer forward slices, which is shown in Table 9. The first and second rows show the total and average number of excessively deleted lines over all slices. As can be verified using Fig. 9, the total for prttok is the number of blue cells (🟦). The average is simply excess. The third and fourth rows show the average and standard deviation of the ratio of this value to the CodeSurfer slice size.

From the table, MOAD reduces the average forward slice size by at most one line for three benchmark programs (mbe, mug, wc) and by 6.82–36.75 lines for the programs of the Siemens suite. Turning to the ratio, the reduction is larger for the Siemens suite (25%–45%) than for the mbe, mug, and wc (7%–16%). The main cause for this greater reduction is that MOAD can determine *false positive* dependences existing in the static analysis by directly observing the program execution. Fig. 10 shows the distribution of the ratio of the number of lines in each CodeSurfer slice only to the size of the CodeSurfer slice for the programs in the Siemens suite. The plot omits the three benchmarks because they lack sufficient slices to be meaningfully represented in a boxplot.

The dots near 100% of the ratio in Fig. 10 show the case for which MOAD eliminates almost every line in the CodeSurfer forward slice from its slice. We investigate what kind of criteria make that happen and present some of the cases of them here.

The first snippet of Listing 4 in Fig. 11 shows the snippet of the source of prttok. The value of `token_id` in struct `token_ptr` determines whether the main loop of the program terminates or not. Therefore, CodeSurfer includes most of the line in the program to its forward slice of Line 147, which assigns a value to `token_id` that makes the main loop to continue. However, when both Line 147 and Line 149 are deleted, *fall-through* happens, which assigns another value to `token_id` in Line 152, keeping the main loop to continue. It is important to note that the test suite we used contains not only the test cases handling the strings with special characters, but also ones without them. From such the observation, MOAD learns that, except those actually related to the handling of special characters, lines in CodeSurfer's forward slice of Line 147 are not affected by Line 147 if the main loop does

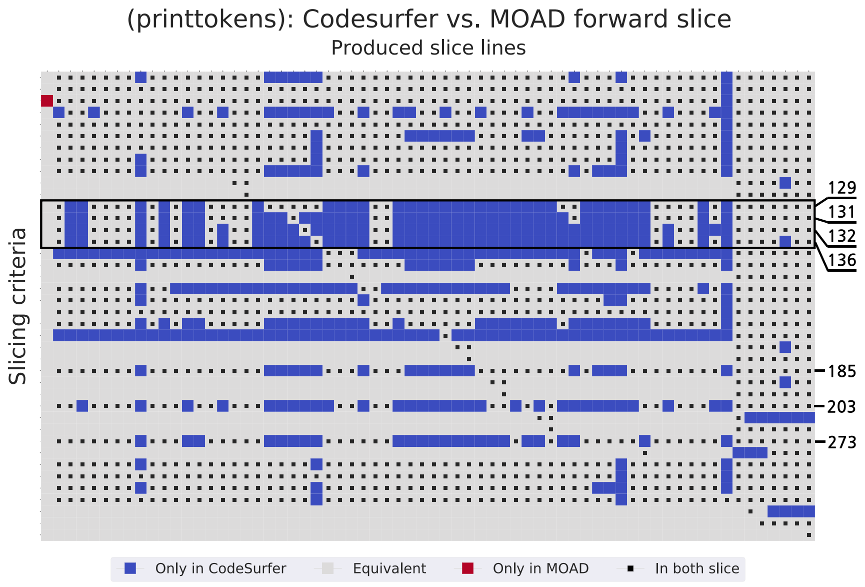


Fig. 9. Difference of the forward slices of MOAD and CodeSurfer of *printtokens*.

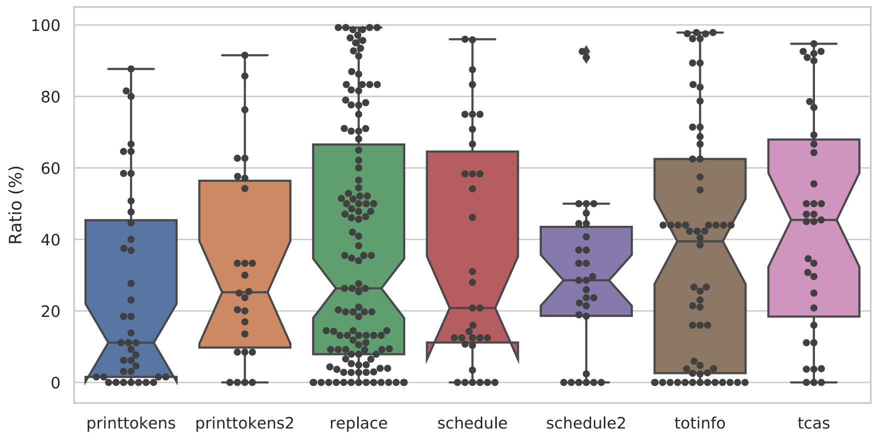


Fig. 10. Distribution of the ratio of the number of lines only in a CodeSurfer slice to the CodeSurfer slice size.

```
143 ...
144     case 24:
145     case 25:
146     case 32:
147       token_ptr->token_id = special(next_st);
148       token_ptr->token_string[0] = '\0';
149       return (token_ptr);
150     case 27:
151     case 29:
152       token_ptr->token_id = constant(next_st,
           token_str, token_ind);
153       get_actual_token(token_str, token_ind);
154       strcpy(token_ptr->token_string, token_str);
155       return (token_ptr);
156     case 30:
157 ...
```

Listing 4: Snippets from prttok

```
175 ...
176   if (check_delimiter(ch) != TRUE) {
177   ...
178     unget_char(ch, tstream_ptr->ch_stream);
179     token_ind--;
180     get_actual_token(token_str, token_ind);
181     strcpy(token_ptr->token_string, token_str);
182     return (token_ptr);
183   }
184   token_ptr->token_id = NUMERIC;
185   unget_char(ch, tstream_ptr->ch_stream);
186   token_ind--;
187   get_actual_token(token_str, token_ind);
188   strcpy(token_ptr->token_string, token_str);
189   return (token_ptr);
190 }
191 ...
```

Listing 5: Snippets from prttok

```
129 ...
130     token get_token(tp)
131     token_stream tp;
132     {
133       ...
134       if (id == 1) {
135         i++;
136         buffer[i] = ch;
137         return (buffer);
138       }...
139     }
140 ...
```

Listing 6: A snippet from prttok2

```
258 ...
259     int get_process(prio, ratio, job)
260     {
261       ...
262       index = ratio * length;
263       index = index >= length ? length -1 : index;
264       for (next = &prio_queue[prio].head; index && *next;
           index--) {
265         next = &(*next)->next;
266       }...
267     }
268 ...
```

Listing 7: A snippet from sched2

Fig. 11. Snippets of the source codes of Siemens suite.

Table 8
Average value of miss and excess in MOAD forward slice when compared to CodeSurfer forward slice. $|L|$ shows the size of the number of lines in comparison. The values in parentheses reflect the percentage of excess and miss compared to $|L|$.

Subject	$\|L\|$	μ (MS_c)	μ (CS_c)	miss (%)	excess (%)	Subject	$\|L\|$	μ (MS_c)	μ (CS_c)	miss (%)	excess (%)
mbe	8	56.2%	68.8%	0 (0)	1 (13)	replace	169	29.7%	47.6%	6 (4)	37 (22)
mug	9	39.7%	42.9%	0 (0)	0 (3)	sched	49	38.5%	43.1%	6 (12)	8 (17)
wc	14	11.9%	15.1%	0 (0)	0 (3)	sched2	52	30.3%	40.6%	1 (3)	7 (13)
prttok	66	50.2%	70.1%	0 (0)	13 (20)	totinfo	93	28.0%	40.4%	0 (0)	12 (12)
prttok2	61	50.9%	66.6%	4 (6)	13 (22)	tcas	42	22.6%	41.5%	0 (0)	8 (19)
						Average		35.8%	47.7%	2 (2)	10 (14)

Table 9
Statistics on the number of lines only in CodeSurfer slice.

		mbe	mug	wc	prttok	prttok2	replace	sched	sched2	totinfo	tcas
Count	Sum	6	7	9	40	26	123	35	30	62	35
	Average (excess)	1.00	0.29	0.44	13.12	13.12	36.75	8.11	6.83	11.53	7.94
Ratio	Average	16%	7%	14%	25%	33%	38%	36%	30%	37%	45%
	stdev	17%	19%	28%	27%	27%	33%	33%	24%	34%	31%

not stop. Deleting all lines that are not related to the handling of special characters from Line 147's forward slice by MOAD illustrates its success at *inferring* accurate dynamic dependence analysis.

A similar pattern is observed in Line 178 of Listing 5 in Fig. 11. CodeSurfer includes most of the lines in prttok in its forward slice of Line 178, because `tstream_ptr` is the buffer containing the input to be processed. When both Line 178 and Line 182 are deleted, the execution does not return at Line 182 but keeps executing the lines after Line 183, which eventually executes Line 185 that does the same thing with Line 178. Therefore, by removing Line 178 instead of Line 185, the lines that are sufficient to maintain their functionality are excluded from Line 178's forward slice by MOAD.

CodeSurfer includes almost all lines of prttok2 to its forward slice of Line 135 (shown in Listing 6 in in Fig. 11). CodeSurfer makes this decision because the value in the memory `buffer` returned at Line 137 determines whether the main loop of the program terminates; if it has an *End of file (EOF)* token, the program terminates. However, what the `if` statement from Line 134 to Line 138 actually does is to put the double quote at the end of the token if it starts with a double quote. Thus, `buffer` in Line 136 already has a value that keeps the main loop running, which indicates that Line 135 is irrelevant to the main loop. Consequently, MOAD removes the lines from its forward slice of Line 135 that are accidentally included in the slice generated by CodeSurfer.

For Line 263 of sched2 (shown in Listing 7 in Fig. 11), the size of the forward slices generated by MOAD is much less than that of the forward slices generated by CodeSurfer. Line 263 handles the case when the desired index is larger than the size of the queue. However, there is no test case that results in a value of index that is larger than the size of the queue. Consequently, MOAD deletes those unobservable dependences from Line 263's forward slice. Such test suite specific information can only be captured by MOAD.

We further analyze the root cause for the lines found only in slices generated by MOAD, which are seen as red cells (🟥) in the figure. Unlike the case of backward slices, declaration statements are not the major cause of difference: declaration statements are not in the slice since they do not contain any slicing criteria. Instead, the major cause for the red cells is *segmentation faults*. Such faults result in an empty log file, which the tool interprets as a change in the trajectories of all executed criteria. These perceived changes prevent MOAD from deleting statements from a slice. For example, in Fig. 12, removing Line 99 of prttok leads to a segmentation fault; thus when MOAD computes the forward slice taken with respect to Line 99 it includes Line 24 (the 🟥 in Fig. 9) even though the criterion on Line 24 is not actually affected by the statement at Line 99.

Another significant root cause is *hidden dependency*, which static analysis is incapable of capturing. This occurs, for example, between a statement updating a database and another querying the database, where the result of the query depends on the update. In our case, consider Fig. 13, which shows a section of code taken from sched. In this code, Lines 256, 260, 261, and 270 read input from the external source. In the configuration used for our empirical evaluation, CodeSurfer assumes that reads from a file are independent from each other: while it can be configured to assume all reads are dependent, such a setting will dramatically increase slice sizes, and is often not what the user wants.

Based on the comparison result, we could answer **RQ6** as follows:

> **RQ6. In-depth Comparison to Static Forward Slicing:** In comparison to the static forward slice, MOAD only misses deleting 0–6 lines from the forward slice, while it could excessively delete 0–37 lines from the slice. A large and excessive deletion compared to the static forward slice often happens to MOAD's forward slice when the criterion is related to (1) the main loop termination, (2) the ground data to process, or (3) the corner case handling. The root cause for the lines found only in MOAD's slice is often the *segmentation faults*.

6. Discussions and future work

6.1. Advanced and adaptive deletion generation scheme

An in-depth analysis of MOAD backward slices compared to static backward slices indicates that a specific set of units should be deleted together or should not be deleted. The category 🟥 **[Keeping Declarations]** describes that MOAD often cannot delete the variable declaration statements since the 2-hot deletion generation scheme does not offer an observation that deletes the declaration statement with all the statements using the declared variable. 🟦 **[Missing Initialization]** and **[Missing Return]** categories produce evidence that unexpected execution behavior could mislead MOAD to excessively delete the initialization statements or return statements from the slice. Such a result suggests that a more advanced and adaptive deletion generation scheme is needed for modeling the source code.

A more principled way would be to use Design of Experiments (Chaloner and Verdinelli, 1995) to systematically select among

```
main(argc, argv)
int argc;
char *argv[];
{
  token token_ptr;
  token_stream stream_ptr;
  if (argc > 2) {
    fprintf(stdout, "The format is print_tokens filename(optional)\n");
```

```
token_stream tstream_ptr;
{
...
  cu_state = token_ind = token_found = 0;
  while (!token_found) {
```

Fig. 12. An example of false positive dependency due to the *segmentation fault*.

```
...
    case UNBLOCK:
        fscanf(stdin, "%f", &ratio);
        unblock_process(ratio);
        break;
    case UPGRADE_PRIO:
        fscanf(stdin, "%d", &prio);
        fscanf(stdin, "%f", &ratio);
        if ((prio > MAXPRIO) || (prio <= 0)) {
            fprintf(stdout, "** invalid priority\n");
            return;
        } else {
            upgrade_process_prio(prio, ratio);
        }
        break;
    case NEW_JOB:
        fscanf(stdin, "%d", &prio);
...
```

Fig. 13. An example of the hidden dependence in sched that static analysis is unable to figure out.

the many possible deletions. This could also allow and utilize the sampling of more than two deletions per build and observe cycle, which could provide faster learning as well as enough observations for deleting declarations. Yet another approach would be to generate the deletion just-in-time (adaptively) with respect to the observations that have been made (e.g., via active learning (Zuluaga et al., 2013)). Using the observation data, an interim model might be learned that can propose future deletions with the highest potential information gain.

While the aforementioned methodologies work in general, we could also think of a more particular method to adopt the deletion generation scheme for a specific limitation. One possible way of overcoming 🟥 **[Keeping Declarations]** limitation is to introduce *meta*-statements, which group together all statements that use a particular variable. In addition to deleting individual statements, MOAD could also attempt to delete an entire group by deleting its single meta-statement. In several of the examples, this would improve the slice generated by MOAD.

A variation of the meta-statement approach attempts to observe more detailed dependence relationships between the statements found in each meta-statement. For example, consider using the n-hot deletion generation scheme. Let meta-statement S include the statements $\{s_1, \ldots, s_m\}$. The n-hot deletion generation scheme covers the observations of deleting every combination of i statements ($1 \leq i \leq n$) in S. Hence, we additionally try to delete combinations that are not covered by the n-hot deletion generation scheme, which are the combinations of j statements in S where $n+1 \leq j \leq m$. For example, assume we are using the 2-hot deletion generation scheme with the following program:

```
int a;        (s1)
...
a = 3;        (s2)
...
c = 7 + a;    (s3)
d = g(a);     (s4)
```

where s_1, s_2, s_3, and, s_4 belongs to the same meta-statement. Since every single statement deletions ($\{s_1\}, \cdots, \{s_4\}$) and deletions of a pair of statements ($\{s_1, s_2\}, \{s_1, s_3\}, \cdots, \{s_3, s_4\}$) are already covered by 2-hot deletion generation scheme, we additionally observe deletions of other combinations with the statements such as $\{s_1, s_2, s_3\}$, $\{s_1, s_2, s_4\}$, $\{s_1, s_3, s_4\}$, $\{s_2, s_3, s_4\}$, and $\{s_1, s_2, s_3, s_4\}$. While the additional cost could be much larger compared to the problem we want to solve, such a systematic approach may introduce various interesting observations.

For 🟦 **[Missing Initialization]** and **[Missing Return]**, we might simply mark initialization statements or return statements as *undeletable* during the observation phase. While this restriction will make the dependence model that MOAD learns less *complete*, it may make it more *sound*. Then after constructing a slice, we could perform some post-processing aimed at removing initialization statements of unused variables and return statements in unused methods. Doing so would enable us to explore the trade-off between *completeness* (e.g., how small the slice is) and *soundness* (e.g., avoiding unwantedly deleted lines).

6.2. Conditional dependency modeling and alternative inference models

We have seen that approximate dependency modeling can have benefits compared to traditional, static modeling approaches. While the latter consider only binary relations between program elements, MOAD models the probability that one program element is dependent on another. Note, however, that MOAD is an average or marginal probability model; it answers with a probability over all inputs. This points to important future work to build approximate models that are dependent on specific inputs, i.e. input-specific approximate dependency models. For some program analysis and software quality tasks such, even more refined models, might be beneficial.

As one example, in fault-localization, we have precise information about the testcase(s) that failed, and thus know which inputs to the program to consider. We would like to condition our probability model on the input(s) and get a more precise estimate of the probability for the situation at hand; a conditional dependency model. But we can go even further and consider other information to condition with. For example, in program slicing we might want to consider to condition on different slicing criteria.

These modeling situations will likely require more sophisticated modeling techniques and inference algorithms. Models such as Bayesian Networks (Jensen, 1996) might be used to encode the conditional independences between parts of programs and to infer the likelihood of program unit inclusion in slices based on the distributed joint distribution encoded therein. But they should also be investigated for answering more complex, input- and criteria-specific, modeling questions. 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 graph-like dependency relations of software, it is unclear if they are flexible enough for general, conditional dependency modeling. More general statistical inference models (e.g. Gaussian processes (Rasmussen, 2003)) and novel, 'deep' statistical modeling approaches (e.g. Sum-Product Networks (Poon and Domingos, 2011)) should also be explored in future work.

Overall, our argument points to a hierarchy of different program analysis questions and corresponding models to help answer them. Just like a traditional, binary program dependency model can be seen as subsumed by an averaging model as used in MOAD, the latter, in turn, would be subsumed by a dependent, input- and/or criteria-specific model. Which model to ultimately select likely varies with the downstream analysis task to be supported and with the specific program and its semantics. Future work should strive to clarify which model to select for a given task.

6.3. Additional applications

Future work will evaluate if the models we build can also be effectively used for tasks such as fault localization and fault comprehension (Baah et al., 2010; Feng and Gupta, 2010; Gong et al., 2015). It will also consider the trade-offs between precision and the usefulness of these models (e.g., early comprehension for orientation may benefit from slightly less precise models that retain some situational context in the slice, whereas bug-location may benefit from more precise slices).

7. Related work

There have been multiple proposals to approximate dependencies by complementing a statically extracted graph with dynamic information (Baah et al., 2010; Feng and Gupta, 2010; Gong et al., 2015). A notable example is Baah et al. (2010) 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 edges having conditional probabilities that relate states of child nodes to the states of their parents. Similarly, Feng and Gupta (2010) (using Bayesian Networks) and Gong et al. (2015) (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 (Santelices and Harrold, 2010) by augmenting static forward slices with relevance scores (labeling dependence edges in the interprocedural dependence graph with probabilities relating to coverage and propagation). While we also propose probabilistic modeling of dependences of program elements we do not need an existing program dependence graph nor detailed information about program execution state. Our coarser modeling is thus more generally applicable whilst still useful, at least for slicing.

8. Conclusion

This paper introduces and studies a new technique for 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 but also allow more several and specific analysis tasks to be supported. Furthermore, the cost is all upfront: once the model is built, inferring results is very inexpensive. A single model can also be reused for multiple analysis scenarios. 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, etc.).

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 5, illustrate the value of the probabilistic model. Specifically, MOAD can produce slices that are only 12% larger than the 1-minimal W-ORBS slices on average, from models built using only 18.6% of the observations required by W-ORBS. Furthermore, once the model is built, the slicing cost is negligible, and both backward and forward slices can be produced from a single model.

Artifact

Our implementation of MOAD is publicly available via zenodo (Lee, 2021).

CRediT authorship contribution statement

Seongmin Lee: Conceptualization, Methodology, Investigation, Software, Data curation, Writing - original draft, Writing - review & editing. **David Binkley:** Methodology, Investigation, Writing - original draft, Writing - review & editing. **Robert Feldt:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing. **Nicolas Gold:** Methodology, Writing - original draft, Writing - review & editing. **Shin Yoo:** Conceptualization, Methodology, Writing - original draft, Writing - Review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Seongmin Lee and Shin Yoo have been supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068179), as well as by National Research Foundation of Korea (NRF), South Korea Grant NRF-2020R1A2C1013629. Robert Feldt has been supported by the Swedish Scientific Council (No. 2015-04913, Basing Software Testing on Information Theory).

References

Baah, G.K., Podgurski, A., Harrold, M.J., 2010. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans. Softw. Eng. 36 (4), 528–545.

Binkley, D.W., 1993. Precise executable interprocedural slices. ACM Lett. Program. Lang. Syst. 3 (1–4), 31–45.

Binkley, D., 1997. Semantics guided regression test cost reduction. IEEE Trans. Softw. Eng. 23 (8), 498–516.

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

Binkley, D., Gold, N., Harman, M., Islam, S., Krinke, J., Yoo, S., 2015. ORBS and the limits of static slicing. In: Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM 2015. pp. 1–10.

Binkley, D., Gold, N., Islam, S., Krinke, J., Yoo, S., 2019. A comparison of tree- and line-oriented observational slicing. Empir. Softw. Eng. http://dx.doi.org/10.1007/s10664-018-9675-9.

Binkley, D., Harman, M., 2005. Locating dependence clusters and dependence pollution. In: 21St IEEE International Conference on Software Maintenance. Los Alamitos, California, USA. pp. 177–186.

Chaloner, K., Verdinelli, I., 1995. Bayesian experimental design: A review. Statist. Sci. 273–304.

Collard, M.L., Decker, M.J., Maletic, J.I., 2013. SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In: Int'L Conf. Softw. Maintenance. IEEE, pp. 516–519. http://dx.doi.org/10.1109/ICSM.2013.85.

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

Ettinger, R., Verbaere, M., 2004. Untangling: A slice extraction refactoring. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development. In: AOSD '04, ACM, New York, NY, USA, pp. 93–101. http://dx.doi.org/10.1145/976270.976283, URL: http://doi.acm.org/10.1145/976270.976283.

Feng, M., Gupta, R., 2010. 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, pp. 81–88.

Gallagher, K.B., 1989. Using Program Slicing in Software Maintenance (Ph.D. thesis). University of Maryland, Baltimore, Maryland.

Gold, N., Binkley, D., Harman, M., Islam, S., Krinke, J., Yoo, S., 2017. 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. ESEC/FSE 2017. pp. 547–558.

Gong, D., Su, X., Wang, T., Ma, P., Yu, W., 2015. State dependency probabilistic model for fault localization. Inf. Softw. Technol. 57, 430–445.

Grammatech Inc., 2002. The codesurfer slicing system. URL: www.grammatech.com.

Hajnal, Á., Forgács, I., 2012. A demand-driven approach to slicing legacy COBOL systems. J. Softw. Evol. Process 24 (1), 67–82. http://dx.doi.org/10.1002/smr.533.

Horwitz, S., Liblit, B., Polishchuk, M., 2009. Better debugging via output tracing and callstack-sensitive slicing. IEEE Trans. Softw. Eng. 36 (1), 7–19.

Horwitz, S., Reps, T., Binkley, D.W., 1990. Interprocedural Slicing Using Dependence Graphs. Vol. 12. pp. 26–61.

Jensen, F.V., 1996. Introduction to Bayesian Networks, first ed. Springer-Verlag, Berlin, Heidelberg.

Jiang, S., McMillan, C., Santelices, R., 2017. Do programmers do change impact analysis in debugging?. Empir. Softw. Eng. 22 (2), 631–669. http://dx.doi.org/10.1007/s10664-016-9441-9.

Karim, R., Tip, F., Sochurkova, A., Sen, K., 2019. Platform-independent dynamic taint analysis for javascript. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2018.2878020.

Lattner, C., Adve, V., 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA. pp. 75–88.

Lee, S., 2021. Coinse/MOAD: First release, may, Zenodo, v1.0.0, http://dx.doi.org/10.5281/zenodo.4740439.

Lee, S., Binkley, D., Feldt, R., Gold, N., Yoo, S., 2019. MOAD: Modeling observation-based approximate dependency. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp. 12–22.

Lee, S., Binkley, D., Gold, N., Islam, S., Krinke, J., Yoo, S., 2020. Evaluating lexical approximation of program dependence. J. Syst. Softw. 160, 110459. http://dx.doi.org/10.1016/j.jss.2019.110459, URL: http://www.sciencedirect.com/science/article/pii/S016412121930233X.

Livadas, P.E., Roy, P.K., 1992. Program dependence analysis. In: Proceedings of the International Conference on Software Maintenance 1992. Los Alamitos, California, USA. pp. 356–365.

Poon, H., Domingos, P., 2011. Sum-product networks: A new deep architecture. In: 2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops). IEEE, pp. 689–690.

Rasmussen, C.E., 2003. Gaussian processes in machine learning. In: Summer School on Machine Learning. Springer, pp. 63–71.

Santelices, R., Harrold, M.J., 2010. Probabilistic Slicing for Predictive Impact Analysis. Technical Report GIT-CERCS-10-10, Georgia Institute of Technology.

Szumilas, M., 2010. Explaining odds ratios. J. Canad. Acad. Child Adolesc. Psychiatry 19 (3), 227.

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

Weiser, M., 1979. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method (Ph.D. thesis). University of Michigan, Ann Arbor, MI.

Weiser, M., 1984. Program Slicing, Vol. 10. pp. 352–357.

Yoo, S., Binkley, D., Eastman, R., 2017. Observational slicing based on visual semantics. J. Syst. Softw. 129, 60–78.

Zhifeng Yu, Rajlich, V., 2001. Hidden dependencies in program comprehension and change propagation. In: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001. pp. 293–299. http://dx.doi.org/10.1109/WPC.2001.921739.

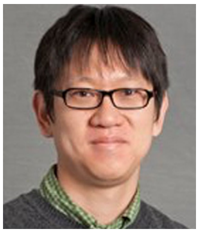
Zuluaga, M., Sergent, G., Krause, A., Püschel, M., 2013. Active learning for multi-objective optimization. In: International Conference on Machine Learning. pp. 462–470.

Seongmin Lee is a PhD candidate at School of Computing, KAIST, in Republic of Korea. He received BSc with a double major in School of Computing and Department of Mathematical Sciences from KAIST. His research interest includes program analysis, dependency analysis, program slicing, and genetic improvement.

David Binkley is a professor of Computer Science at Loyola University Maryland where he has worked since earning his doctorate from the University of Wisconsin in 1991. He has been a visiting faculty researcher at the National Institute of Standards and Technology (NIST), worked with Grammatech Inc. on CodeSurfer development, and was a member of the Crest Centre at Kings' College London. Dr. Binkley's current research, partially funded by NSF, focuses on change recommendation and observational program analysis. He recently completed a sabbatical year working under Fulbright award with the researchers at Simula Research, Oslo Norway.

Robert Feldt is a professor of Software Engineering at Chalmers University of Technology, Sweden, and at Blekinge Institute of Technology, Sweden. He has broad research interests spanning from human factors to automation, statistics, and applied machine learning, and he works, in particular, on software testing and quality, requirements engineering, as well as human-centered (behavioral) software engineering. Most of his research is empirical and conducted in close collaboration with industry partners in Sweden, Europe and Asia, but he also leads more basic research. Dr Feldt received a PhD in Computer Engineering from the Chalmers University of Technology in 2002, has studied Psychology at Gothenburg University in the '90s and has also worked as an IT and software consultant for more than 25 years. He is co-Editor in Chief of the Empirical Software Engineering journal and on the editorial board of two other journals (STVR and SQJ).

Nicolas Gold is an Associate Professor of Computer Science at University College London. He was awarded his doctorate from the University of Durham in 2000 and worked at UMIST and King's College London before joining UCL in 2010. His current research interests include program analysis, ethics, and applications of computer music in e-health and education.



Shin Yoo is an associate professor in the School of Computing at Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea. From 2012 to 2015, he was a lecturer of software engineering in Centre for Research on Evolution, Search, and Testing (CREST) at University College London, UK. He received PhD in Computer Science from King's College London, UK, in 2009. He received MSc and BSc from King's College London and Seoul National University, Korea, respectively. His main research interest lies in Search Based Software Engineering, i.e. the use of metaheuristics and computational intelligence, such as genetic algorithm, to automatically solve various problems in software engineering, especially those related to testing and debugging.