The Journal of Systems and Software 160 (2020) 110459

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Evaluating lexical approximation of program dependence

Seongmin Lee[a,*], David Binkley[c], Nicolas Gold[b], Syed Islam[b], Jens Krinke[b], Shin Yoo[a]

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

ARTICLE INFO

Article history:
Received 23 March 2019
Revised 3 September 2019
Accepted 4 November 2019
Available online 5 November 2019

Keywords:
ORBS
Program slicing
Lexical analysis

ABSTRACT

Complex dependence analysis typically provides an underpinning approximation of true program dependence. We investigate the effectiveness of using lexical information to approximate such dependence, introducing two new deletion operators to Observation-Based Slicing (ORBS). ORBS provides direct observation of program dependence, computing a slice using iterative, speculative deletion of program parts. Deletions become permanent if they do not affect the slicing criterion. The original ORBS uses a bounded deletion window operator that attempts to delete consecutive lines together. Our new deletion operators attempt to delete multiple, non-contiguous lines that are lexically similar to each other. We evaluate the lexical dependence approximation by exploring the trade-off between the precision and the speed of dependence analysis performed with new deletion operators. The deletion operators are evaluated independently, as well as collectively via a novel generalization of ORBS that exploits multiple deletion operators: Multi-operator Observation-Based Slicing (MOBS). An empirical evaluation using three Java projects, six C projects, and one multi-lingual project written in Python and C finds that the lexical information provides a useful approximation to the underlying dependence. On average, MOBS can delete 69% of lines deleted by the original ORBS, while taking only 36% of the wall clock time required by ORBS.

© 2019 Elsevier Inc. All rights reserved.

1. Introduction

Program slicing often acts as a preprocess for tasks such as testing (Binkley, 1998), debugging (Agrawal et al., 1993), maintenance (Gallagher and Lyle, 1991), and program comprehension (Korel and Rilling, 1998). Traditional program slicing techniques require complex dependence analysis to *approximate* the underlying dependencies (Binkley et al., 2015). Recently introduced Observation-Based Slicing (ORBS) (Binkley et al., 2014a; Gold et al., 2017) foregoes this need, discovering the *exact* dynamic dependencies given a set of inputs. However, this discovery can be expensive since, in the case of statements separated by (large amounts of) other code, many iterations may be required to remove all of the independent code. Given this shortcoming, we conjecture that lexical similarity (e.g., two lines that both include the string `tax_rate`) might provide a useful approximation to the dependence information. If so, then in the case of ORBS, similar lines can likely be deleted together, irrespective of their relative separation. Thus lexical similarity should bring value in the form of an efficiency gain (assuming that lexical dependence can act as a proxy for the true dependence).

As an example, consider a situation where source lines l_1 and l_2 relate to a (single) computation, unrelated to the slicing criterion, but sufficiently far apart from each other that they are never in the same deletion window (the contiguous group of lines selected by ORBS as a deletion candidate). Since they are unrelated to the slicing criterion they can in principle be deleted simultaneously. A lexical approximation of dependence seeks to identify the two as related and therefore potentially deletable in a single step. For example, l_1 might assign the variable `tax_rate`, which is subsequently used in l_2: `tax = tax_rate * sale`, while the slicing criterion concerns a variable independent from `tax_rate`. The lexical connection between l_1 and l_2 (i.e., the common occurrence of the words 'tax' and 'rate') might be exploited in an attempt to delete both lines simultaneously.

Exploiting the lexical dependence approximation, techniques that rely on the repeated compilation and execution for their operation should benefit because doing so would permit operations on spatially separated parts of the code. In the case of ORBS, for example, exploiting lexical dependence should enable the deletion of multiple spatially-separated lines in a single compile/execute step. Doing so would permit the full scope of the program to be examined for deletion candidates at any one time rather than just adja-

* Corresponding author.
E-mail addresses: bohrok@kaist.ac.kr (S. Lee), binkley@cs.loyola.edu (D. Binkley), n.gold@ucl.ac.uk (N. Gold), s.islam@cs.ucl.ac.uk (S. Islam), j.krinke@ucl.ac.uk (J. Krinke), shin.yoo@kaist.ac.kr (S. Yoo).

https://doi.org/10.1016/j.jss.2019.110459
0164-1212/© 2019 Elsevier Inc. All rights reserved.

cent lines. The net effect would be witnessed as an improvement in slicing performance.

This paper presents and investigates two approaches to the lexical approximation of dependence: one using the Vector Space Model (Salton et al., 1975) and the other using Latent Dirichlet Analysis (LDA) (Blei et al., 2003). To investigate the two, we use them as the basis for two new deletion operators within ORBS. The investigation first considers each new operator in isolation and then in concert with each other and the original ORBS deletion operator.

In the later case, we investigate the possibility of employing multiple deletion operators within ORBS. Ideally, the slicer could select the operator that leads to the largest successful deletion. In the absence of such knowledge, we consider two selection heuristics: Fixed Operator Selection (FOS), which relies on probabilities determined *a priori*, and Rolling Operator Selection (ROS), which learns the relative applicability of each deletion operator as it computes a slice. The resulting generalization of ORBS uses FOS or ROS to select between multiple deletion operators at each step.

The effectiveness of the lexical dependence approximation is measured using slicing *effectiveness* (i.e., how small are the resulting slices when compared to the original ORBS slices?) and slicing *efficiency* (i.e., how fast can the slicing be undertaken in comparison with the original ORBS algorithm?). In other words, what is the trade-off approximating the dependence lexically brings in terms of slice size and slice time.

We compare the original ORBS implementation (Binkley et al., 2014a) against the new deletion operators in isolation, and their combination with the existing deletion operator, using thirty slicing criteria selected from ten projects written in various programming languages including Java, C, and Python: eighteen slicing criteria from three real world Java projects, six slicing criteria from six C programs taken from the Siemens suite, and six slicing criteria from a multi-lingual open-source project written in Python and C. The results show that slicing using only lexical deletion operators leads to larger slices but take significantly less time. Furthermore, the combined use of the new and existing deletion operators deletes 63% to 83% of lines deleted by ORBS in 27% to 45% of the time.

This paper makes the following technical contributions:

- We introduce two new lexical dependence approximations.
- To study the two approximations independently we introduce two new variants of observation-based slicing, VSM-ORBS and LDA-ORBS, which each exploit lexical information. Broad stroke, the experiments show that lexical information can be used to complement existing techniques.
- To study the approximations in concert with the original, we introduce MOBS, *Multi-operator Observation-Based Slicing*, which selects a deletion operator from a set of deletion operators using four different operator selection strategies.
- We empirically evaluate the new lexical dependence approximations using thirty slicing criteria from Java and C projects, and a multi-lingual project written in Python and C. The results show that, on average, MOBS deletes 69% of the code deleted by ORBS, but requires only 36% of its execution time.

2. Observation-Based slicing

Based on Weiser's original slicing definition (Weiser, 1979), observation-based slicing (ORBS) dynamically checks whether one or more consecutive lines of program source code can be deleted by observing the impact of their deletion (Binkley et al., 2014a; Gold et al., 2017): if the source code after deletion either fails to compile or does not preserve the value trajectory of the slicing criterion when executed using the given set of inputs, then the deletion is rejected. While ORBS is designed to under-approximate the full semantics of program dependence through test executions, it has surpassed static slicing in some tasks, such as capturing dependencies that static slicers cannot handle (Binkley et al., 2015), and slicing multi-lingual systems (Binkley et al., 2014a), XML-based modelling languages (Gold et al., 2017), as well as Picture Description Languages, which have visual semantics (Yoo et al., 2017).

ORBS takes as input a slicing criteria and a set of inputs. It computes a slice as a series of decisions about whether to accept the deletion of successive source code lines. If, after deleting a line, the program still compiles and preserves the observed behaviour of the slicing criterion for all inputs, then ORBS accepts the deletion. ORBS iterates until no further deletions can be made. ORBS is thus able to uncover the "ground-truth" dependence with respect to the set of inputs.

One of the drawbacks of finding ground-truth dependence using ORBS is its inefficiency: ORBS requires a large number of compilations and executions. This partly arises because ORBS is relatively naive in its view of a program, working at the level of lines of text, and treating each line as simply deleted or not. Dependence is thus discovered on the basis of the presence or absence of a line, with no regard for its content (thus permitting language independence). Furthermore, considering only single lines limits ORBS ability to delete mutually dependent lines such as the braces around an empty block.

To address the mutually dependent lines problem, the original ORBS implementation (Binkley et al., 2014a), uses a *deletion window* approach, which enables it to handle sequences of source lines that can only be deleted together (e.g., the pair of brackets that enclose an empty block). Applied to line l_i, ORBS attempts to delete from one up 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, 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. ORBS performs multiple passes over the code until it cannot delete anything further. Because in practice it is more efficient, the implementation considers the lines of source code in reverse order, for example, in the hope of deleting all of a variable's uses before attempting the deletion of its declaration.

Algorithm 1: ORBS Parameterised with Deletion Operators.

```
input  : Source program P = {l_1, ..., l_n},Slicing criterion
         (v, l, I),Deletion Operator, D
output: A slice, S, of P for (v, l, I)
1  O ← SETUP(P, v, l)
2  V ← EXECUTE(BUILD(O), I)
3  repeat
4  |  deleted ← False
5  |  for i ← LENGTH(O) to 1 do
6  |  |  O', n_d, s_d ← D(O, V, i, I)
7  |  |  if s_d = success then
8  |  |  |  O ← O'
9  |  |  |  deleted ← True
10 until ¬deleted
11 return O
```

As shown in Algorithm 1, ORBS is parameterised by a deletion operator, D, along with the source program $\mathcal{P}$ and a slicing criterion composed of variable v, line l, a set of inputs $\mathcal{I}$. In the code the function SETUP inserts probe statements to capture the trajectory of the slicing criterion, while function EXECUTE executes the

program with inputs $\mathcal{I}$ and returns the trajectory associated with the slicing criterion.

Deletion operator D has four parameters: the current observational slice O, the original trajectory V, the index of the current line i, and the inputs $\mathcal{I}$. It first performs a deletion resulting in O', and then builds and executes O' using $\mathcal{I}$, capturing the trajectory and comparing it with V. The operator returns O', the number of deleted lines, n_d, and a result of the deletion attempt, s_d (either `compilation-error`, `trajectory-change`, or `success`). In the case of a successful deletion, O' is made the current observational slice.

Because it deletes the *maximal window up to size k*, the original ORBS deletion operator, shown as Algorithm 2, is denoted DMw^k. This deletion operator is defined in terms of window deletion operator, Dw$^\delta$, which attempts to delete a window of exactly δ consecutive lines. DMw^k successively applies the k deletion operators, from Dw^1 to Dw^k, and then returns success if any deletion operator succeeds, immediately, or returns failure if none of Dw^1 through Dw^k succeeds. To emphasize its use of a deletion *window*, we refer to the original ORBS algorithm, which implicitly uses DMw^k, as W-ORBS.

Algorithm 2: Pseudocode for Deletion Operator DMw^k.

```
input : Instrumented Source Code O,Value Trajectory of
        Slicing Criterion Variable, V,Index of Current Target
        Line, i,Set of inputs, I
output: A candidate slice, O',
        Number of Lines Deleted, n_d,
        Result of the Deletion Attempt, s_d
for δ ← 1 to k do
    O', n_d, s_d ← Dw^δ(O, V, i, I)
    if s_d = success then
        return O', n_d, s_d
return O', 0, s_d
```

3. Lexical similarity deletion operators

This section introduces the two new *lexical* deletion operators that seek to exploit lexical similarity within the text of a program. Lexical similarity can be a good proxy for syntactic or semantic similarity (Ragkhitwetsagul et al., 2018). Our new deletion operators are based on the intuition that if a source line can be safely deleted with respect to a given slicing criterion, then there are likely other lexically similar lines that can also be safely deleted. For example, if the slicing criterion involves a variable that holds an account balance in a banking system, then a line that handles logging will be deleted by ORBS. When deleting this line, it may be beneficial to attempt to delete all other lines that include the lexical token `log`. In other words, we posit that we can approximate the semantics of program dependence using lexical similarity. To this end we introduce the two lexical deletion operators: Dvsm and Dlda.

The first lexical deletion operator, Dvsm, makes use of the Vector Space Model (VSM), which has been used in Information Retrieval (IR) to calculate the distances between a collection of text documents and a query (Salton et al., 1975). VSM represents each document and the query as a vector of weights, that associate a value with each unique term (word) that occurs in any of the documents. The distance between vectors captures the similarity between the documents and the query.

In our application, Dvsm$^\gamma$ is parameterized by a threshold γ and treats all lines of text in the source code as individual documents. It identifies two lines as *similar* if their lexical similarity is greater than γ. In greater detail, let L be the set of all non-comment lexical tokens in the code and let K be a list of stop words typically composed of programming-language reserved keywords. The vocabulary used is the set of terms, $T = L \setminus K$. A range of techniques for assigning term weights and computing distances has been used in VSMs (Mitra and Chaudhuri, 2000). We use *tf–idf* to determine term weights (Rajaraman and Ullman, 2011) and measure distances using cosine similarity (Singhal, 2001).

The second lexical deletion operator, Dlda, makes use of Latent Dirichlet Allocation (LDA), which models a collection of documents using two probability distributions: each document is represented as a probability distribution of *topics* where each topic is a probability distribution over the words of the vocabulary (Blei et al., 2003). The similarity between two documents is measured as the distance between their corresponding topic vectors. Similar to Dvsm$^\gamma$, the deletion operator Dlda$^\gamma$ treats each source code line as a document, uses the same vocabulary, and also computes distances using cosine similarity.

4. Variants of ORBS

Our empirical investigation is designed to investigate the effectiveness of using lexical dependence approximation. To do so, we first introduce two variants of ORBS based on the two lexical deletion operators, Dvsmand Dlda. We then consider two additional variants that, analogous to the use of multiple window sizes by DMw^k, each consider a range of parameter (threshold) values. Looking ahead, the empirical analysis considers two final variants (bringing the total to six). These final two follow lexical deletion with the application of W-ORBS. Finally, having a range of deletion operators suggests the potential in trying the leverage the strengths of each. Thus the bulk of this section presents MOBS, Multi-operator Observational Slicing, which aims to selectively apply a range of deletion operators.

4.1. Lexical ORBS variants

The first two variants of ORBS are based on Algorithm 1 with D being passed one of the two lexical deletion operators. We refer to ORBS when using Dvsm$^\gamma$ as VSM-ORBS. Similarly, we refer to ORBS when using the deletion operator Dlda$^\gamma$ as LDA-ORBS. By virtue of using lexical deletion operators, VSM-ORBS and LDA-ORBS share a few distinguishing features that may yield advantages over W-ORBS.

1. There is no limit to the number of lines that can be deleted in a single deletion.
2. They can delete non-consecutive lines.
3. Because Dvsm and Dlda are lexical, ORBS language independence is preserved.
4. During a single iteration, only one deletion is attempted at each slicing point, unlike W-ORBS, which may attempt multiple deletions at each slicing point depending on the window size.

The other two variants successively apply a deletion operator with a range of parameter values analogous to Dw's successive application of a range of window sizes. Considering first VSM, Algorithm 3 shows the pseudo code for the successive application of Dvsm$^\gamma$ for a range of values of γ to target source code line, i. We denote this deletion operator as DMvsm$^\Gamma$, where Γ is a list of thresholds (parameter values). While it may spend more than one deletion attempt on a single line, DMvsm$^\Gamma$ thoroughly checks the lexical dependence the line could have with different thresholds. We refer to ORBS when using multiple thresholds with DMvsm$^\Gamma$ as VSM-ORBS-M. Likewise, we refer to ORBS when using DMlda$^\Gamma$ as LDA-ORBS-M.

Algorithm 3: Pseudocode for Deletion Operator $\text{DMVSM}^{\{\gamma_1,\ldots,\gamma_n\}}$. The input and output are the same as with Algorithm 2.

```
1 for γ_i ∈ {γ_1, ..., γ_n} do
2 |  O', n_d, s_d ← DVSM^{γ_i}(O, V, i, I)
3 |  if s_d = success then
4 |  |  return O', n_d, s_d
5 return O', 0, s_d
```

4.2. MOBS: multi-operator observational slicing

With three deletion operators, DW, DVSM, and DLDA, we can instantiate Algorithm 1 in different ways producing multiple ORBS' variants. However, because each deletion operator attempts to delete different parts of the code, a more synergistic approach might better exploit the strengths of each operator. Here our goal is to improve slicing performance by using the 'right' deletion operator at the 'right' time and in the 'right' place. To study the range of possibilities we introduce MOBS: *Multi-operator Observational Slicing*, which selectively applies multiple deletion operators while slicing.

4.2.1. MOBS Algorithm

Algorithm 4 presents MOBS, which has the same basic structure as Algorithm 1. MOBS makes use of three helper functions. The first of these, INITOPERATOR, initializes the deletion operator selection probabilities. The second, SELECTOPERATOR, chooses a deletion operator to apply at each line using roulette-wheel selection (Goldberg, 1989) based on the probability distribution on operators. Once chosen, the speculative deletion by MOBS is the same as that done by ORBS except that UPDATEOPERATORupdates the probability distribution according to the updater function U.

Algorithm 4: MOBS .

```
input : Source program P = {l_1, ..., l_n},Slicing criterion
        (v, l, I),Set of deletion operators
        D = {D_1, ..., D_n},Probability Updater U,Static
        Proportion R
output: A slice of P for (v, l, I)
1  O ← SETUP(P, v, l)
2  V ← EXECUTE(BUILD(O), I)
3  D ← INITOPERATOR(D, R)
4  repeat
5  |  deleted ← False
6  |  for i ← LENGTH(O) to 1 do
7  |  |  D_k ← SELECTOPERATOR(D)
8  |  |  O', n_d, s_d ← D_k(O, V, i, I)
9  |  |  D ← UPDATEOPERATOR(D, U, D_k, n_d, s_d)
10 |  |  if s_d = success then
11 |  |  |  O ← O'
12 |  |  |  deleted ← True
13 until ¬deleted
14 return O
```

4.2.2. Fixed operator selection (FOS)

The remainder of this section considers two operator selection strategies: Fixed Operator Selection (FOS) and Rolling Operator Selection (ROS). For a given slicing criterion FOS computes fixed probabilities based on the success proportion of each deletion operator and stores them in $\mathcal{D}$ in Line 3 of Algorithm 4. Such an exhaustive approach is not viable in production where an approximation over multiple criteria would be required, but for the experimental evaluation it serves to establish an upper bound on FOS performance.

We use two methods to compute the success proportion of an operator: the number of successful deletions and the number of lines deleted. We call the proportions calculated by each method its 'applicability' and 'effect', respectively. In addition, we study the use of a uniform proportion as a baseline. Note that for FOS the probabilities remain constant throughout slicing. In other words, the probability updater U is the identity function.

Algorithm 5: Applicability/Effect Measurement for FOS.

```
input : Source program P = {l_1, ..., l_n},
        Slicing criterion (v, l, I),
        Set of deletion operators D = {D_1, ..., D_n}
output: Static Proportion R
1  O ← SETUP(P, v, l)
2  V ← EXECUTE(BUILD(O), I)
3  R ← INITIALIZE(D, LENGTH(O))
4  for i ← LENGTH(O) to 1 do
5  |  foreach D_k ∈ D do
6  |  |  O', n_d, s_d ← D_k(O, V, i, I)
7  |  |  if s_d = success then
8  |  |  |  R_{D_k}[i] ← 1                    ▷~{applicability}
9  |  |  |  R_{D_k}[i] ← n_d                  ▷~{effect}
10 return R
```

Algorithm 5 details the calculation of the initial probabilities. It can compute applicability (if Line 8 is used) or effect (if Line 9 is used). Given a source program, a slicing criterion, and a set of deletion operators, $\mathcal{D}$, this algorithm returns a proportion array, R, for each deletion operator $\text{D}_k \in \mathcal{D}$. The function INITIALIZE first assigns each entry the value 0. The algorithm then iteratively applies each operator D_k to each source line and records in the proportion array R_{D_k}, either the deletion's successful application (when using Line 8) or the number of lines deleted (when using Line 9). INITOPERATOR, used in Line 3 of Algorithm 4, takes the proportion array and initializes the selection probability of each deletion operator D_k as follows:

$$P(\text{D}_k) = \frac{\sum_{1\le i\le n} R_{\text{D}_k}[i]}{\sum_{\text{D}_j\in\mathcal{D}} \sum_{1\le i\le n} R_{\text{D}_j}[i]} \tag{1}$$

4.2.3. Rolling operator selection (ROS)

In contrast to the FOS strategy, the Rolling Operator Selection, ROS, updates the probability after each deletion attempt. The intuition here is that early on different operators will be effective than when the slice approaches its final state. The probability distribution over the operators $P(\text{D}_k)$ is initialized with a uniform distribution. UPDATEOPERATOR, first, changes the probability of the current deletion operator using the probability updater U with respect to the result of the deletion attempt. Then, it normalizes the sum of the probability distribution to be 1.

In our study, we used Eq. (2) as the probability updater U. The penalty factor for a compilation failure, ω_{comp}, and an execution failure ω_{exec}, both range from zero to one. We penalise compilation failure more severely (i.e., $\omega_{\text{comp}} \le \omega_{\text{exec}}$) because successful compilation is necessary for a successful deletion. On the other hand, a successful deletion always increases the probability as $\log_{10}(n_d + 1) > 0$. Based on our empirical investigation, we set ω_{comp} as 0.98 and ω_{exec} as 0.99. The selection of the next deletion

operator makes use of the updated distribution.

$$P_{\text{new}}(\mathrm{D}_k) = \begin{cases} \omega_{\text{comp}} \cdot P(\mathrm{D}_k) & \texttt{compilation-error} \\ \omega_{\text{exec}} \cdot P(\mathrm{D}_k) & \texttt{trajectory-change} \\ (1 + \log_{10}(n_d + 1)) \cdot P(\mathrm{D}_k) & \texttt{success} \end{cases} \tag{2}$$

5. Research questions

We investigate the following six research questions:

RQ1. Lexical Deletion Operators: *How efficient and how effective are the lexical deletion operators?*

We compare the results of VSM-ORBS, LDA-ORBS, and W-ORBS with respect to the number of lines deleted (effectiveness) and the time taken to compute a slice (efficiency). We also investigate the impact of the similarity threshold parameter γ of the deletion operators $\textsc{Dvsm}^{\gamma}$ and $\textsc{Dlda}^{\gamma}$ used by VSM-ORBS and LDA-ORBS, respectively. Finally, we consider the impact of successive application of a range of different thresholds using VSM-ORBS-M and LDA-ORBS-M.

RQ2. Operator Comparison: *How different are the deletion operators from each other, both quantitatively and qualitatively?*

We compare the lexical dependence approximation provided by the two lexical deletion operators by comparing various statistics gathered when applying each operator to each line of the source code.

RQ3. Operator Selection Strategy: *What impact does the operator selection strategy have on MOBS's ability to exploit lexical dependence?*

In contrast to RQ2's head-to-head comparison, RQ3 begins the investigation into how the deletion operators complement each other. Our goal here is to determine which selection strategy to use in the subsequent experiments.

RQ4. Strategy Impact: *How does MOBS using the chosen selection strategy compare with W-ORBS?*

RQ4 compares MOBS with its best strategy (as determined when considering RQ3) against the original ORBS algorithm, W-ORBS. We again compare the results in terms of effectiveness and efficiency.

RQ5. Qualitative Analysis: *What impact do differences in the lexical dependencies considered have on the resulting slices?*

To provide a more intuitive feel for the impact of lexical dependence, we investigate characteristics of the slices produced by variants of ORBS and MOBS. The comparison considers several qualitative properties of the resulting slices.

RQ6. Scalability: *How well does the lexical dependency approximation scale?*

The lexical deletion operators preserve the language agnostic nature of ORBS. With this research question, we investigate the scalability of lexical deletion operators using a larger, multi-lingual program.

6. Experimental setup

6.1. Metrics

We define several performance metrics for use in the quantitative analysis. The first three, CPD (Compilations Per Deletion), EPD (Executions Per Deletion), and TPD (Time Per Deletion) capture the efficiency of a slicing method. For these metrics, the smaller the value, the better. On the other hand, DPS (Deletions Per Success) is the number of deleted lines per one successful application of the deletion operator. DPS evaluates the efficiency of a deletion operator: the larger the value, the better (the more efficient the operator).

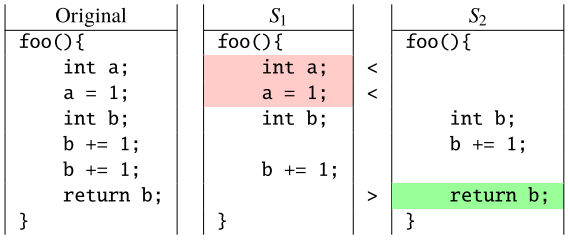

Fig. 1. Diff result of two slices of the original code. Our modified Jaccard similarity between S_1 and S_2 is: $\text{Jaccard}(S_1, S_2) = 4/(4+2+1) \approx 0.57$.

Finally, we use a variant of Jaccard similarity to calculate the similarity between two slices. The traditional definition of Jaccard similarity, which is a measure of similarity between two sets, is not appropriate for our purpose because, from the lexical viewpoint, a slice is an ordered multiset, i.e., the order of the lines matter and it may contain multiple instances of the same line. Furthermore, it is not viable to calculate the Jaccard similarity by making each line uniquely identifiable (e.g., by adding the tuple (file name, line number)). Fig. 1 shows an example of two slices, S_1 and S_2 where S_1 and S_2 delete '`b += 1;`' from different locations, but deletions result in the same common subsequence '`foo(){, int b;, b += 1;, }`'. Adding unique identifiers to lines would have unwontedly not produced the same result.

Given two sequences, S_1 and S_2, our variant of Jaccard similarity, modified for ordered multisets, is defined as follows:

$$\text{Jaccard}(S_1, S_2) = \frac{|C|}{|C| + |O_1| + |O_2|}$$

where C as the longest common subsequence[1] of S_1 and S_2, $O_1 = S_1 - C$, and $O_2 = S_2 - C$.

6.2. Subjects and environment

Table 1 shows the programs chosen for our empirical evaluation. For Java, we choose three open-source projects: commons-cli [2] and commons-csv [3] from Apache Commons Project, and guava ,[4] which is a core Java library developed by Google. We choose five slicing criteria from commons-cli, three from commons-csv, and ten from guava (five each from com.google.common.escape and com.google.common.net). For the C code, we choose the Siemens suite (Do et al., 2005). The program tcas has been excluded from the experiment, as it was too small for the lexical similarity models such as LDA to be applicable (tcas has only 120 Non-Comment Lines of Code (NCLOC). We choose one slicing criterion for each C program. Finally, the table provides the size of each subject program in NCLOC, as well as the number of test cases provided by the developers. The provided test cases will be used as inputs as part of the slicing criteria. The second from the last row shows the statistics of misaka, an open-source multi-lingual benchmark used to study RQ6. misaka includes both C and Python source code and has a total of over 5000 NCLOC.

To avoid the task of generating an obvious slice, we choose all slicing criteria thoroughly, making the dependency analysis challenging enough. Each slicing criterion consists of a variable located at a call depth of at least three. For Java, the slicing criteria are chosen from the class which has a dependency with at least three (seven on average) other classes.

[1] Note that a longest common subsequence refers to a non-consecutive subsequence, such as '`foo(){, int b;, b += 1;, }`' in S_2 of Fig. 1, whereas a longest common substring is a consecutive substring that is common to two strings.

[2] https://commons.apache.org/proper/commons-cli/

[3] https://commons.apache.org/proper/commons-csv/

[4] https://github.com/google/guava

Table 1
Subject programs and slicing criteria. The notation (A + B) for misaka denotes the statistics for its C and Python source code, respectively.

Lang.	Proj.	# of Files	NCLOC	# of Test cases	# of slicing criteria
Java	commons-cli	23	2081	26	5
	commons-csv	11	1504	13	3
	guava-escape	10	590	6	5
	guava-net	9	1569	8	5
C	prttok	1	410	11	1
	prttok2	1	387	10	1
	replace	1	508	15	1
	sched	1	208	6	1
	sched2	1	276	6	1
	totinfo	1	261	6	1
C + Python	misaka	15 (10 + 5)	5125 (4742 + 473)	92	6
	Total				30

For the purposes of this investigation, we filter out comments and reserved words prior to analysis. Although this violates language independence to some degree, it does not demand even the construction of a program's parse trees but is restricted to the lexical analysis in the matching of (regular expression) tokens to the stoplist. Consideration of all program elements forms part of our future work.

Experiments were performed on machines with Intel Core i7-6700K and 32GB RAM, running Ubuntu 14.04.5 LTS. Operator specific variants of ORBS (W-, VSM-, and LDA-) as well as MOBS have been implemented and executed in Python version 3.6.5. Java subjects have been built and executed using Java version 1.8.0_141 and JUnit version 4.12. C subjects have been built using GCC version 4.8.4.

6.3. Configuration

W-ORBS has a single parameter, δ, the maximum size of the deletion window. Our prior empirical study with W-ORBS has found that using four as the maximum window size provides the best performance. Thus, we use W-ORBS with maximum window size $\delta = 4$ as the baseline. While the studied subject programs and their test suites are deterministic, during the slicing process ORBS can produce nondeterministic candidate slices due to changes in control flow that arise from the deletions. Looking ahead, the impact of this non-determinism can be seen in Table 7 where the standard deviation of the number of deleted lines, σ_{del} by W-ORBS is non-zero in some cases. To account for this, as well as the randomness from the use of wall clock execution time, each W-ORBS slice was computed ten times.

Both VSM-ORBS and LDA-ORBS are parameterised by similarity threshold γ. Since cosine similarity is used, the similarity is in the range of [0, 1]. We report results using thresholds of 0.6, 0.7, 0.8, and 0.9. LDA-ORBS also requires selecting a topic count, which determines how many topics exist in the model. The best topic count depends heavily on properties such as the size and vocabulary of the documents. Tuning it typically requires manual inspection. We evaluated the values 25, 50, 100, 300, 500, 700, and 900 during the experiment for RQ1 and choose 500 as the best performer for Java projects; we also evaluated the values 25, 50, 75, 100, 200, and 300 for C projects, and chose 200 as the best performer (see Section 7.1 for more details). We set the LDA hyperparameters, α and β, which affect the sparsity of the document-topic and topic-word distributions of the LDA model, respectively, to the inverse of the topic count. Finally, for VSM-ORBS-M and LDA-ORBS-M we use two sequences for Γ: {0.6, 0.7, 0.8, 0.9} and {0.9, 0.8, 0.7, 0.6}, which we refer to as increasing order and decreasing order. For LDA-ORBS-M we use the same topic count as used with LDA-ORBS. Note that there is no need to repeat the VSM-ORBS(-M) and LDA-ORBS(-M) runs because VSM-ORBS is deterministic and LDA-ORBS is deterministic apart from the generation of the topic model. Our experience is that the variance from other parameters, such as the similarity threshold and topic count, is much more significant than the variation from rerunning the topic modelling.

The library of deletion operators used by MOBS includes the following twelve operators, which are different paramaterisations of the operators Dw, Dvsm, and Dlda:

- Dw^{δ} for deletion window size $\delta = 1$, 2, 3, and 4
- Dvsm^{γ} for threshold $\gamma = 0.6$, 0.7, 0.8, and 0.9
- Dlda^{γ} for threshold $\gamma = 0.6$, 0.7, 0.8, and 0.9

Due to the stochastic nature of the operator selection, and the use of wall clock time, like W-ORBS, we repeat each MOBS slice ten times for each slicing criterion.

7. Results

7.1. Lexical deletion operators

Before comparing the efficiency and effectiveness of VSM-ORBS, LDA-ORBS, and W-ORBS, we investigate how lexical deletion operators delete source code. Table 2 shows several example deletions. The first criterion of commons-cli, `cli-1`, involves the program's option-setting function. Dvsmsimultaneously deletes lines that are related to the option printing, but are irrelevant to the criterion. In the second example, Dvsmdeletes all function calls that handle a deficient token error from prttok2, since the criterion checks whether the input token is an identifier. Similar to `cli-1`, for the criterion `csv-1`, Dldadeletes all string building functions from commons-csv, because they are unrelated to the line-break checking function for a csv file.

To answer **RQ1**, we report results from comparisons between VSM-ORBS, LDA-ORBS and W-ORBS. As an example, Fig. 2 shows the comparison between the three slicers for guava-escape [5]: the x-axis indicates the variant of ORBS (γ denotes the threshold; n the topic count of LDA). On the left is a bar chart showing the number of deleted lines (blue), the number of compilations (light grey), and the number of executions (dark grey), required by each slicer. On the right is a bar chart showing the wall-clock time (red), and the number of lines deleted (blue) for each slicer. We also report the performance metrics CPD (denoted by ◯), EPD (×), and TPD (◊) with connected lines. Both VSM-ORBS and LDA-ORBS

[5] Plots for other slicing criteria as well as other RQs are available at https://coinse.github.io/MOBS_data_webpage/.

Table 2
Lines that have been deleted by lexical deletion operators. `cli-1` and `csv-1` represent the first criterion of commons-cli and commons-csv. `prt2` represents the criterion of prttok2.

	File-name:Line-num	Code line
`cli-1`, DVSM	HelpFormatter.java:166	`buff.append(’’ ’’);`
	HelpFormatter.java:173	`buff.append(’’[’’);`
	HelpFormatter.java:182	`buff.append(’’ | ’’);`
	HelpFormatter.java:186	`buff.append(’’]’’);`
	HelpFormatter.java:191	`buff.append(’’[’’);`
	HelpFormatter.java:203	`buff.append(’’]’’);`
	OptionGroup.java:46	`buff.append(’’[’’);`
	OptionGroup.java:50	`buff.append(’’-’’);`
	OptionGroup.java:53	`buff.append(’’--’’);`
	OptionGroup.java:57	`buff.append(’’ ’’);`
	OptionGroup.java:61	`buff.append(’’, ’’);`
	OptionGroup.java:64	`buff.append(’’]’’);`
`prt2` DVSM	print_tokens2.c:123	`unget_error(tp);`
	print_tokens2.c:130	`unget_error(tp);`
	print_tokens2.c:142	`unget_error(tp);`
`csv-1`, DLDA	CSVFormat.java:410	`sb.append(’’Delimiter=<’’).append(delimiter).append('>');`
	CSVFormat.java:416	`sb.append(' ');`
	CSVFormat.java:428	`sb.append(' ');`
	CSVFormat.java:443	`sb.append(’’HeaderComments:’’) .append(Arrays.toString(headerComments));`
	CSVFormat.java:447	`sb.append(’’Header:’’).append(Arrays.toString(header));`

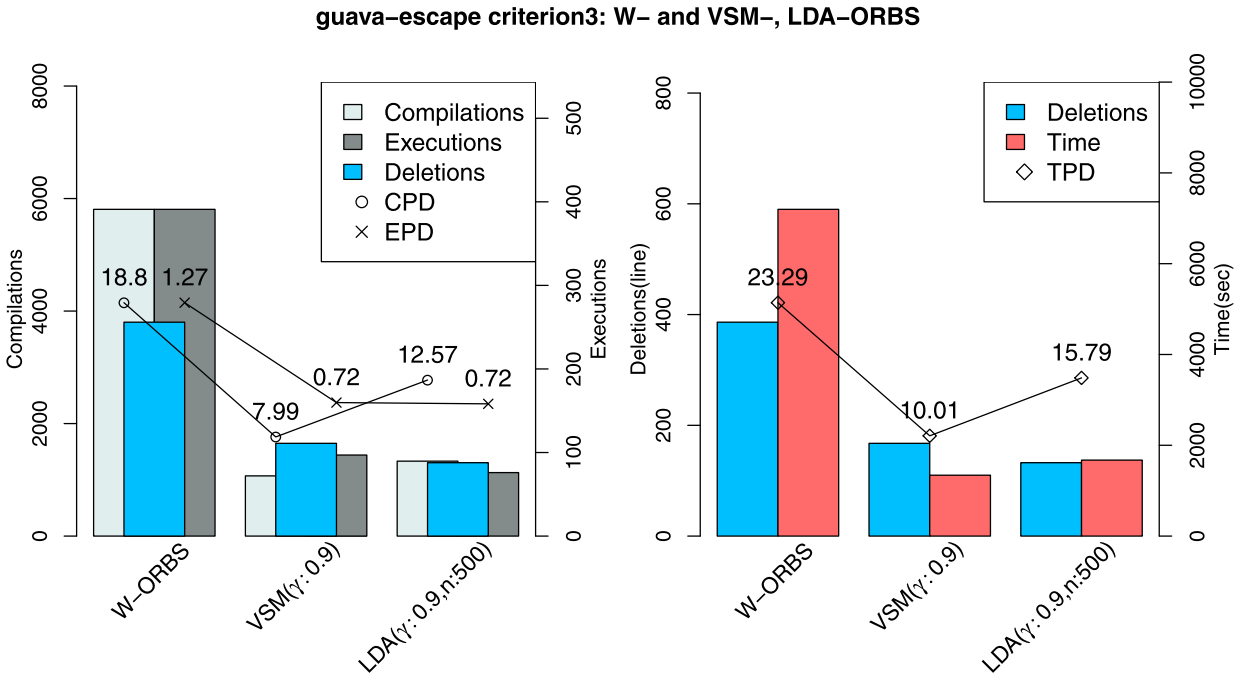

Fig. 2. Efficiency of ORBS variants. CPD, EPD, and TPD are the number of compilations, executions, and time taken per deleted line, respectively.

delete many fewer lines, but with significantly higher efficiency, as can be seen in their lower CPD, EPD, and TPD values: a similar trend is observed across all subjects.

The data in Table 3 compares the efficiency of W-ORBS, VSM-ORBS, and LDA-ORBS. For these results, the similarity threshold of VSM-ORBS and LDA-ORBS is set to 0.9; the topic count for LDA-ORBS is set to 500 for Java projects and 200 for C projects (the remainder of this Section explains the rationale behind these choices). On average, VSM-ORBS and LDA-ORBS delete 42.2% and 33.2% of the number of lines deleted by W-ORBS, respectively. However, VSM-ORBS uses only 19.4% of compilations and 29.2% of executions used by W-ORBS, requiring only 21.0% of the wall clock execution time of W-ORBS. Similarly, LDA-ORBS uses 19.4% of compilations, 22.1% of executions, and takes only 20.0% of the wall clock execution time of W-ORBS.

The results in Table 3 can be summarised using the per deleted line efficiency metrics as follows. For VSM-ORBS, its CPD, EPD, and TPD values are, on average across all subjects, 45.9%, 63.1%, and 49.5% of the corresponding W-ORBS's value. For LDA-ORBS they are 59.9%, 65.9%, and 62.2%, respectively.

Fig. 3 shows the results of VSM-ORBS with various similarity thresholds. As the threshold increases, the number of deleted lines tends to increase: that is, as VSM-ORBS targets similar lines with higher thresholds, it becomes more likely that it can delete those lines together. On the other hand, the number of compilations is relatively stable because compilation is performed every time ORBS attempts a deletion. Since the number of compilations is much larger than that of executions, the wall clock execution time tends to follow the trends of compilations. Consequently, the CPD and TPD values show similar trends and are dependent on how the

Table 3
Comparison of the number of compilations (C), executions (E), execution time (T, sec), and deleted lines (D) for W-ORBS, VSM-ORBS, and LDA-ORBS .

Criteria	W-ORBS				VSM-ORBS				LDA-ORBS			
	C	E	T	D	C	E	T	D	C	E	T	D
cli-1	21,052	2413	28,975	980	3680	470	5510	325	4330	408	5596	240
cli-2	22,241	2318	25,168	1123	3297	370	3920	375	4076	359	4813	291
cli-3	21,498	1886	26,165	1160	3583	391	4805	393	4195	359	5535	290
cli-4	23,163	2760	32,246	818	4472	579	7005	285	4306	478	6178	217
cli-5	24,340	2463	30,812	1144	3571	368	4594	364	4197	348	5119	279
csv-1	14,627	1373	25,146	696	2706	323	4805	219	2499	234	3939	155
csv-2	13,751	933	16,909	903	2718	253	3767	302	2897	187	3527	199
csv-3	11,979	760	13,713	1017	3240	250	3911	299	2834	173	2897	209
esc-1	5840	415	7894	239	1099	117	1586	115	1022	77	1229	95
esc-2	7174	517	9983	228	1100	124	1578	114	1012	77	1286	101
esc-3	5808	391	7196	309	1070	97	1341	134	1332	76	1673	106
esc-4	5387	284	7185	337	1178	90	1636	137	824	67	1049	119
esc-5	6163	458	8189	216	1109	127	1465	110	1001	102	1467	85
net-1	12,576	814	15,727	877	2780	397	3652	448	2630	234	3406	278
net-2	12,288	781	15,861	905	2779	393	3796	453	2569	233	3367	285
net-3	13,115	901	16,672	844	2373	396	3199	443	2272	231	2940	291
net-4	12,911	1806	17,933	842	2446	455	3854	368	2157	281	3035	231
net-5	12,004	739	14,736	925	2360	379	3014	463	2796	237	3522	308
prttok	2926	726	882	212	693	206	302	114	657	176	197	108
prttok2	3220	570	596	223	558	126	69	59	565	132	114	54
replace	6208	1254	1539	157	1219	392	373	92	839	230	210	61
sched	3186	589	208	93	459	136	34	32	743	216	81	34
sched2	1661	372	252	86	445	103	79	42	453	113	69	55
totinfo	1883	261	54	98	415	93	11	48	392	66	10	38

number of deleted lines changes. If the number of deleted lines varies significantly (as in commons-cli), CPD and TPD tend to increase; if the number of deletions varies little (as in commons-csv), CPD and TPD tend to be more stable.

Fig. 4 shows the results of LDA-ORBS with various threshold parameters. Overall, the results show similar trends to those of VSM-ORBS: the smaller the threshold value, the worse the efficiency. However, for C projects, the difference between the number of the compilations and executions is less than that of the Java projects; their compilation time is also smaller. Consequently, the trend of TPD is more likely to follow the EPD.

Let us briefly discuss the impact of topic count on LDA-ORBS. Fig. 5 shows the results of LDA-ORBS with various topic counts for commons-cli. In commons-cli, we observe the number of deleted lines increases until it reaches a maximum at topic count of $n = 500$, where it levels off. This is because, when n is too small, the topic model cannot capture sufficient features of the source code lines resulting in insufficient similarity and, consequently, fewer deleted lines. A similar trend is observed in the C subjects: the number of deleted lines increases as n brows to 200, and then levels off. Based on these observations, we use the topic counts 500 for Java and 200 for C in the remainder of our experiments, as these are the values around which the number of deletions improves and remains stable afterwards.

Finally, we consider the performance of VSM-ORBS-M and LDA-ORBS-M. Fig. 6 compares W-ORBS, VSM-ORBS with $\text{DVSM}^{0.9}$, and VSM-ORBS-M using the increasing and decreasing values for Γ. Results suggest that there is almost no difference in the number of lines deleted by VSM-ORBS and VSM-ORBS-M using either order. VSM-ORBS-M deletes only 0.8% and 0.5% more lines than VSM-ORBS with Γ increasing and decreasing, respectively. Meanwhile, VSM-ORBS-M (with either Γ) employs 49% more compilations than VSM-ORBS; thus, CPD increases 48% when compare to VSM-ORBS. LDA-ORBS and LDA-ORBS-M show a similar pattern; LDA-ORBS-M with Γ increasing and decreasing requires 48% and 50% more compilations than LDA-ORBS while deleting only 1.7% and 1.1% more lines than LDA-ORBS. The results imply that the different thresholds have minimal impact in terms of the lines that can be deleted using our lexical dependence approximation. Even so, we continue to consider different thresholds with the lexical deletion operators since their efficiency might differ from each other.

In summary for **RQ1**, while lexical deletion operators delete fewer lines, they use significantly fewer compilations and executions, reducing wall clock time. Both lexical deletion operators are highly attractive in terms of their per-deleted-line efficiency, motivating MOBS's use of multiple deletion operators.

7.2. Operator comparison

To answer **RQ2**, we investigate the relative applicability of different deletion operators. This is done by applying each deletion operator to all non-comment lines of code in the program's source to identify which lines are successful application points. Let W, V, and L be the set of lines against which $\text{Dw}^{\delta}(\delta \in \{1, \ldots, 4\})$, and $\text{DVSM}^{\gamma}\,\text{DLDA}^{\gamma}$ $(\gamma \in \{0.6, 0.7, 0.8, 0.9\})$ can be successfully applied: let $\sqcup$ denote the union of W, V, and L. In this experiment, we compute all pair-wise set differences to check how uniquely the operators can be applied to different locations.

Table 4 presents sizes of W, V, and L as well as set differences between them, along with DPS_W, DPS_V, and DPS_L. DPS_W is the average number of deleted lines across all successful applications of each Dw^{δ} operator to all source code lines. Similarly, DPS_V and DPS_L are the average number of deleted lines across the applications of DVSM and DLDA to all lines of source code, respectively.

In all cases, the size of W is either identical or very close to that of the union, while sizes of V and L are significantly smaller than that of W; $|V \backslash W|$ and $|L \backslash W|$ are always close to zero. Overall, this suggests that Dwcan be applied to the largest number of lines successfully.

However, the results from the DPS analysis provide evidence that lexical deletion can improve the efficiency of ORBS. The results show that, for most of the Java subjects, DPS_V is notably higher than DPS_W, suggesting that, when successful, DVSMis capable of deleting more lines per attempt than Dw. DPS_L shows mixed results. For C projects, DPS values are almost equal between window and lexical deletion operators. We suspect that, due to the smaller

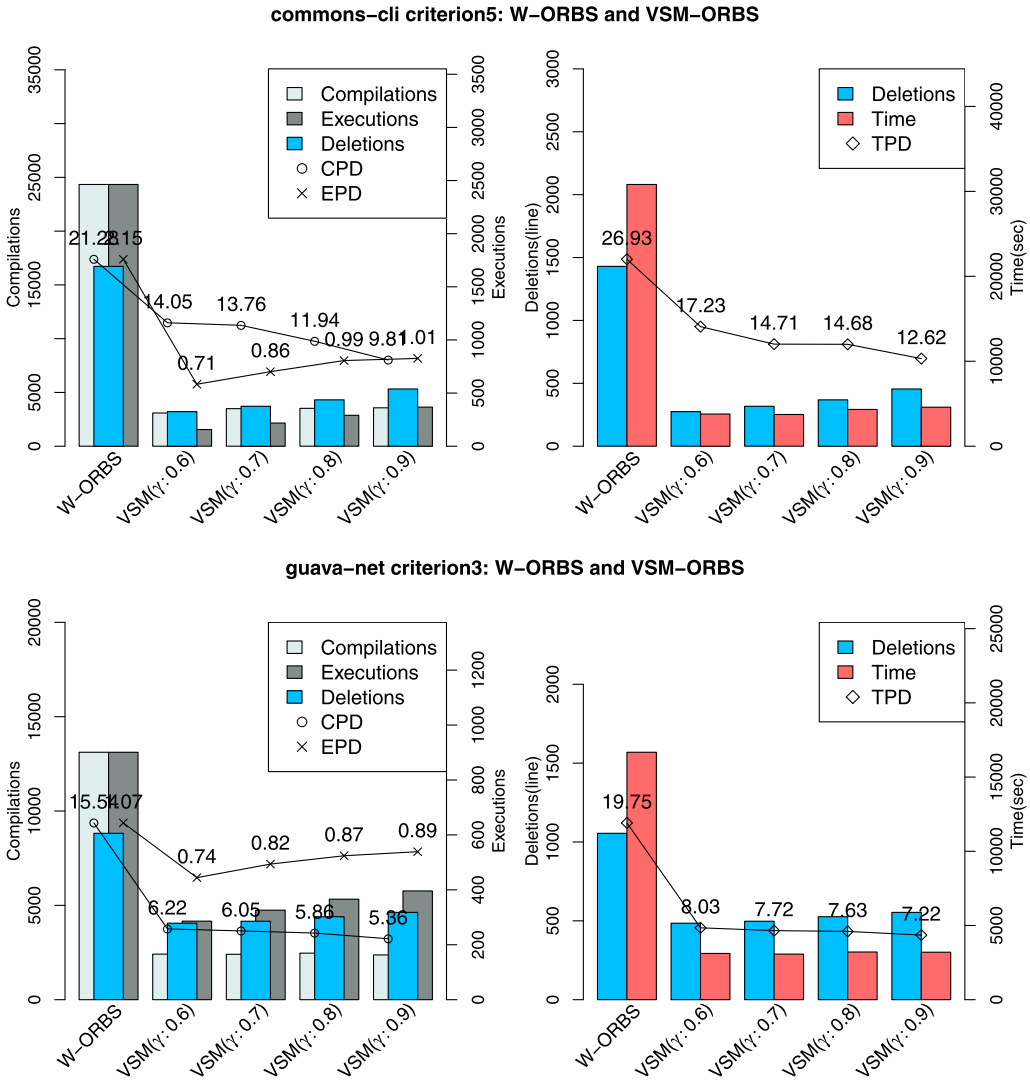

Fig. 3. VSM-ORBS: Threshold analysis.

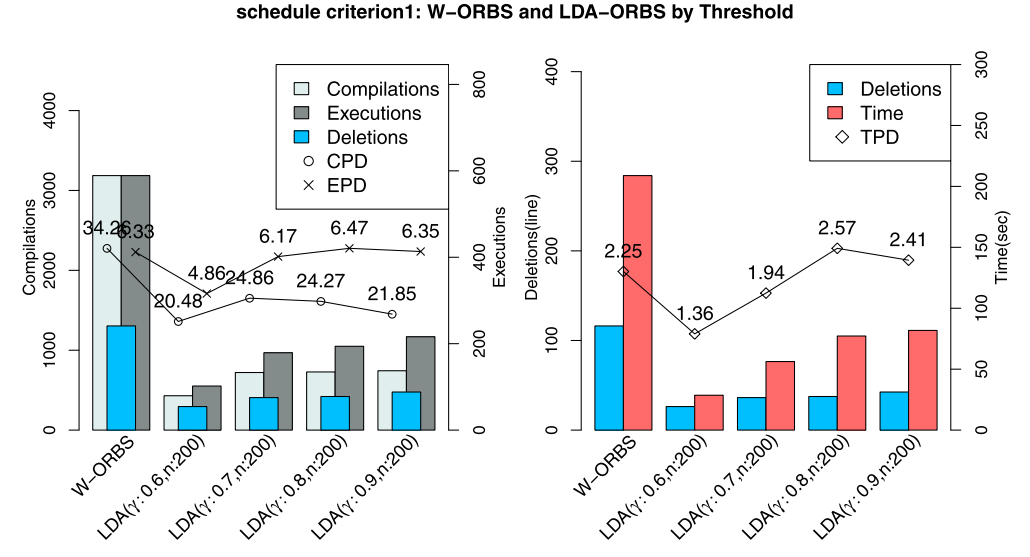

Fig. 4. LDA-ORBS: Threshold value comparison.

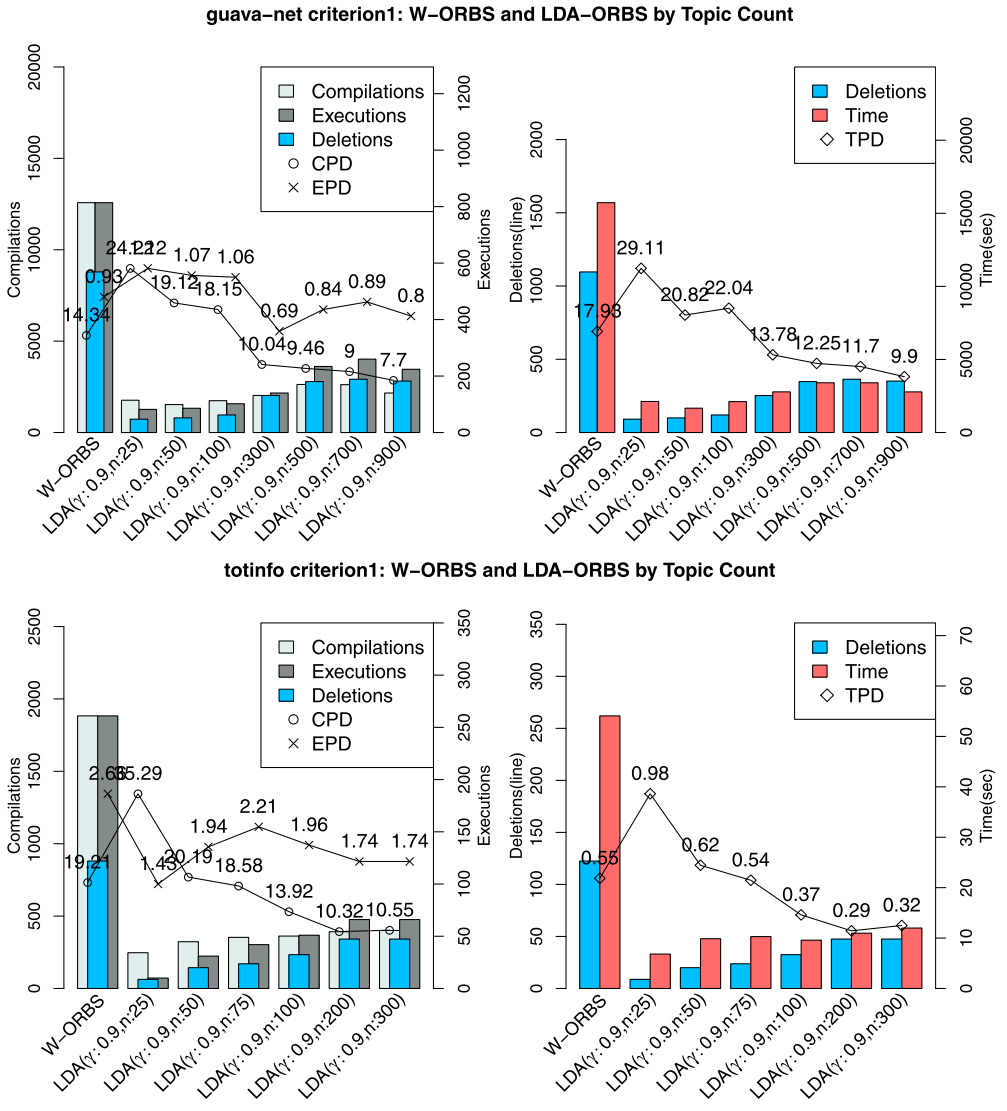

Fig. 5. LDA-ORBS: Impact of topic count.

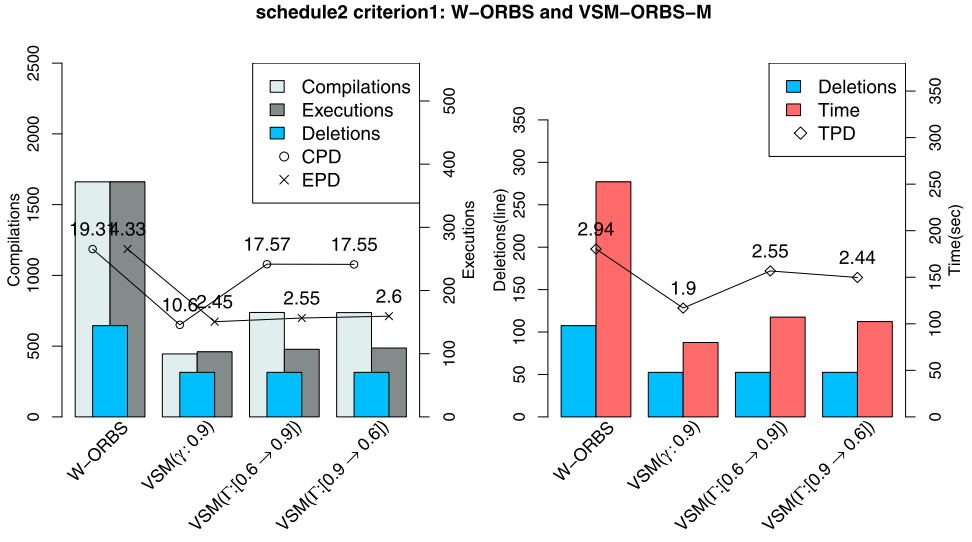

Fig. 6. Efficiency of VSM-ORBS-M. The notation $\Gamma = [0.6 \rightarrow 0.9]$ denotes the values 0.6, 0.7, 0.8, 0.9, while $[0.9 \rightarrow 0.6]$ denotes the same values in decreasing order.

Table 4
Comparison between deletion operators.

Criteria	⊔	*W*	*V*	*L*	*W**V*	*W**L*	*V**W*	*V**L*	*L**W*	*L**V*	*DPS*		
											W	*V*	*L*
`cli-1`	661	660	288	213	373	447	1	102	0	27	2.11	3.98	1.22
`cli-2`	804	802	347	285	456	518	1	95	1	33	2.11	4.45	1.94
`cli-3`	767	766	363	280	404	486	1	108	0	25	2.15	4.37	2.0
`cli-4`	549	548	225	192	323	357	0	58	1	25	2.04	5.35	2.84
`cli-5`	722	721	334	263	388	458	1	96	0	25	2.13	4.26	1.96
`csv-1`	530	530	197	141	333	389	0	67	0	11	2.31	3.76	1.59
`csv-2`	624	623	254	176	369	448	0	89	1	11	2.31	3.48	1.62
`csv-3`	670	670	255	186	415	484	0	82	0	13	2.34	3.46	1.55
`esc-1`	190	185	95	80	92	109	2	29	4	14	1.96	5.48	2.03
`esc-2`	169	160	89	77	73	91	2	25	8	13	1.89	5.75	2.04
`esc-3`	215	207	111	89	98	125	2	38	7	16	1.98	5.08	1.85
`esc-4`	252	237	118	111	122	140	3	30	14	23	2.03	4.78	4.93
`esc-5`	176	170	90	73	82	102	2	28	5	11	1.94	5.66	2.15
`net-1`	679	675	421	269	257	409	3	162	3	10	2.31	2.76	1.76
`net-2`	667	658	422	275	239	391	3	164	8	17	2.3	2.78	1.62
`net-3`	649	642	416	288	228	361	2	147	7	19	2.32	2.8	2.53
`net-4`	606	602	341	222	265	382	4	130	2	11	2.28	3.05	1.82
`net-5`	702	697	438	292	262	408	3	161	3	15	2.31	2.73	1.79
`prttok`	179	179	60	56	119	123	0	10	0	6	2.25	1.38	1.26
`prttok2`	189	184	107	95	82	93	5	24	4	12	2.23	2.81	3.21
`replace`	120	120	75	51	45	69	0	26	0	2	1.42	1.36	1.32
`sched`	91	90	40	52	51	39	1	2	1	14	1.61	1.39	1.45
`sched2`	54	54	30	26	24	28	0	6	0	2	1.56	1.15	1.34
`totinfo`	92	90	47	38	45	54	2	10	2	1	2.01	1.38	1.43

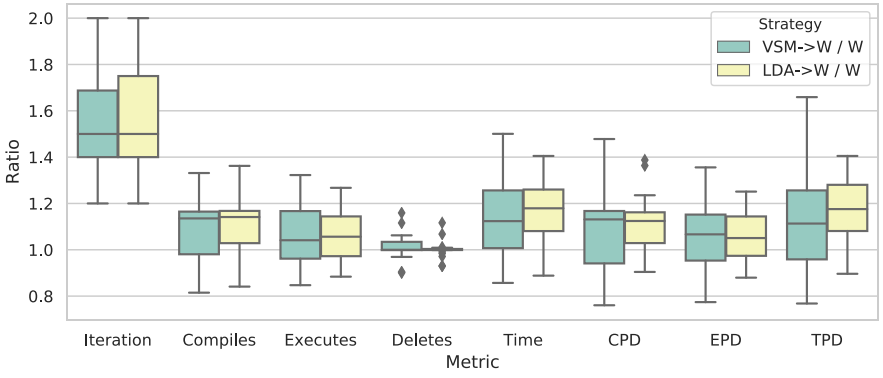

Fig. 7. Ratios comparing VSM-ORBS or LDA-ORBS followed by W-ORBS to W-ORBS alone.

size of these subjects, there are fewer similar source code lines for the lexical operators to exploit.

Note that higher DPS values do not necessarily mean that the corresponding operator will be highly *applicable*, as it measures the expected deletions per *successful* application. However, the results in Table 4 suggest that, if applied appropriately, lexical deletion operators stand to improve the efficiency of ORBS by deleting a greater number of lines per attempt.

The contrast in operator behaviours provides an answer to **RQ2** and suggests the use of both window and lexical deletion operators during observation-based slicing. We initially tried a naive combination that ran W-ORBS on the slice generated by VSM-ORBS or LDA-ORBS. The strategy first deletes lines that can be detected using lexical deletion and then applies window deletion to the remaining lines. In doing so, it aims to combine the efficiency gain of lexical deletion with the smaller slices attained using window deletion.

Fig. 7 compares the results of applying VSM-ORBS or LDA-ORBS followed by W-ORBS to those attained by applying W-ORBS alone. For the numerator, the number of iterations, compilations, executions, lines deleted, and the time taken is the sum of the value for VSM-ORBS or LDA-ORBS plus that for W-ORBS. The box plots in the figure summarize the ratios for each of the 24 slicing criteria studied. The green box is the ratio of W-ORBS after VSM-ORBS to W-ORBS, while the yellow box is the ratio of W-ORBS after LDA-ORBS to W-ORBS. On average, there is a slight increase in the number of deleted lines. However, the cost (e.g., the number of compilations, executions, and the time taken) increase dramatically. Interestingly, the initial application of lexical deletion does reduce the number iterations used in the subsequent application of W-ORBS. In 11 of the 24 slicing criteria, W-ORBS following VSM-ORBS or LDA-ORBS requires one fewer iteration, while for `replace` does it required two fewer.

The data shows that applying window deletion as a “second pass” does not improve slicing efficiency. Even so, initial lexical deletions do help to break certain dependence chains and thus reduce the slice size and the number of iterations W-ORBS requires. This potential synergy between window and lexical deletion further motivates MOBS. The interplay between the two suggests that a more sophisticated combination might be able to exploit the strengths of each approach. It is this potential that motivates our introduction and subsequent study of MOBS.

Table 5
The selection probability for FOS with applicability for each deletion operator.

Criteria	Dw with $\delta =$				Dvsm with $\gamma =$				Dlda with $\gamma =$			
	1	2	3	4	0.6	0.7	0.8	0.9	0.6	0.7	0.8	0.9
cli-1	0.162	0.069	0.082	0.059	0.065	0.080	0.098	0.119	0.050	0.058	0.072	0.088
cli-2	0.155	0.062	0.088	0.052	0.064	0.080	0.097	0.118	0.055	0.063	0.076	0.091
cli-3	0.153	0.073	0.081	0.063	0.061	0.077	0.096	0.119	0.050	0.059	0.076	0.092
cli-4	0.159	0.056	0.075	0.048	0.064	0.082	0.103	0.113	0.054	0.066	0.085	0.095
cli-5	0.152	0.074	0.077	0.061	0.062	0.077	0.092	0.114	0.056	0.065	0.079	0.091
csv-1	0.139	0.075	0.098	0.081	0.059	0.077	0.094	0.109	0.059	0.060	0.069	0.078
csv-2	0.135	0.080	0.102	0.079	0.061	0.081	0.098	0.111	0.054	0.057	0.068	0.075
csv-3	0.131	0.076	0.110	0.079	0.060	0.079	0.096	0.108	0.057	0.060	0.069	0.077
esc-1	0.140	0.043	0.049	0.040	0.088	0.098	0.107	0.114	0.068	0.071	0.089	0.094
esc-2	0.142	0.045	0.042	0.036	0.092	0.104	0.109	0.116	0.060	0.067	0.089	0.097
esc-3	0.146	0.046	0.052	0.043	0.093	0.103	0.113	0.126	0.059	0.059	0.074	0.087
esc-4	0.131	0.041	0.052	0.043	0.084	0.092	0.101	0.111	0.074	0.073	0.095	0.103
esc-5	0.145	0.047	0.048	0.040	0.093	0.104	0.112	0.118	0.056	0.062	0.086	0.088
net-1	0.118	0.072	0.080	0.073	0.096	0.101	0.108	0.112	0.049	0.053	0.064	0.074
net-2	0.120	0.074	0.079	0.071	0.098	0.104	0.111	0.115	0.044	0.048	0.063	0.073
net-3	0.115	0.071	0.075	0.072	0.095	0.100	0.107	0.111	0.050	0.056	0.069	0.079
net-4	0.125	0.063	0.078	0.070	0.094	0.101	0.111	0.116	0.048	0.053	0.067	0.075
net-5	0.117	0.069	0.076	0.072	0.095	0.101	0.108	0.112	0.051	0.056	0.068	0.074
prttok	0.134	0.091	0.080	0.073	0.065	0.077	0.089	0.092	0.073	0.073	0.075	0.080
prttok2	0.160	0.099	0.130	0.082	0.052	0.055	0.069	0.085	0.058	0.060	0.067	0.084
replace	0.191	0.047	0.017	0.015	0.069	0.073	0.109	0.140	0.075	0.075	0.094	0.096
sched	0.180	0.051	0.035	0.012	0.055	0.090	0.094	0.118	0.082	0.090	0.090	0.102
sched2	0.151	0.034	0.041	0.009	0.073	0.084	0.089	0.089	0.096	0.103	0.114	0.116
totinfo	0.137	0.054	0.041	0.056	0.090	0.092	0.099	0.106	0.083	0.079	0.081	0.081

Table 6
The selection probability for FOS with effect for each deletion operator.

Criteria	Dw with $\delta =$				Dvsm with $\gamma =$				Dlda with $\gamma =$			
	1	2	3	4	0.6	0.7	0.8	0.9	0.6	0.7	0.8	0.9
cli-1	0.064	0.054	0.097	0.093	0.140	0.138	0.134	0.152	0.024	0.029	0.035	0.040
cli-2	0.053	0.042	0.091	0.072	0.131	0.148	0.133	0.137	0.046	0.046	0.051	0.048
cli-3	0.053	0.050	0.084	0.087	0.126	0.142	0.131	0.135	0.043	0.045	0.053	0.051
cli-4	0.046	0.032	0.065	0.055	0.132	0.155	0.142	0.129	0.035	0.062	0.087	0.062
cli-5	0.054	0.053	0.082	0.087	0.121	0.143	0.129	0.128	0.046	0.053	0.055	0.049
csv-1	0.053	0.058	0.112	0.124	0.113	0.121	0.124	0.132	0.039	0.040	0.040	0.044
csv-2	0.053	0.063	0.120	0.124	0.118	0.120	0.118	0.122	0.037	0.041	0.042	0.041
csv-3	0.052	0.060	0.131	0.125	0.116	0.118	0.117	0.120	0.039	0.038	0.041	0.043
esc-1	0.041	0.025	0.043	0.047	0.159	0.163	0.165	0.165	0.047	0.046	0.050	0.048
esc-2	0.040	0.025	0.036	0.040	0.167	0.172	0.170	0.171	0.043	0.046	0.044	0.046
esc-3	0.044	0.028	0.047	0.053	0.164	0.168	0.169	0.171	0.041	0.039	0.040	0.037
esc-4	0.032	0.020	0.038	0.042	0.113	0.113	0.114	0.114	0.104	0.103	0.108	0.100
esc-5	0.040	0.026	0.040	0.044	0.165	0.169	0.170	0.170	0.046	0.047	0.045	0.037
net-1	0.050	0.061	0.101	0.123	0.135	0.122	0.117	0.113	0.040	0.041	0.044	0.053
net-2	0.051	0.063	0.101	0.122	0.139	0.127	0.123	0.117	0.032	0.033	0.044	0.048
net-3	0.045	0.055	0.088	0.112	0.123	0.113	0.110	0.105	0.049	0.059	0.069	0.074
net-4	0.050	0.050	0.094	0.112	0.141	0.128	0.126	0.121	0.035	0.040	0.049	0.053
net-5	0.050	0.059	0.097	0.123	0.131	0.121	0.118	0.113	0.045	0.044	0.050	0.050
prttok	0.050	0.067	0.087	0.106	0.101	0.085	0.074	0.074	0.093	0.092	0.088	0.082
prttok2	0.091	0.111	0.219	0.182	0.053	0.043	0.051	0.057	0.042	0.042	0.048	0.061
replace	0.140	0.067	0.034	0.040	0.074	0.073	0.107	0.136	0.076	0.076	0.088	0.089
sched	0.135	0.076	0.079	0.035	0.047	0.079	0.082	0.100	0.085	0.091	0.091	0.100
sched2	0.102	0.047	0.084	0.025	0.075	0.085	0.082	0.076	0.102	0.102	0.115	0.104
totinfo	0.087	0.067	0.074	0.138	0.083	0.084	0.086	0.087	0.081	0.070	0.071	0.071

7.3. Operator selection strategy

Tables 5 and 6 contain the probability of each operators calculated using applicability and effect for FOS, respectively (Section 4.2.2). Table 5 shows that operators that delete fewer lines tend to take higher probability (i.e., opportunities to delete a large number of lines together are rare) for all subjects. Dw[1] has the highest probability, and consequently will be most frequently selected by FOS-MOBS when using applicability. However, in an interesting contrast, Table 6 shows that Dvsmhas a much higher probability than other deletion operators for the Java projects (i.e., when those rare opportunities arise, Dvsmcan delete a sufficiently large number of lines to overcome its rareness). Among the Dw operators, applicability shows a negative correlation while a positive correlation is observed between δ and effect. Note that there does not exist an observable trend in effect for the C subjects. We suspect the higher verbosity of Java code compared to C may yield the higher applicability of Dvsm, due to the richer lexical information in the source code: however, further study is required to confirm this. Finally, probabilities of Dlda remain relatively low across all subjects.

Fig. 8 shows the result of Vargha-Delaney $\hat{A}_{12}$ statistic (Vargha and Delaney, 2000) for the TPD values of the four different operator selection strategies used by MOBS. Each of ROS,

Table 7
Statistics on execution time and the number of deleted lines for W-ORBS and MOBS.

Criteria	Strategy	μ_{del}	σ_{del}	μ_{time}	σ_{time}	μ_{tpd}	σ_{tpd}	Criteria	Strategy	μ_{del}	σ_{del}	μ_{time}	σ_{time}	μ_{tpd}	σ_{tpd}
cli-1	ROS-MOBS	**721.40**	28.64	11272.02	906.65	**15.65**	1.41	esc-5	ROS-MOBS	140.70	8.50	2434.85	128.74	17.37	1.47
(W-ORBS	FOS-uni-MOBS	608.20	17.33	**10961.53**	513.61	18.06	1.26	(W-ORBS	FOS-uni-MOBS	**154.00**	8.88	2457.16	103.25	**16.03**	1.36
Iter.:5)	FOS-app-MOBS	597.50	21.38	11175.68	480.81	18.74	1.25	Iter.:4)	FOS-app-MOBS	137.70	8.12	2542.52	114.95	18.53	1.39
	FOS-eff-MOBS	598.90	23.42	11158.46	320.87	18.67	1.05		FOS-eff-MOBS	129.30	7.07	**2418.29**	118.81	18.77	1.56
	W-ORBS	979.30	0.46	28134.40	1088.38	28.73	1.11		W-ORBS	216.00	0.00	8069.86	355.38	37.36	1.65
cli-2	ROS-MOBS	**791.40**	139.13	**9593.28**	1932.33	**12.09**	0.90	net-1	ROS-MOBS	561.00	90.04	**5558.79**	1396.09	**9.79**	1.37
(W-ORBS	FOS-uni-MOBS	726.40	16.04	10276.15	476.00	14.16	0.85	(W-ORBS	FOS-uni-MOBS	621.80	14.08	6776.59	349.01	10.91	0.65
Iter.:6)	FOS-app-MOBS	709.20	16.15	10568.47	406.63	14.91	0.70	Iter.:5)	FOS-app-MOBS	617.90	20.76	6821.67	328.34	11.05	0.69
	FOS-eff-MOBS	690.10	23.39	10316.25	404.77	14.97	0.91		FOS-eff-MOBS	**645.30**	19.74	6778.88	333.59	10.52	0.76
	W-ORBS	1123.00	0.00	25224.20	702.45	22.46	0.63		W-ORBS	877.00	0.00	16250.56	878.72	18.53	1.00
cli-3	ROS-MOBS	**872.70**	114.47	**10837.41**	1376.55	**12.47**	0.92	net-2	ROS-MOBS	497.80	93.83	**4589.39**	1364.66	**9.06**	1.03
(W-ORBS	FOS-uni-MOBS	763.40	16.05	11228.13	437.73	14.71	0.57	(W-ORBS	FOS-uni-MOBS	637.60	17.10	6902.16	376.68	10.84	0.72
Iter.:6)	FOS-app-MOBS	774.00	26.24	11226.59	521.40	14.53	0.94	Iter.:5)	FOS-app-MOBS	638.30	17.75	6952.53	314.39	10.90	0.55
	FOS-eff-MOBS	752.20	19.16	11110.52	459.07	14.79	0.88		FOS-eff-MOBS	**659.20**	14.10	6668.74	335.09	10.12	0.60
	W-ORBS	1160.00	0.00	26109.18	737.05	22.51	0.64		W-ORBS	905.00	0.00	15605.63	753.18	17.24	0.83
cli-4	ROS-MOBS	**645.20**	17.91	12126.29	593.73	**18.78**	0.48	net-3	ROS-MOBS	474.80	76.77	**4212.45**	1394.51	**8.70**	1.69
(W-ORBS	FOS-uni-MOBS	504.20	17.38	11692.39	392.95	23.21	1.01	(W-ORBS	FOS-uni-MOBS	**623.10**	16.63	6826.59	300.39	10.97	0.60
Iter.:5)	FOS-app-MOBS	493.40	22.50	11899.10	487.58	24.19	1.88	Iter.:5)	FOS-app-MOBS	609.60	13.37	6893.63	273.25	11.31	0.51
	FOS-eff-MOBS	434.90	16.02	**11616.53**	532.42	26.76	1.86		FOS-eff-MOBS	613.30	17.05	6666.82	361.27	10.89	0.77
	W-ORBS	817.20	0.75	31665.95	930.87	38.75	1.13		W-ORBS	844.00	0.00	16499.28	681.01	19.55	0.81
cli-5	ROS-MOBS	**826.80**	119.27	**10490.76**	1205.87	**12.79**	0.95	net-4	ROS-MOBS	471.20	86.42	**5243.92**	944.86	**11.16**	0.95
(W-ORBS	FOS-uni-MOBS	726.60	13.45	10914.85	564.61	15.03	0.91	(W-ORBS	FOS-uni-MOBS	518.80	15.33	6432.89	279.95	12.40	0.51
Iter.:6)	FOS-app-MOBS	725.20	18.48	11124.64	531.14	15.35	0.92	Iter.:4)	FOS-app-MOBS	516.70	9.57	6451.88	306.13	12.49	0.66
	FOS-eff-MOBS	702.10	16.36	10916.40	513.33	15.56	0.80		FOS-eff-MOBS	**527.60**	17.20	6347.30	252.24	12.05	0.78
	W-ORBS	1144.00	0.00	29801.89	1227.73	26.05	1.07		W-ORBS	842.00	0.00	17701.25	838.83	21.02	1.00
csv-1	ROS-MOBS	**507.70**	74.33	9581.00	1155.33	**19.17**	2.58	net-5	ROS-MOBS	505.00	95.67	**4072.71**	1325.93	**7.89**	1.11
(W-ORBS	FOS-uni-MOBS	439.80	12.28	9117.84	364.35	20.73	0.66	(W-ORBS	FOS-uni-MOBS	659.60	22.26	6521.51	330.75	9.90	0.61
Iter.:5)	FOS-app-MOBS	443.10	13.09	9541.62	506.47	21.58	1.67	Iter.:5)	FOS-app-MOBS	651.50	20.91	6692.08	362.52	10.29	0.74
	FOS-eff-MOBS	456.10	13.22	**9072.71**	461.40	19.91	1.24		FOS-eff-MOBS	**674.20**	16.96	6394.88	330.91	9.50	0.62
	W-ORBS	696.00	0.00	27677.29	3280.38	39.77	4.71		W-ORBS	925.00	0.00	15093.32	748.07	16.32	0.81

(*continued on next page*)

Table 7 (*continued*)

csv-2	ROS-MOBS	**673.50**	103.54	**6872.09**	1054.63	**10.28**	1.15	prttok2	ROS-MOBS	158.20	14.57	**182.38**	76.60	**1.14**	0.46
(W-ORBS	FOS-uni-MOBS	601.30	12.03	7629.31	365.57	12.70	0.73	(W-ORBS	FOS-uni-MOBS	145.40	8.67	245.81	60.02	1.70	0.43
Iter.:6)	FOS-app-MOBS	622.20	10.67	7726.95	346.49	12.42	0.59	Iter.:5)	FOS-app-MOBS	160.50	8.19	255.50	70.59	1.60	0.48
	FOS-eff-MOBS	629.50	17.63	7530.18	333.35	11.97	0.61		FOS-eff-MOBS	**171.20**	4.51	218.37	42.34	1.28	0.27
	W-ORBS	903.00	0.00	16996.75	837.50	18.82	0.93		W-ORBS	223.00	0.00	601.91	4.78	2.70	0.02
csv-3	ROS-MOBS	**723.60**	71.90	**5956.72**	1068.92	**8.23**	1.34	prttok	ROS-MOBS	**175.40**	9.11	**361.10**	63.76	**2.07**	0.40
(W-ORBS	FOS-uni-MOBS	642.10	22.72	6945.58	436.10	10.85	1.00	(W-ORBS	FOS-uni-MOBS	167.70	6.33	464.75	24.12	2.78	0.21
Iter.:6)	FOS-app-MOBS	684.50	23.41	6791.43	395.51	9.93	0.66	Iter.:5)	FOS-app-MOBS	170.90	5.43	489.20	31.97	2.87	0.21
	FOS-eff-MOBS	699.80	22.44	6759.21	231.33	9.67	0.52		FOS-eff-MOBS	164.00	6.05	429.71	36.88	2.62	0.24
	W-ORBS	1017.00	0.00	13378.98	504.69	13.16	0.50		W-ORBS	210.80	1.83	860.68	151.17	4.08	0.70
esc-1	ROS-MOBS	158.10	11.39	2419.24	128.02	15.35	1.02	replace	ROS-MOBS	**118.70**	10.37	700.25	137.06	5.90	1.01
(W-ORBS	FOS-uni-MOBS	**166.80**	8.45	**2385.50**	148.60	**14.35**	1.26	(W-ORBS	FOS-uni-MOBS	98.20	10.32	**586.78**	78.29	5.96	0.37
Iter.:4)	FOS-app-MOBS	148.00	6.77	2460.76	89.41	16.66	0.93	Iter.:6)	FOS-app-MOBS	113.90	7.13	652.74	72.45	**5.74**	0.63
	FOS-eff-MOBS	148.20	7.85	2400.78	40.82	16.25	0.95		FOS-eff-MOBS	109.70	6.81	662.87	77.21	6.05	0.72
	W-ORBS	239.00	0.00	7875.26	290.58	32.95	1.22		W-ORBS	150.40	7.03	1421.99	211.15	9.42	1.05
esc-2	ROS-MOBS	163.70	7.72	2972.71	147.53	18.21	1.38	sched2	ROS-MOBS	**64.70**	5.95	117.17	24.86	1.80	0.30
(W-ORBS	FOS-uni-MOBS	**166.00**	11.86	2963.51	186.19	**17.93**	1.62	(W-ORBS	FOS-uni-MOBS	61.20	4.24	**97.31**	7.11	**1.60**	0.17
Iter.:5)	FOS-app-MOBS	150.40	4.20	3016.77	122.80	20.08	1.16	Iter.:3)	FOS-app-MOBS	62.50	3.07	113.17	17.45	1.82	0.32
	FOS-eff-MOBS	139.10	7.08	**2923.23**	117.45	21.06	1.17		FOS-eff-MOBS	61.40	3.01	104.53	11.35	1.71	0.21
	W-ORBS	228.00	0.00	9522.98	357.31	41.77	1.57		W-ORBS	86.00	0.00	254.91	2.58	2.96	0.03
esc-3	ROS-MOBS	**209.90**	15.95	2680.11	116.08	**12.85**	1.17	sched	ROS-MOBS	58.20	10.49	88.28	5.91	1.57	0.30
(W-ORBS	FOS-uni-MOBS	204.80	9.34	2626.56	166.92	12.87	1.20	(W-ORBS	FOS-uni-MOBS	53.20	7.81	**65.11**	8.02	**1.27**	0.37
Iter.:5)	FOS-app-MOBS	186.40	9.17	2704.12	90.17	14.55	1.04	Iter.:4)	FOS-app-MOBS	56.70	6.62	83.43	9.62	1.50	0.28
	FOS-eff-MOBS	179.30	8.04	**2617.82**	118.46	14.63	0.87		FOS-eff-MOBS	**58.70**	6.87	83.38	7.78	1.44	0.24
	W-ORBS	309.00	0.00	7030.67	326.03	22.75	1.06		W-ORBS	93.00	0.00	220.23	10.44	2.37	0.11
esc-4	ROS-MOBS	**223.30**	21.39	2635.30	309.91	**11.82**	1.13	totinfo	ROS-MOBS	**68.20**	6.05	**17.79**	0.69	**0.26**	0.03
(W-ORBS	FOS-uni-MOBS	221.30	10.06	**2610.84**	145.66	11.84	1.06	(W-ORBS	FOS-uni-MOBS	67.60	4.45	17.84	0.58	0.27	0.02
Iter.:5)	FOS-app-MOBS	198.30	8.78	2760.42	91.40	13.95	0.86	Iter.:3)	FOS-app-MOBS	65.00	5.14	18.06	0.68	0.28	0.02
	FOS-eff-MOBS	177.60	10.32	2617.47	186.62	14.80	1.49		FOS-eff-MOBS	67.30	7.01	18.24	1.26	0.27	0.04
	W-ORBS	337.00	0.00	7118.56	240.77	21.12	0.71		W-ORBS	98.00	0.00	54.79	1.55	0.56	0.02

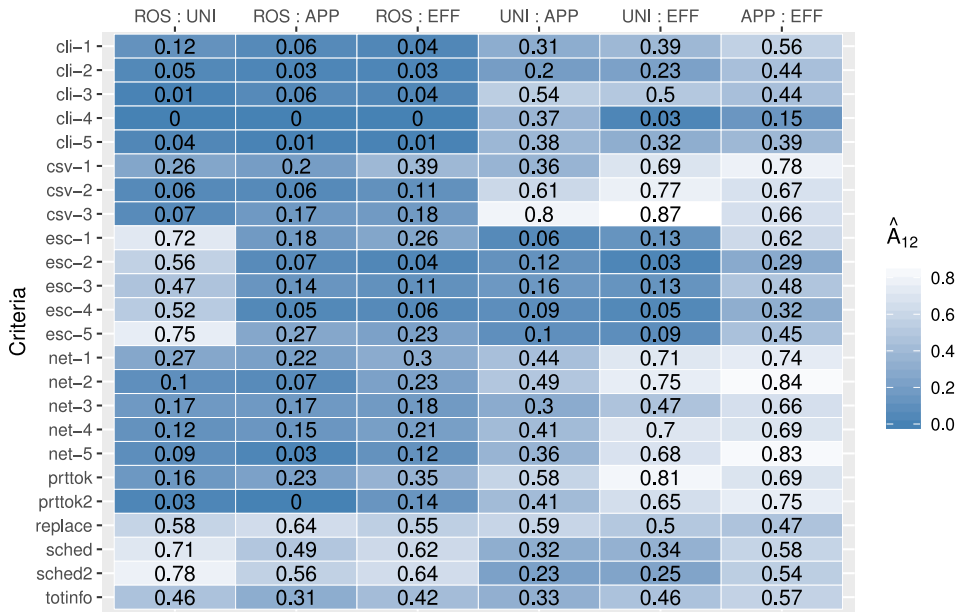

Fig. 8. Vargha-Delaney $\hat{A}_{12}$ on TPD between selection strategies. Each of ROS, APP, EFF, and UNI represents MOBS with rolling operator selection, fixed operator selection using 'applicability', 'effect', and uniform proportion, respectively.

APP, EFF, and UNI appearing in the name of columns represents MOBS with rolling operator selection, fixed operator selection using 'applicability', 'effect', and uniform proportion, respectively. To facilitate the comparison to W-ORBS, we terminate MOBS after the number of iterations W-ORBS requires to complete a slice, and compute TPD values until that iteration. In column *P*: *Q*, the $\hat{A}_{12}$ statistic is the probability that a score sampled at random from the first population, *P*, will be greater than a score sampled at random from the second, *Q*. Columns starting with ROS contain more dark blue cells than others, indicating that ROS tends to show higher efficiency (marked by lower TPD). Variants of FOS show little difference from each other. Based on these results, we answer **RQ3** by concluding that the Rolling Operator Selection (ROS) has the best performance.

7.4. Comparison between MOBS and W-ORBS

We compare MOBS to W-ORBS. Table 7 shows the means and standard deviations for the wall clock execution time, the number of deleted lines, and per-deletion efficiency for W-ORBS and MOBS using the four different operator selection strategies. The largest number of deleted lines, the shortest execution time, and the lowest TPD values among the four strategies are typeset in bold. The box plots shown in Fig. 9 show the distributions of these values at the end of each iteration for four slicing criteria. Note that the *y*-axis for the execution time box plots on the right use a logarithmic scale.

ROS-MOBS (found to be the most efficient variant in RQ3) deletes 63% to 83% of lines deleted by W-ORBS in 27%–45% of the time required by W-ORBS. The worst case efficiency of ROS-MOBS is observed in sched, whose TPD value is highest when compared to that of W-ORBS. For this program ROS-MOBS only deletes 63% of the lines deleted by W-ORBS, while taking only 40% of its execution time. However, even in this worst case, the trade-off is better than linear. Based on these results, we answer **RQ4** by concluding that ROS-MOBS can be both effective and efficient, being capable of deleting an average of 69% of the number of lines deleted by W-ORBS, while requiring only 36% of wall clock execution time required by W-ORBS.

7.5. Qualitative analysis of the slices

This section answers **RQ5** with a qualitative analysis of the slices. We investigate the differences between the slices generated by W-ORBS, VSM-ORBS, and MOBS. We omit LDA-ORBS from the analysis as the results hitherto clearly suggest it is not as effective as VSM-ORBS.

7.5.1. Similarity between W-ORBS and MOBS slices

Fig. 10 shows our modified Jaccard similarity computed between the results of ROS-MOBS and W-ORBS for two slicing criteria, `net-3` and `replace`. The values on the leftmost column represent the similarity between the slice produced by W-ORBS and ten trials of MOBS; values above the diagonal represent the similarity between the ten trials of MOBS where darker squares indicate greater similarity between the two slices. Except for small fluctuations, slices generated by MOBS are much more similar to each other, than they are similar to the slices generated by W-ORBS. This implies that, despite stochasticity, slices generated by MOBS share common patterns that are different from those generated by W-ORBS. We next consider whether lexical approximation of dependence is particularly effective or ineffective against specific types of statements.

7.5.2. Characteristics of slices

Table 8 shows the top thirty lines that are most frequently found in slices by VSM-ORBS but not in those by W-ORBS. We observe the following patterns that make it difficult for VSM-ORBS to delete these lines:

- *Multi-line Statements*: Lexical deletion operators do not try to delete consecutive lines. Thus, they may attempt to delete one line from a multi-line statement, raising compilation errors. This case includes lines ending with a left bracket that marks the beginning of a compound statement, such as `if (it.hasNext()){`.
- *Declarations*: As can be seen in Section 7.1, the lexical deletion operators cannot delete as many lines as W-ORBS. Consequently, they often fail to delete all uses of a variable before attempting to delete its declaration. Similarly, they often fail to

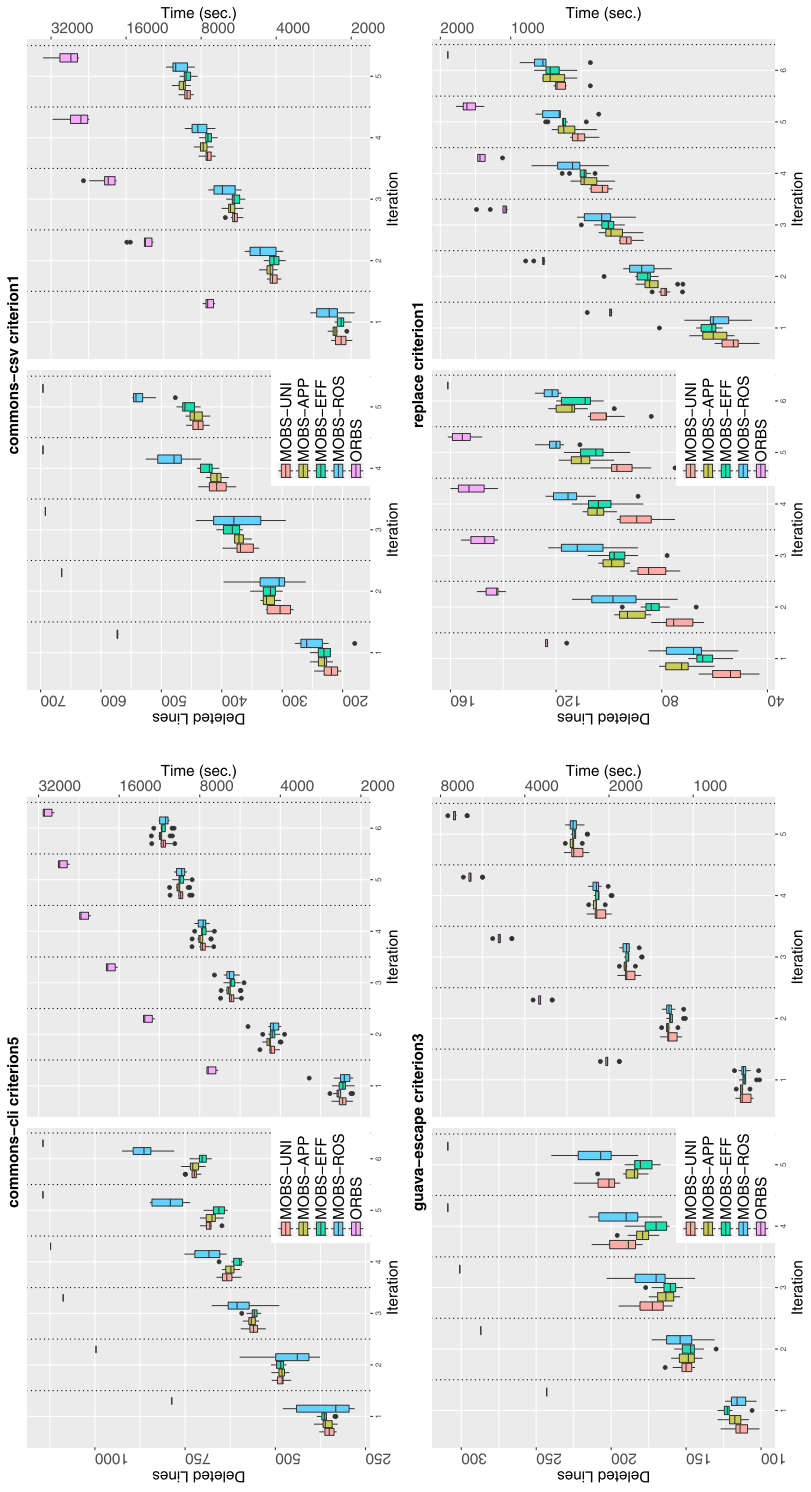

Fig. 9. Box plots of execution time and number of deleted lines for W-ORBS and variants of MOBS over 10 repeated runs.

delete the body of a method before attempting to delete the method declarations. This means that declaration statements are harder for VSM-ORBS to delete than for W-ORBS.

- *Frequent Lexemes*: certain lexemes occur frequently throughout the code in many different, and potentially unrelated, semantic contexts. They may be lexically similar to each other, but this does not necessarily mean that there also exists dependence. For example, `return this;` is a lexeme that can be found in many different methods: attempting to delete all instances of `return this;` is most likely to fail, even if the specific instance under consideration can actually be deleted.

We investigate how capable W-ORBS, VSM-ORBS, and MOBS are at deleting lines having these characteristics. Fig. 11 contains box plots that show how many lines of interest (i.e., either lines that are part of multi-line statements or declarations) can be deleted by each technique. Fig. 12 shows the results of a similar analysis for frequent lexemes.

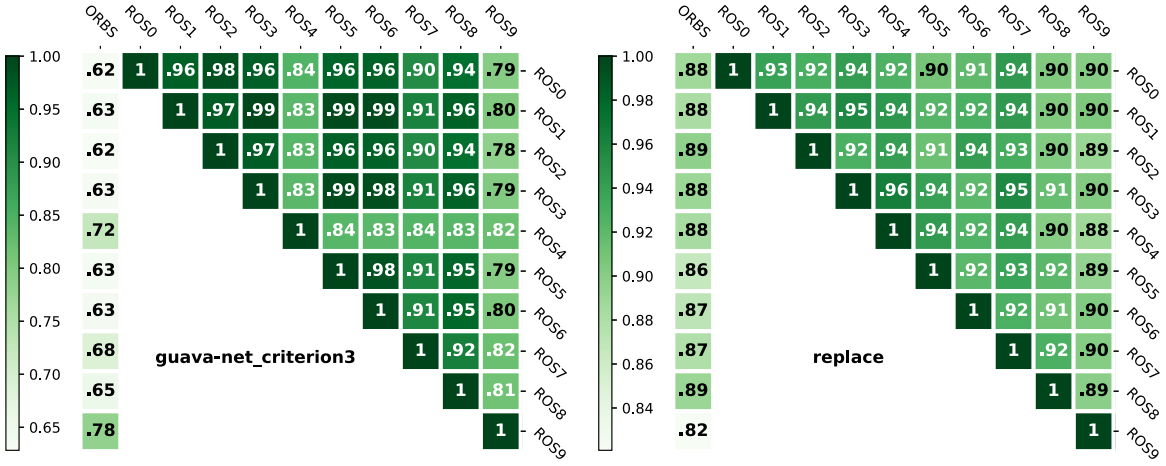

Fig. 10. Jaccard similarity between ROS-MOBS and W-ORBS for `net-3` and `replace`.

Table 8
Lines that are retained in VSM-ORBS slices but not in W-ORBS slices.

Code line (top thirty with frequency)	Freq.
`}`	10,228
`import static com.google.common.base.Preconditions.checkNotNull;`	212
`checkArgument(`	168
`throw new IllegalArgumentException(`	144
`public String toString() {`	140
`return (TRUE);`	136
`currentOption = null;`	135
`eatTheRest = true;`	120
`return (this);`	104
`if (it.hasNext()) {`	92
`{`	92
`public int hashCode() {`	84
`import java.util.Iterator;`	80
`return (token);`	80
`hostPortString);`	76
`return (s);`	75
`return (false);`	73
`quote = true;`	72
`opt = Util.stripLeadingHyphens(opt);`	68
`return (true);`	64
`this.option = option;`	60
`import java.util.List;`	60
`if (opt == null) {`	60
`}else if (matchingOpts.size() > 1) {`	60
`else{`	60
`import static com.google.common.base.Preconditions.checkArgument;`	60
`return (dest);`	60
`dest[3] = '%';`	60
`import java.util.ArrayList;`	56
`import javax.annotation.Nullable;`	56

The box plots in Fig. 11 show that, for all slicing criteria, VSM-ORBS deletes the fewest lines of interest, followed by MOBS, and W-ORBS, which deletes the most lines of interest. The results suggest that syntactic structures in source code presents challenges to the lexical deletion operators, while the window deletion used by W-ORBS can circumvent this challenge.

Analysis of frequent lexemes requires a more subtle approach, as we cannot anticipate all such lexemes. Instead, we posit that these lexemes are more likely to consist of stop words, as our characterising definition of frequent lexemes is a lexeme that can appear in many different contexts. Non-stop words (i.e., solution-domain identifiers) are more likely to be bound to specific local contexts. Consequently, we compare the average number of non-stop word tokens per deleted line in Fig. 12.

While there is variance in the difference, VSM-ORBS deletes more non-stop word tokens per deleted line than W-ORBS. One implication of this is that lexical deletion operators may find it more difficult to delete irrelevant control structures, as deleting them would require deleting lines with frequent lexemes that are related to control flow, such as `return;` or `} else {`. However, as long as lines that mostly consist of unique identifiers are deleted, the resulting slices may still be useful in cases where the user is interested more in understanding the dependencies between individual lines than in slicing out entire control flow structures.

7.6. Scalability

To investigate how lexical dependency scales to both larger and multi-lingual systems, we consider the open-source project mis-

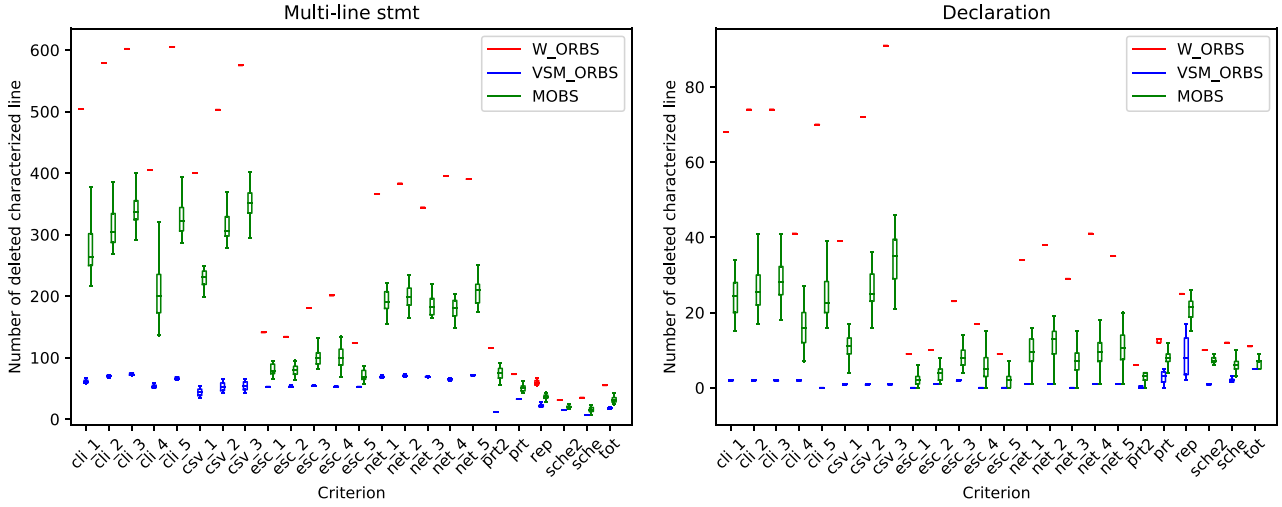

Fig. 11. Number of deleted characterized lines.

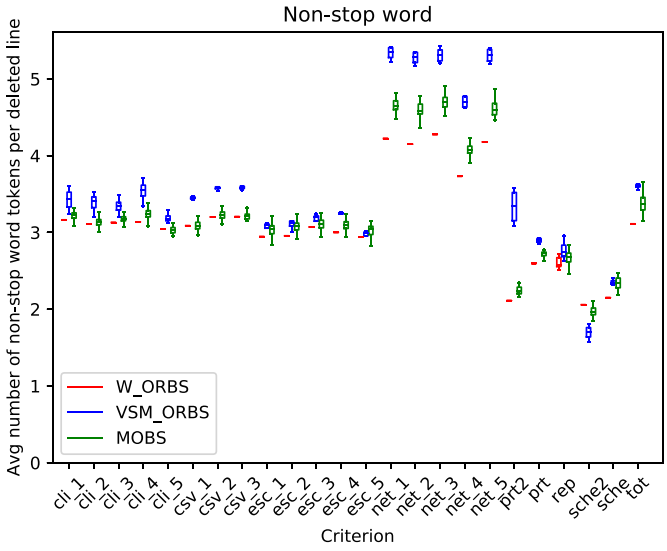

Fig. 12. Average number of non-stop word tokens on a deleted line.

aka [6] as a benchmark program. Misaka is a CFFI-based binding for Hoedown, a fast markdown processing library written in C. The project consists of ten C files from Hoedown, which performs text parsing, and five Python files that wrap the C functions to produce a Python module. Misaka has a rich test suite containing 92 test cases written in Python that focus on evaluating the linkage between the Python and C functions, rather than the Hoedown library itself. The test suite consists of two unit tests that test the input arguments and 39 integration tests that test the binding of the C functions to Python methods. The remaining 41 system-level tests involve 41 different markdown text files and their corresponding HTML files.

We consider six slicing criteria for misaka, in an attempt to cover as diverse a set of functionalities as possible. We select slicing criteria in the C code, which are eventually reached from the Python test scripts through the CFFI binding. The first slicing criterion (crit-1) involves a variable tracking the index of the beginning of each line in a buffer while rendering a regular markdown document. The second slicing criterion (crit-2) targets the size of the text to render. The third slicing criterion (crit-3) is a variable containing the maximum size of the custom stack before it is changed by a method that grows the stack to a given size. The fourth slicing criterion (crit-4) targets the size of misaka's renderer object while allocating a regular HTML renderer. The fifth slicing criterion (crit-5) is a temporary variable which discriminates the starting index of the row from the padding when parsing a markdown table row. The last slicing criterion (crit-6) is the index of the beginning of a markdown link in the method that calculates the index of the end of the markdown link.

We ran W-ORBS, VSM-ORBS, and LDA-ORBS on the six slicing criteria using a threshold of 0.9 for γ with both DVSM and DLDA, and $n = 500$ topics for DLDA. We also ran ROS-MOBS ten times for each slicing criteria. The results for VSM-ORBS and LDA-ORBS when compared to W-ORBS shows a similar trend to those of the previous experiments. On average for all slicing criterion, VSM-ORBS and LDA-ORBS slice the code 2.82 times and 2.31 times faster than W-ORBS while they delete 32.1% and 33.5% of lines that W-ORBS could delete, respectively.

Table 9 shows the result comparing W-ORBS and ROS-MOBS, and box plots in Fig. 13 show the ratio between two. In Fig. 13, the red box plot on the left represents how many times ROS-MOBS run faster compare to W-ORBS, and the blue box plot on the right represents the ratio of the number of deleted lines by ROS-MOBS to the number of deleted lines by W-ORBS. According to the result, crit-1 shows good performance from MOBS, where it deletes 71% of the lines deleted by W-ORBS while executing 2.6 times faster. MOBS shows poor performance on crit-4 and crit-5. For crit-5 it is only 1.4 times faster while for crit-4 the timing is essentially the same with the average speed up of 1.01. For these slices it deletes 48% and 57% of the lines deleted by W-ORBS, respectively. We investigated this difference in performance. The main cause is the size of the slice. For slicing criterion crit-4, the size of the renderer object calculated by simply calling `sizeof` method on the object, has no control or data dependence between the surrounding source code. The dependency chains reaching crit-4 are also simple and shallow, making most of the code easy to delete in early stages of W-ORBS and MOBS. The small number of remaining lines reduce MOBS's advantage of fewer deletion attempts on a single line when compared to W-ORBS. Similarly, the dependence of crit-5 is limited. It focuses solely on parsing the markdown table, which is a local function of the program, and has little dependence on other parsing methods in the program; thus, its slice size is the second smallest among all slicing criteria. On the other hand,

[6] https://misaka.61924.nl

Table 9
Statistics comparing ROS-MOBS and W-ORBS on the six misaka slicing criteria. The data for ROS-MOBS is an average of ten trial runs. Misaka has a total of 5125 lines of code.

Criterion	Strategy	Iteration	Compile	Execute	Deleted lines	Time
crit-1	ROS-MOBS	5.9	17,683	5135	2209	34,890
	W-ORBS	6.0	42,292	16,027	2950	92,305
crit-2	ROS-MOBS	5.1	13,561	2937	2145	22,765
	W-ORBS	6.0	25,612	8849	3880	50,970
crit-3	ROS-MOBS	5.4	13,316	3133	2284	23,930
	W-ORBS	6.0	23,978	8448	3956	47,203
crit-4	ROS-MOBS	4.9	10,579	1602	2626	10,953
	W-ORBS	6.0	12,700	2787	4642	10,683
crit-5	ROS-MOBS	4.8	11,697	2019	2140	18,036
	W-ORBS	6.0	15,370	3723	4465	24,704
crit-6	ROS-MOBS	5.4	16,231	3994	2488	30,922
	W-ORBS	6.0	26,878	9163	3794	50,762

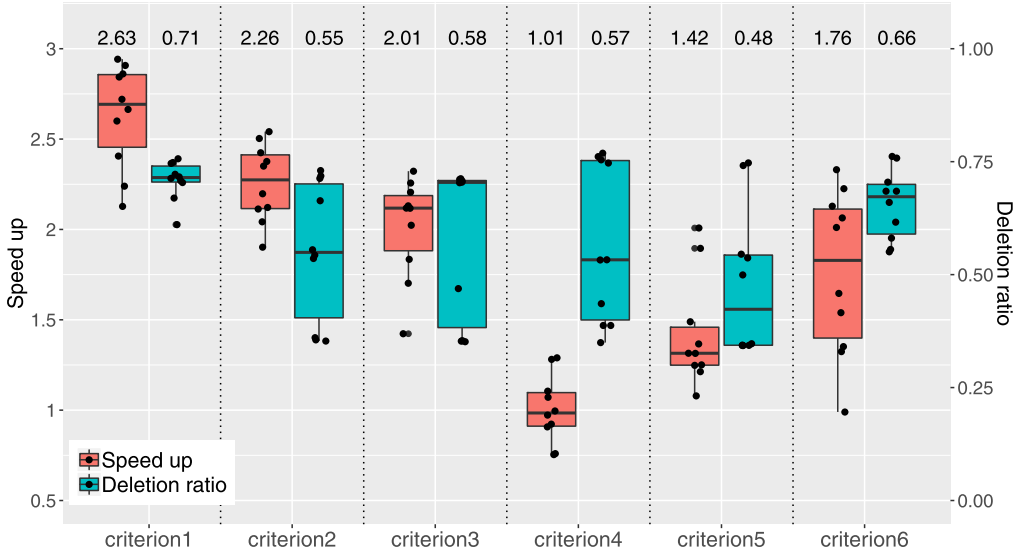

Fig. 13. Ratios comparing the number of deleted lines and time taken by ROS-MOBS and W-ORBS .

criterion crit-1 targets the central logic of the Hoedown library. It occurs inside a while loop calculating the beginning index of every line of the document. Thus, the dependence chains weave through much of the code, and the final slice is the largest over all slicing criteria. A large number of remaining lines enhances the advantage of MOBS over W-ORBS, making the slice much faster than W-ORBS. The other three slicing criteria (crit-2, crit-3, and crit-6) show a similar trend. Their slices are smaller than the slice of crit-1, but larger than the slices of crit-4 and crit-5. Both crit-2 and crit-6 involves greater dependence with other parsing methods than crit-5. Finally, slicing criterion crit-3 targets a function that increases the stack's size limit, which is used as a data structure to buffer parts of a document.

Table 10 shows examples of successful multi-lingual deletion by the lexical deletion operators. The first column shows which operator was used; both of DvsMand DLDAare successfully applied to the code lines in the last row. DvsMsuccessfully deletes three lines: two from 'callbacks.py' and one from 'html.c'. Terms '`table`', '`align`', and '`left`' are shared among the three causing them to be considered similar by the Vector Space Model. DLDAsuccessfully delete lines from 'api.py', 'document.c', and 'html_smartypants.c', together. In this case, the terms '`hoedown`', '`buffer`', and '`text`' are shared among three lines making them similar under LDA. Both DvsMand DLDAsuccessfully deletes the lines `result = renderer.blockhtml(text)` from 'callbacks.py' and `renderer->blockhtml = NULL;` from 'html.c', together. This result exemplifies that our new lexical deletion operators can capture (an approximation to) inter-language dependence in a multi-lingual program.

7.7. Threats to validity

This section considers three threats to the validity of our experiments: external validity, internal validity, and construct validity. To begin with, external validity consider how well our results generalize to other environments. The subjects studied, shown in Table 1, include imperative and object-oriented codes of modest size. It is possible that our technique is not effective when larger programs or programs written in other languages are considered. Mitigating both of these threats is the previous application of ORBS to larger programs written in a range of programming languages. In fact this is one of ORBS strengths. The more serious external threat that larger programs bring is that the naming would become *cluttered* resulting in a lowering of the effectiveness of the IR based lexical approximations. In general, IR systems scale to very large corpora where then often perform better in the presence of more data.

Next, internal validity is the causal effect of the explanatory variables on the response variables. The use of ORBS, which empirically identifies the exact dynamic dependencies provides an excellent bellwether for assessing the lexical approximation of pro-

Table 10
Example successful multi-lingual deletions by the lexical deletion operators.

Operator	File-name:Line-num	Code line
DVSM	callbacks.py:97	`elif align_bit == TABLE_ALIGN_LEFT:`
	callbacks.py:98	`align = 'left'`
	hoedown/html.c:393	`case HOEDOWN_TABLE_ALIGN_LEFT:`
DLDA	api.py:29	`lib.hoedown_buffer_puts(ib, text.encode('utf-8'))`
	hoedown/document.c:2490	`hoedown_buffer_free(text);`
	hoedown/html_smartypants.c:195	`hoedown_buffer_putc(ob, text[0]);`
DVSM, DLDA	callbacks.py:125	`result = renderer.blockhtml(text)`
	hoedown/html.c:635	`renderer->blockhtml = NULL;`

gram dependence. When a lexical operator successful deletes a set of lexically related lines the ORBS's framework ensures that these lines are not semantically related to the slicing criterion and thus they can all be safely removed. In the other direction, just because one of a set of lexically similar lines *can not* be deleted, does not mean that other members of this set can not be deleted. Therefore we have strong evidence when a deletion is accepted, but the approximation is more suspect when a deletion is rejected.

The final threat considered is the threat to construct validity, which considers how well our approach measures what it claims. In our experiments construct validity is not a significant issue because we can directly measure dynamic dependence by running the program using the given test suite.

8. Related work

Lexical analysis, especially techniques borrowed from Information Retrieval, have been widely studied and applied in software engineering. For example, LDA has been applied to program comprehension (Binkley et al., 2014b) and traceability recovery (Panichella et al., 2013); Vector Space Models have been applied to fault localisation, based on the intuition that bug reports and the faulty program code may tend to be lexically similar (Saha et al., 2013; Le et al., 2014; 2015; Wang et al., 2015). More broadly, application of Natural Language Processing (NLP) techniques to source code has been studied in the context of subjects such as the natural language model of source code (Hindle et al., 2012), coding conventions (Allamanis et al., 2014), and code snippet recommendation (Campbell and Treude, 2017). As far as we know, MOBS is the first approach to program slicing and, more generally, dependence analysis that exploits lexical information in program source code.

Since its introduction by Weiser in the 1970s (Weiser, 1979), program slicing has been widely studied and developed (Anderson and Teitelbaum, 2001; Horwitz et al., 1990; Amtoft and Banerjee, 2016; Hur et al., 2014). Static program slicing (Weiser, 1981) produces slices that are correct for all possible program executions, whereas dynamic slicing aims to tailor slices to a particular set of program inputs (Korel and Laski, 1988).

Many flavours of static slicing algorithms attempt to reduce the size of the resulting slice. Incremental Slicing (Orso et al., 2001) allows selection of the type of data dependencies that are considered while slicing. Stop-list slicing (Gallagher et al., 2006) allows the programmer to define variables that are not of interest, information that is subsequently used to purge the dependence graph before computing slices, resulting in smaller slices. Barrier Slicing (Krinke, 2003) allows the programmer to specify which parts of the program can and cannot be traversed while constructing the slice. A barrier is specified with a set of nodes or edges of the program's program dependence graph that cannot be passed during the graph traversal, also resulting in a focused and thus smaller slice.

Amorphous Slicing (Harman and Danicic, 1997) is an approach that aims to preserve the semantics of the program, but not its syntax. Amorphous slices use program transformation to simplify programs, preserving the semantics of the program with respect to the slicing criterion. In contrast MOBS (and ORBS) only transform a program using deletion.

Korel and Laski (1988, 1990) considered several algorithms to compute dynamic slices based on their definition. In contrast, most later work on dynamic slicing 'defines' dynamic slicing based on the algorithms used to compute it (e.g., Agrawal and Horgan, 1990 and DeMillo et al., 1996). Although many research prototypes and approaches exist (Beszedes et al., 2001; 2006; Mund and Mall, 2006; Szegedi and Gyimóthy, 2005; Zhang and Gupta, 2004; Zhang et al., 2007; Barpanda and Mohapatra, 2011), all these approaches are for a single specific programming language and requires additional analysis for the interface between languages to support multi-language programs.

Finally, union slicing (Beszédes et al., 2002) is also related to ORBS. Union slicing approximates a static slice by unioning dynamic slices obtained using a set of inputs. However, union slicing inherits the critical difference between dynamic and observation-based slicing: dependencies considered by union slicing are dynamically *occurring* (but statically determined) dependencies, rather than dynamically *observed* dependencies as in ORBS. Moreover, unioning of slices does not necessarily lead to correct slices (De Lucia et al., 2003), whereby ORBS computes dynamic slices for multiple criteria without unioning.

MOBS builds upon Observation-Based Slicing (ORBS), a type of dynamic slicing: it only preserves program dependencies that are observable (Binkley et al., 2014a) via program execution. The dynamic nature of ORBS means it under-approximates the semantics of program dependence, limited by the test suites used as input. However, accepting deletions of source code lines based on purely dynamic observation has its own benefits, such as being able to handle dependencies that no static slicers can cope with (Binkley et al., 2015), slicing multi-lingual systems (Binkley et al., 2015), and slicing languages with highly unconventional program semantics such as Picture Description Language (Yoo et al., 2017). While MOBS and ORBS uses deletions of source code lines, a later variant (Gold et al., 2017; Binkley et al., 2019) represents source code as a tree structure and the proceeds to delete subtrees. Binkley et al. (2014a) also introduced a parallel ORBS. Rather than applying window deletion successively, the parallel version applies all deletion operators of different window size in parallel and then selects the largest deletion that succeeds. By definition, MOBS is very much parallelisable. Furthermore, if the deletion operators MOBS uses subsume all the window deletion operators used by W-ORBS, parallel MOBS is a super set of parallel W-ORBS. Theoretically, its worst case performance will match that of parallel W-ORBS, but it has the potential to opportunistically take advantage of successful lexical deletions. In this paper, we focus evaluating the impact of lexical dependence; thus, we leave the study of parallelisation's impact to future work.

The notion of deleting parts of a program or inputs also features prominently in Delta Debugging (Zeller, 1999; Cleve and Zeller, 2000; Zeller and Hildebrandt, 2002). Some variants of delta debugging try to reduce the cost of the original Delta Debugging by exploiting language syntax and semantics. For example, Hierarchical Delta Debugging (Misherghi and Su, 2006) exploits tree structures providing a tree-based Delta Debugging approach. Delta (McPeak et al., 2006), a well known implementation of Delta Debugging, uses a separate tool to flatten the tree structures in source code, before applying delta debugging. Regehr et al. (2012) exploit the syntax and semantics of C for four delta-debugging based algorithms to minimize C programs that trigger compiler bugs. Coarse Hierarchical Delta Debugging (Hodován et al., 2017) is a recently introduced variant of Hierarchical Delta Debugging that filters out tree nodes that are not allowed to be deleted by the grammar of the language, thereby speeding up Hierarchical Delta Debugging.

Finally, Jiang et al. (2014) introduced a forward dynamic slicing approach similar to ORBS: their technique mutates the value of the variable at the location of the slicing criterion, and subsequently observes the computed values in the state trajectory. The dynamic slice consists of all statements for which the computed values have changed compared to the trajectory of the original program.

9. Conclusion

Given program slicing's wide range of applications, an efficient, language-independent slicing technique can bring significant benefits to developers. The small increase in slice size produced by MOBS is likely acceptable if it is accompanied by a significant decrease in slicing time.

This paper makes two novel technical contributions. First, we present a novel generalisation of observational slicing that can take advantage of a wide range of deletion operators rather than the original algorithm's use of only one, the deletion window. Second, we introduce lexical deletion operators that exploit lexical similarities between source code lines to improve the efficiency of ORBS. MOBS is the resulting observational slicer that uses multiple deletion operators including the existing deletion window operators and the newly-introduced lexical deletion operators.

The results of our empirical evaluation of MOBS show a significantly improve in efficiency over W-ORBS, which is based solely on window deletion: MOBS deletes approximately 69% of the lines deleted by W-ORBS, while taking only about 36% the wall clock execution time. Furthermore, ROS's ability to learn the relative applicability of different operators dynamically during slicing produced the best result for MOBS. Finally, we qualitatively considered the lexically deletable lines of code and the scalability of MOBS using the multi-lingual open-source project misaka, which include both Python and C code.

These results show that using a lexical approximation of dependence is viable. Future work will include investigating the impact of involving reserved words and comments in the lexical similarity computations as well as the impact of increasing the size of the test corpus and programs.

Acknowledgements

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. 2017M3C4A7068179). Dr. Binkley is supported by NSF grant 1626262.

References

Agrawal, H., DeMillo, R.A., Spafford, E.H., 1993. Debugging with dynamic slicing and backtracking. Softw. Pract. Exp. 23 (6), 589–616.
Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. In: Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation (PLDI), pp. 246–256.
Allamanis, M., Barr, E.T., Bird, C., Sutton, C., 2014. Learning natural coding conventions. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 281–293.
Amtoft, T., Banerjee, A., 2016. A theory of slicing for probabilistic control flow graphs. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 180–196.
Anderson, P., Teitelbaum, T., 2001. Software inspection using CodeSurfer. Workshop on Inspection in Software Engineering (CAV 2001).
Barpanda, S.S., Mohapatra, D.P., 2011. Dynamic slicing of distributed object-oriented programs. IET Software 5 (5), 425–433.
Beszédes, Á., Faragó, C., Szabó, Z.M., Csirik, J., Gyimóthy, T., 2002. Union slices for program maintenance. In: Proc. of the 18th Intl. Conf. on Software Maintenance (ICSM), pp. 12–21.
Beszedes, A., Gergely, T., Gyimóthy, T., 2006. Graph-less dynamic dependence-based dynamic slicing algorithms. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM), pp. 21–30.
Beszedes, A., Gergely, T., Szabó, Z.M., Csirik, J., Gyimothy, T., 2001. Dynamic slicing method for maintenance of large C programs. In: Proc. of the 5th Conf. on Software Maintenance and Reengineering, pp. 105–113.
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, 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, 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. 24 (5), 3077–3113.
Binkley, D., Heinz, D., Lawrie, D., Overfelt, J., 2014b. Understanding LDA in source code analysis. In: Proceedings of the 22Nd International Conference on Program Comprehension. ACM, pp. 26–36.
Binkley, D.W., 1998. The application of program slicing to regression testing. Inf. Softw. Technol. Spec. Issue Progr. Slicing 40 (11 and 12), 583–594.
Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent Dirichlet allocation. J. Mach. Learn. Res. 3 (Jan), 993–1022.
Campbell, B.A., Treude, C., 2017. NLP2Code: code snippet content assist via natural language tasks. CoRR abs/1701.05648.
Cleve, H., Zeller, A., 2000. Finding failure causes through automated testing. In: Intl. Workshop on Automated Debugging, pp. 254–259.
DeMillo, R.A., Pan, H., Spafford, E.H., 1996. Critical slicing for software fault localization. In: Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA), pp. 121–134.
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.
Gallagher, K.B., Binkley, D., Harman, M., 2006. Stop-list slicing. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM), pp. 11–20.
Gallagher, K.B., Lyle, J.R., 1991. Using program slicing in software maintenance. IEEE Trans. Software Eng. 17 (8), 751–761.
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, pp. 547–558.
Goldberg, D.E., 1989. Genetic algorithms in search, optimization & machine learning. Addison-Wesley, Reading, MA.
Harman, M., Danicic, S., 1997. Amorphous program slicing. In: 5th IEEE International Workshop on Program Comprenhesion (IWPC'97). IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.
Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P., 2012. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 837–847.
Hodován, R., Kiss, Á., Gyimóthy, T., 2017. Coarse hierarchical delta debugging. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 194–203. doi:10.1109/ICSME.2017.26.
Horwitz, S., Reps, T., Binkley, D.W., 1990. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12 (1), 26–61.
Hur, C.-K., Nori, A.V., Rajamani, S.K., Samuel, S., 2014. Slicing probabilistic programs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 133–144.
Jiang, S., Santelices, R., Grechanik, M., Cai, H., 2014. On the accuracy of forward dynamic slicing and its effects on software maintenance. In: Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM), pp. 145–154.
Korel, B., Laski, J., 1988. Dynamic program slicing. Inf. Process. Lett. 29 (3), 155–163.
Korel, B., Laski, J., 1990. Dynamic slicing in computer programs. J. Syst. Softw. 13 (3), 187–195.
Korel, B., Rilling, J., 1998. Program slicing in understanding of large programs. In: 6th IEEE International Workshop on Program Comprenhesion (IWPC'98). IEEE Computer Society Press, Los Alamitos, California, USA, pp. 145–152.

Krinke, J., 2003. Barrier slicing and chopping. In: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003). IEEE Computer Society Press, Los Alamitos, California, USA, pp. 81–87.
Le, T.B., Thung, F., Lo, D., 2014. Predicting effectiveness of IR-based bug localization techniques. In: 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3–6, 2014, pp. 335–345.
Le, T.-D.B., Oentaryo, R.J., Lo, D., 2015. Information retrieval and spectrum based bug localization: Better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 579–590.
De Lucia, A., Harman, M., Hierons, R., Krinke, J., 2003. Unions of slices are not slices. In: European Conference on Software Maintenance and Reengineering (CSMR 2003), pp. 363–367.
McPeak, S., Wilkerson, D. S., Goldsmith, S., 2006. Delta(http://delta.tigris.org).
Misherghi, G., Su, Z., 2006. HDD: hierarchical delta debugging. In: Proc. of the 28th Intl. Conf. on Software Engineering (ICSE), pp. 142–151.
Mitra, M., Chaudhuri, B., 2000. Information retrieval from documents: a survey. Inf. Retr. Boston 2 (2), 141–163.
Mund, G., Mall, R., 2006. An efficient interprocedural dynamic slicing method. J. Syst. Softw. 79 (6), 791–806.
Orso, A., Sinha, S., Harrold, M.J., 2001. Incremental slicing based on data-dependences types. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001). IEEE Computer Society Press, Los Alamitos, California, USA, pp. 158–167.
Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A., 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 522–531.
Ragkhitwetsagul, C., Krinke, J., Clark, D., 2018. A comparison of code similarity analysers. Empir. Softw. Eng. 23 (4), 2464–2519.
Rajaraman, A., Ullman, J.D., 2011. Mining of Massive Datasets. Cambridge University Press, New York, NY, USA.
Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X., 2012. Test-case reduction for C compiler bugs. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 335–346.
Saha, R.K., Lease, M., Khurshid, S., Perry, D.E., 2013. Improving bug localization using structured information retrieval. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, pp. 345–355.
Salton, G., Wong, A., Yang, C.S., 1975. A vector space model for automatic indexing. Commun. ACM 18 (11), 613–620.
Singhal, A., 2001. Modern information retrieval: a brief overview. IEEE Data Eng. Bull. 24 (4), 35–43.
Szegedi, A., Gyimóthy, T., 2005. Dynamic slicing of Java bytecode programs. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM), pp. 35–44.
Vargha, A., Delaney, H.D., 2000. A critique and improvement of the "CL" common language effect size statistics of mcgraw and wong. J. Educ. Behav. Stat. 25 (2), pp.101–132.
Wang, Q., Parnin, C., Orso, A., 2015. Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12–17, 2015, pp. 1–11.
Weiser, M., 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. University of Michigan, Ann Arbor, MI Ph.D. thesis.
Weiser, M., 1981. Program slicing. In: Proc. of the 5th Intl. Conf. on Software Engineering, pp. 439–449.
Yoo, S., Binkley, D., Eastman, R., 2017. Observational slicing based on visual semantics. J. Syst. Softw. 129, 60–78.
Zeller, A., 1999. Yesterday, my program worked. today, it does not. Why? In: European Software Engineering Conf. and Foundations of Software Engineering, pp. 253–267.
Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. 28 (2), 183–200.
Zhang, X., Gupta, N., Gupta, R., 2007. A study of effectiveness of dynamic slicing in locating real faults. Empir. Softw. Eng. 12 (2).
Zhang, X., Gupta, R., 2004. Cost effective dynamic program slicing. In: Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation, pp. 94–106.

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, 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.

Nicolas Gold is an Associate Professor in 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 includes program analysis and applications of computer music in e-health.

Syed Islam currently works as an Information Architect in the industry. Previously, he was a Senior Lecturer at the University of East London. Dr. Islam studied his PhD in program analysis at CREST, Centre for Research on Evolution, Search and Testing at University College London and was awarded his PhD in 2014. His current research interests include data management, program analysis, software metrics and intelligent systems.

Jens Krinke is Associate Professor in the Software Systems Engineering Group at the University College London, where he is Director of CREST, the Centre for Research on Evolution, Search, and Testing. His main focus is software analysis for software engineering purposes. His current research interests include software similarity, modern code review, and mutation testing. He is well known for his work on program slicing and clone detection.

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.