Noname manuscript No.
(will be inserted by the editor)

A Comparison of Tree- and Line-Oriented Observational Slicing

David Binkley · Nicolas Gold · Syed Islam · Jens Krinke · Shin Yoo

the date of receipt and acceptance should be inserted later

Abstract Observation-based slicing and its generalization observational slicing are recently-introduced, language-independent dynamic slicing techniques. They both construct slices based on the dependencies observed during program execution, rather than static or dynamic dependence analysis. The original implementation of the observation-based slicing algorithm used lines of source code as its program representation. A recent variation, developed to slice modelling languages (such as Simulink), used an XML representation of an executable model. We ported the XML slicer to source code by constructing a tree representation of traditional source code through the use of srcML.

This work compares the tree- and line-based slicers using four experiments involving twenty different programs, ranging from classic benchmarks to million-line production systems. The resulting slices are essentially the same size for the majority of the programs and are often identical. However, structural constraints imposed by the tree representation sometimes force the slicer to retain enclosing control structures. It can also "bog down" trying to delete single-token subtrees. This occasionally makes the tree-based slices larger and the tree-based slicer slower than a parallelised version of the line-based slicer. In addition, a Java versus C comparison finds that the two languages lead to similar slices, but Java code takes noticeably longer to slice. The initial experiments suggest two improvements to the tree-based slicer: the addition of a size threshold, for ignoring small subtrees, and subtree replacement. The former enables the slicer to run 3.4 times faster while producing slices that are only about 9% larger. At the same time the subtree replacement reduces size by about 8–12% and allows the tree-based slicer to produce more natural slices.

D. Binkley
Loyola University Maryland, Baltimore, MD, USA

N. Gold, J. Krinke, S. Islam
Department of Computer Science, University College London, UK

S. Yoo
KAIST, Daejeon, Republic of Korea

1 Introduction

Observation-based slicing and its generalisation observational slicing are two recently introduced dynamic program slicing techniques that handle two long-standing challenges in program slicing: slicing systems consisting of components written in multiple programming languages, and slicing systems that include binary components or libraries (Binkley et al, 2014; Gold et al, 2017). Observation-based slicing replaces the complex static and dynamic analysis required by existing slicing techniques with the *observation* of program behaviour via execution of a given test suite. Operationally, it speculatively deletes parts of the code, builds, executes, and then observes the program's behaviour: it only commits to a deletion if the desired behaviour is still observed. All of these steps can be constructed using the existing build tool-chain, obviating the need to replicate much of the compiler's infrastructure (e.g., parsing the code) for each specific language.

While similar to dynamic slices in their reliance on a selected set of inputs, observation-based slices are based on *observed dependencies*, rather than the *statically determined* but dynamically *witnessed* dependencies used by dynamic slicers. That is, a dynamic slice contains a statement if a (statically determined) dependence is witnessed during some execution. By contrast, an observation-based slice contains a statement if its deletion has an observable effect on the slicing criterion.

Traditional and observation-based slicing compare projections of state trajectories, i.e., values of variables during the execution, and require that the projections are the same for the original and the sliced program. The notion of state trajectories and variable values are not appropriate for slicing modelling languages or picture description languages. Thus, generalised *observational slicing* (Gold et al, 2017) allows any form of observation.

The original implementation of observation-based slicing processed traditional source code at the line-of-text level. A subsequent implementation enhanced the algorithm for observation-based slicing to observational slice tree-represented modelling languages (Gold et al, 2017). This second implementation suggests the slicing of traditional source code as a(n XML) tree. We use srcML (Collard, 2005) to transform source code from lines of text into an XML tree. Using this representation maintains the slicer's language independence (within the limits of the languages supported by srcML), while allowing it to exploit the more natural organization of the source code; for example, deleting the entire body of a function in a single step rather than having to consider each of the function's lines.

This paper compares and contrasts the two observational implementations in the domain of the original algorithm. It compares the actual slices produced by the two algorithms, the time taken for slicing, and the impact of programming language on both. The paper extends our preliminary work on the subject (Binkley et al, 2017) in the following ways.

- We have added six Java systems and replaced the previous production systems with three newer and larger systems, including one with over five million lines of code.
- We pose an additional research question that investigates the influence of the programming language on the slices and the slicing process.
- We present two modifications to the tree-based slicer that enable it to (a) ignore nodes that represent only a small amount of code, and (b) replace a node with one of its subtrees.
- We study the modified slicer to determine its characteristics.

The remainder of the paper is structured as follows. Section 2 provides basic slicing definitions including that of observation-based and observational slicing, while Section 3 describes the two implementations of observational slicing. Then Section 4 states our five research questions, and Section 5 provides demographics for the systems studied. Results of the empirical comparison are presented in Section 6. Finally, related work is discussed in Section 7 and Section 8 summarises the contributions of the paper.

2 Slicing Definitions

Informally, Weiser defined a slice as a subset of a program that preserves the behaviour of the program for a specific slicing criterion (Weiser, 1982). This section briefly describes traditional static and dynamic slicing before considering observation-based slicing and generalised observational slicing.

2.1 Static and Dynamic Slicing

Static (Weiser, 1982) and Dynamic (Korel and Laski, 1988) slicing seek to find an executable subset of a program's statements that exhibits the same behaviour as the original program for a specified variable at a specified location (referred to as a slicing criterion). A static slice does so *for all possible inputs*, while a dynamic slice does so for a selected set of inputs.

It is interesting to note that, while Weiser's original definition of program slicing (Weiser, 1982) is based on statement deletion, static and dynamic slicers tend to use dependency analysis to determine which statements *cannot* be deleted. In contrast, observation-based slicing actually *deletes* statements and then *observes* the behaviour at the slicing criterion.

Definition 1 (Static and Dynamic Slice) A slice S of program P taken with respect to slicing criterion C (composed of variable v and line l) and set of inputs $\mathcal{I}$ is any executable program with the following two properties:

1. S can be obtained from P by deleting zero or more statements from P.
2. Whenever P halts on input I *from* $\mathcal{I}$ with state trajectory T, then S also halts on input I with state trajectory T' and $PROJ_C(T) = PROJ_C(T')$.

The projection function $PROJ_C(T)$ (Weiser, 1982) returns the elements of trajectory T produced at C. For a static slice the set $\mathcal{I}$ is the set of all possible inputs to the program, while for a dynamic slice it is a subset of this set. Usually, the criterion for a dynamic slice explicitly includes $\mathcal{I}$ and is thus given as $(v, l, \mathcal{I})$ denoting variable v at location l for all occurrences in the trajectory, or as $(v_i, l, \mathcal{I})$ where v_i is the ith occurrence of variable v in the trajectory.

2.2 Observation-Based Slicing

Observation-Based Slicing is a recently-introduced alternative to dependence-based slicing: rather than relying on dependency analysis to identify allowed deletions, observation-based slicing uses observation to preserve the relevant part of the state trajectory. Operationally, it does this by tentatively deleting some portion of the program. Only if the result of the deletion compiles and yields the correct output is the deletion made permanent. Because certain lines are only deletable after other lines have been deleted, multiple passes are performed until a pass performs no deletions. One advantage that observation-based slicing brings is the ability to slice any system for which it is possible to delete components and then observe the computation at the criterion.

While similar, the definition of static and dynamic slicing projects elements from the complete state trajectory. In contrast observation-based slicing does not require the complete trajectory. Instead it observes only the relevant values (Binkley et al, 2014):

Definition 2 (Observation-Based Slice) An *observation-based* slice S of a program P taken with respect to slicing criterion $C = (v, l, \mathcal{I})$ composed of variable v, line l, and set of inputs $\mathcal{I}$, is any executable program with the following properties:

1. S can be obtained from P by deleting zero or more components from P.
2. The execution of P for every input I in $\mathcal{I}$ halts and produces a sequence of values $V(P, I, v, l)$ for variable v at line l.
3. The execution of S for every input I in $\mathcal{I}$ halts and produces a sequence of values $V(S, I, v, l)$ for variable v at line l.
4. $\forall_{I \in \mathcal{I}} V(P, I, v, l) = V(S, I, v, l)$.

In practice, the sequence of values produced is observed by injecting a statement that outputs the value of v, just before line l. As this captures the subsequence from the trajectory specific to the criterion, the terms trajectory and trajectory-based are used in the following for the observations made by an observation-based slice. Furthermore, while the definition of the *components* deleted can simply be “statements” to match the definition used with static and dynamic slicing, it can also be entirely language independent. For example, by deleting *lines of text* or *white-space-delimited tokens*, it is possible to effectively slice multi-language systems (Binkley et al, 2014).

2.3 Generalised Observational Slicing

Observation-based slicing was generalised to observational slicing (Gold et al, 2017). The original definition, Definition 2, compares sequences of values observed during execution. *Observational slicing* generalises this comparison by introducing an observer O and a matching relation R as part of the criterion. More generally, an observer $O(P, I)$ extracts from program P some subset of the behaviour for a given input I. Furthermore, the relation between the behaviour of the original program and its slice is related by the matching relation R. *Generalised Observational Slicing* is defined as follows:

Generalised Observational Slice: A *generalised observational* slice S of a program P on a slicing criterion $C = (O, R, \mathcal{I})$ composed of an observer O, a matching relation R, and a set of inputs $\mathcal{I}$, is any executable program with the following properties:

1. S can be obtained from P by deleting zero or more elements from P.
2. The execution of P for every input I in $\mathcal{I}$ halts and produces the observed behaviour $O(P, I)$.
3. The execution of S for every input I in $\mathcal{I}$ halts and produces the observed behaviour $O(S, I)$.
4. $\forall_{I \in \mathcal{I}} O(S, I) \sim_R O(P, I)$.

A simple output-focussed instantiation defines the observer $O(P, I)$ as the output of a program P when P is run on each input $I \in \mathcal{I}$. If the matching relation, R, is *equality*, then the corresponding generalised observational slice S is program P with code that does not influence the output elided.

Observation-based slicing is an instance of generalised observational slicing: given the criterion $C = (v, l, \mathcal{I})$ for an observation-based slice, the observer is $O(P, I) = V(P, I, v, l)$

```
1 if (x < 0) {
2   print x;
3 }
4 y = 42; // Slice taken w.r.t. y
```

Fig. 1 Deletion Window Motivation

(trajectory-based) and the matching relation R is equality. In this paper, trajectory-based observations and equality dominate. However, for some larger systems, the *more general* observer and a more lenient matching relation are used. While the paper uses the two definitions precisely, as a simplification it can be read keeping only the more general notion of observational slicing in mind along with the default sequence observer and equality matching relation.

3 Observational Slicers

This section describes two observational slicing implementations: a line-based slicer, ORBS, and a more recent tree-based slicer, T-ORBS. To begin with the components considered by ORBS are lines of text (Binkley et al, 2014). If source files are formatted with one statement per line, then ORBS can produce 1-minimal statement slices from which it is not possible to delete any single statement, however, it may be possible to delete a combination of multiple statements; consequently the slices are not necessarily *n-minimal*. Unfortunately, finding such slices is computationally intractable.

The core of the observation-based ORBS algorithm, shown as Algorithm 1, loops through each undeleted line in current slice S. For each such line, cl, the algorithm attempts to delete a sequence of lines up to the maximum windows size, *max_ws*. This enables mutually dependent lines (e.g., opening and closing braces on successive lines) to be deleted. The maximum deletion window size places an upper bound on the number of lines that can be deleted together in one deletion. Higher values offer potentially smaller slices at the cost of increased slicing time. As the algorithm is observation-based, the Execute step extracts the sequence of variable values (Line 14), which is compared against the oracle sequence (Line 15). To improve efficiency, ORBS caches results from previous Build and Execute steps. If a subsequent build or execution produces a cache hit then the cached result is used. For simplicity of presentation, the algorithm is stated as working from Lines 1 through length(S). The actual implementation processes the lines of text in the opposite order in the hope, for example, of deleting all uses of a variable before attempting the deletion of its declaration.

As an example, consider the code segment shown in Figure 1. ORBS cannot produce the minimal slice (i.e., just Line 4) by attempting to delete only a single line at a time. While deleting Line 2 alone *is* a legitimate slicing action, Lines 1 and 3 can only be deleted in tandem because deleting only one of them results in a syntax error. ORBS avoids this issue by increasing the deletion window until the result compiles. Using a maximum deletion window size of two or more, ORBS produces the desired slice by first deleting the body of the conditional and then on a subsequent pass the two lines of the conditional itself.

For the generalised observational variant of ORBS shown as Algorithm 2, the Build and Execute phases are replaced by an invocation of the observer O (Line 10), returning a set of observations that are not necessarily sequences of variable values (or trajectories). Moreover, the observations are no longer compared for equality, but rather by matching relation R. Moreover, the observer O executes the candidate for all inputs $I \in \mathcal{I}$, folding properties 2–4 of the Generalised Observational Slice Definition into one observer.

Algorithm 1: Core of the ORBS slicer

```
ORBS_Core($S$, $\mathcal{I}$, $V$, $max\_ws$)
Input: Current slice $S$, input set $\mathcal{I}$, Oracle output $V$, and maximum deletion window size, $max\_ws$
Output: Updated slice, $S$
(1)     $cl \leftarrow 1$ // for each current line
(2)     while $cl \le$ length($S$)
(3)         if $s_{cl} \notin S$   // i.e., if $s_{cl}$ has been deleted
(4)             $cl \leftarrow cl + 1$
(5)             continue
(6)         $builds \leftarrow$ False
(7)         for $ws = 1$ to $max\_ws$
(8)             $S' \leftarrow S - \{s_{cl}, \ldots, s_{\min(\text{length}(S), cl+ws-1)}\}$
(9)             $B' \leftarrow$ Build($S'$)
(10)            if $B'$ built successfully
(11)                $builds \leftarrow$ True
(12)                break
(13)        if $builds$
(14)            $V' \leftarrow$ Execute($B'$, $\mathcal{I}$)
(15)            if $V = V'$
(16)                $S \leftarrow S'$
(17)                $cl \leftarrow cl + ws$
(18)    return $S$
```

Algorithm 2: Core of the *generalised observational* ORBS slicer

```
ORBS_Core($S$, $O$, $R$, $\mathcal{I}$, $V$, $max\_ws$)
Input: Current slice $S$, the criterion consisting of observer $O$, matching relation $R$, and input set $\mathcal{I}$, Oracle output $V$, and maximum deletion window size, $max\_ws$
Output: Updated slice, $S$
(1)     $cl \leftarrow 1$ // for each current line
(2)     while $cl \le$ length($S$)
(3)         if $s_{cl} \notin S$   // i.e., if $s_{cl}$ has been deleted
(4)             $cl \leftarrow cl + 1$
(5)             continue
(6)         for $ws = max\_ws$ to 1
(7)             $S' \leftarrow S - \{s_{cl}, \ldots, s_{\min(\text{length}(S), cl+ws-1)}\}$
(8)             $V' \leftarrow O(S', \mathcal{I})$
(9)             if $V \sim_R V'$
(10)                $S \leftarrow S'$
(11)                $cl \leftarrow cl + ws$
(12)                break
(13)    return $S$
```

The experiments use a parallelised version of observational ORBS (Islam and Binkley, 2016; Binkley et al, 2014) that considers several different deletion window sizes in parallel. The largest deletion window that succeeds (i.e., compiles and produces the same trajectory) is accepted, while the other attempts are discarded. The algorithm then proceeds to the next line where again a number of deletion windows are tried in parallel. All experiments in this paper involving the line-based slicer make use of the parallelised version of ORBS.

Turning to the the second observational slicing implementation, T-ORBS was built to slice Simulink models and their embedded Stateflow, both of which are stored using XML (Gold et al, 2017). The core of the T-ORBS algorithm is shown as Algorithm 3. It computes slices of trees. Thus, rather than line-by-line, the loop on Line (2) performs a breadth-first tree traversal. During each iteration, T-ORBS attempts to delete the subtree rooted at current node, c. If the resulting system produces the correct sequence of values then c is permanently deleted. Otherwise c's children are placed on the worklist. The breadth first ordering aims to delete (large) top-level structures (e.g., classes or functions) before considering their constituent parts. Other orders, such as a depth-first traversal, are possible, but their study is left to future work. Outside of efficiency, in principle the order makes no difference as a component is either deletable or not. In practice, statement capture, as discussed in Section 6.2, could lead to minor differences.

Algorithm 3: Core of the observational Tree-ORBS Slicer

```
T-ORBS_CORE($T$, $O$, $R$, $\mathcal{I}$, $V$)
Input: Current Tree $T$, the criterion consisting of observer $O$, matching relation $R$, and input set $\mathcal{I}$, and Oracle output $V$
Output: Updated Tree, $T$
(1)   $q \leftarrow$ APPEND($empty\_queue$, $start\_node(T)$)
(2)   while $\neg$ EMPTY($q$)
(3)       $c \leftarrow$ DEQUEUE($q$)
(4)       $T' \leftarrow$ DELETE($T, c$)
(5)       $V' \leftarrow O(T', \mathcal{I})$
(6)       if $V \sim_R V'$
(7)           $T \leftarrow T'$
(8)       else
(9)           $q \leftarrow$ APPEND($q$, CHILDREN($c$))
(10)  return $T$
```

Since the T-ORBS implementation was constructed to operate on XML representation of Simulink models, traditional source code such as C or Java code has to be first transformed into XML to be sliced. This transformation is done using srcML (Collard, 2005). In theory, T-ORBS should be able to slice the resulting XML tree-based source code representation without modification. In practice, this came close to being true. Unlike Simulink's XML representation, srcML includes XML namespaces. Thus it was necessary to extend T-ORBS' command-line arguments to include a namespace specification. The only other change necessary was to transform srcML's output from mixed content, where the source text is free with the tags, to element content. In greater detail, the output from srcML uses mixed content (much like HTML) where an element may contain text and other elements. For example, the <if> tag includes the *text* "if" and several elements including the *element* for the (boolean) condition: <if>if <condition> ... </condition> ... </if>. The transformation to element content moves the "free" text "if" to be an attribute of an element, resulting in the XML <if text="if"> <condition> ... </condition> ... </if>. This transformation avoids ambiguities concerning to which element the intermixed text belongs. The resulting T-ORBS slicer is capable of slicing any language supported by srcML or any other XML creation tool. For example, it was initially developed for C code, but was able to slice C++ and Java code without modification.

Finally, to relate the expected effort expended by the two algorithms, we compare their complexity in terms of the size of representation, the way it is examined, and the cost of executing the observer. The cost, C, of evaluating a candidate slice by the observer is typically the time needed to build the program and run all the tests. It is therefore a combination of the build time and the execution times over the inputs (we assume that the cost of observing the output and comparing it to the oracle is included within the execution time). More formally, each observation requires one build, B, and as many executions, E, as there are inputs $I \in \mathcal{I}$. In other words, in the worst case $C = B + E \times |\mathcal{I}|$.

ORBS' complexity is a function of the number of window sizes, denoted WS, and the number of lines of code, L, while T-ORBS' complexity is a function of the number of XML nodes, N. A single pass of the ORBS algorithm attempts WS deletions starting at each of the L lines yielding a complexity of $O(L \times WS)$ observations. In the worst case each pass deletes only a single line, yielding an overall complexity of $O(L^2 \times WS)$. T-ORBS complexity is similar, but replaces L with the number of nodes, N. A single pass of Algorithm 3 makes $O(N)$ observations. In the worst case this pass deletes a single leaf node, yielding a complexity of $O(N^2)$. Finally, we note that empirically there is a strong linear correlation between L and N where $N = 6.5 \times L$, $R^2 = 0.98$, so for comparison we might write T-ORBS complexity as $O(L^2)$.

3.1 Modifications to T-ORBS

The initial experiments presented in Section 6 led to two interesting insights: first, T-ORBS can get bogged down attempting to delete small trees, and second, by its very nature, it has to preserve the block nesting structure of the code and thus if a statement is in the slice then some form of its enclosing control structure must also be included. These insights led to two modifications to T-ORBS: the ability to specify a *size threshold* and the ability to perform *subtree replacement*. These extensions are presented here for easier comparison to the original algorithm above. The effects of these modifications are explored in Section 6.5.

3.1.1 Size Threshold

To address the first issue, the core of T-ORBS algorithm, Algorithm 3, was updated to accept a minimum size threshold h. Only subtrees that represent at least h lines of code are considered for deletion. The result is Algorithm 4. The implementation includes a *size threshold*, given by the command-line option -st. This option is followed by a list of size thresholds. Thus, -st 0 is equivalent to the original T-ORBS slicer, while using -st 1 will not attempt to delete subtrees that represent less than one complete line of code. Multiple thresholds can be given, where -st 4,2,1 first performs a single pass ignoring subtrees that represent less than four lines of code, then a single pass ignoring subtrees that represent less than two lines, and finally one or more passes ignoring subtrees that represent less than one line of code. Akin to the original T-ORBS algorithm, in all cases, the last value is repeated until a pass is unable to delete any further code.

Algorithm 4: Core of the Tree-ORBS Slicer including *threshold*

```
T-ORBS_CORE($T$, $O$, $R$, $\mathcal{I}$, $V$, $h$)
Input: Current Tree $T$, the criterion consisting of observer $O$, matching relation $R$, and input set $\mathcal{I}$, Oracle output $V$, and threshold $h$
Output: Updated Tree, $T$
(1)   $q \leftarrow$ APPEND($empty\_queue$, $start\_node(T)$)
(2)   while $\neg$ EMPTY($q$)
(3)       $c \leftarrow$ DEQUEUE($q$)
(4)       if $LoC(c) \geq h$
(5)           $T' \leftarrow$ DELETE($T$, $c$)
(6)           $V' \leftarrow O(T', \mathcal{I})$
(7)           if $V \sim_R V'$
(8)               $T \leftarrow T'$
(9)           else
(10)              $q \leftarrow$ APPEND($q$, CHILDREN($c$))
(11)  return $T$
```

3.1.2 Subtree Replacement

The second modification to the core algorithm (for subtree replacement) is shown in Algorithm 5. Given a node N, the algorithm tests if one of N's children represents at least 60% of the source lines represented by N. If it does, then N is replaced with this child and the resulting program is compiled and executed. If the output is unchanged, then the replacement is made permanent. While informal, the 60% cutoff was arrived at empirically where smaller values tend to result in too many failed attempts while larger values preclude viable replacement opportunities.

Algorithm 5: Subtree Replacement

```
SUBTREE_REPLACEMENT($T$, $O$, $R$, $\mathcal{I}$, $V$, $N$)
Input: Current Tree $T$, the criterion consisting of observer $O$, matching relation $R$, and input set $\mathcal{I}$, Oracle
output $V$, and current node $N$
Output: Updated Tree, $T$
(1)     $subtree\_sizes \leftarrow$ MAP($line\_count$, children($N$))
(2)     $big\_kid \leftarrow$ MAX(subtree_sizes)
(3)     if $big\_kid > 0.60 \times line\_count(N)$
(4)         $T' \leftarrow$ REPLACE($T$, $N$, $big\_kid$)
(5)         $V' \leftarrow O(T', \mathcal{I})$
(6)         if $V \sim_R V'$
(7)             $T \leftarrow T'$
(8)     return $T$
```

The initial implementation applied the subtree replacement whenever a node could not be deleted slowing the slicer down on average by a factor of 30% to 50%. Unfortunately, this means that subtree replacement is too expensive for continuous application. Subsequently, T-ORBS was modified to make a subtree replacement pass when it received the special size threshold of `-1`.

4 Research Questions

Prior work (Binkley et al, 2014) compared ORBS with various forms of dynamic slicing, all of which are its 'algorithmic cousins' because they all have common roots in dynamic analysis. Subsequently, ORBS slices were compared to static slices in order to explore the subtleties and limits of static analysis (Binkley et al, 2015). This paper directly compares the two implementations of observational slicing, ORBS and T-ORBS, "head-to-head". The comparison is framed by the following five research questions.

RQ1: How do ORBS and T-ORBS slices compare quantitatively? This quantitative question considers the sizes of the slices produced by the two implementations.

RQ2: How do the slices produced by ORBS and T-ORBS compare qualitatively? This qualitative question considers differences in the slices produced by the two implementations.

RQ3: What impact does implementation have on the time taken to compute a slice? This quantitative question asks if T-ORBS' ability to delete large sub-trees pays for its having to consider a multitude of small subtrees (e.g., each token of an expression such as `a * b + c`).

RQ4: What impact does source language have on slicer behaviour? This question investigates ORBS and T-ORBS behaviour when slicing four systems for which we have implementations in two languages.

Table 1 Subjects Considered in the Empirical Investigation

Program	Language	LoC	SLoC	Slices
Known Semantics				
sumprod	C	20	16	8
wc	C	128	70	17
mug	C	73	62	16
mbe	C	82	62	12
jMBE	Java	62	53	10
Exhaustively Sliced (Sorted by SLoC)				
jPermutation	Java	142	129	43
tcas	C	185	141	43
jHanoi	Java	171	158	62
jTCAS	Java	198	165	43
hanoi	C	206	177	21
schedule2	C	302	256	74
totinfo	C	415	274	54
schedule	C	465	313	58
printtokens2	C	579	361	74
printtokens	C	733	436	81
replace	C	658	541	309
jDaisy	Java	1411	787	101
Production Systems				
DAIDALUS	C++	44 897	22 504	140
jDAIDALUS	Java	38 750	20 361	140
GMAT inc libs	C/C++ etc.	5 219 731	2 912 526	15

RQ5: How does the modified T-ORBS algorithm perform? This question considers the two modifications described in Sections 3.1.1 and 3.1.2: not getting "bogged down" attempting the deletion of small subtrees and enabling *subtree replacement*.

5 Subject Demographics

Our experiments concern the 20 programs shown in Table 1. These are split into three sets, each of which is specifically chosen to help address various aspects of the comparison. The first set includes five widely-studied (tiny) benchmark programs taken from the literature because they have been used to exemplify slicing challenges and techniques. While not large, the programs of the second set are small enough that it is feasible to compute all slices for all computations of scalar values (e.g., values of types int, char, double, etc.). The third set includes three production systems (GMAT, DAIDALUS, jDAIDALUS) and is used to study the scalability of observational slicing. All three sets contain programs that come in two variants for two different programming languages. For example, jHanoi (Java) and hanoi (C) both solve the Tower-of-Hanoi puzzle.

For each program the table includes the languages used in the code, the number of lines of code (LoC) and source (non-comment, non-blank) lines of code (SLoC), and the number of slices of the program computed. For the larger systems, computing all such slices is infeasible and thus a reduced set is considered. The source code of all programs (except for the production systems) has been automatically formatted in the same way so that differences in layout do not impact the slicing results. In particular the Java programs have been formatted in the same way as the C programs (e.g., so that braces are each on their own line).

```
word_count()
{
  while (scanf("%c", &c) == 1)
  {
    characters = characters + 1;

    if (c == ’\n’)
    {
      lines = lines + 1;
    }

    if (isletter(c))
    {
      if (inword == 0)
      {
        words = words + 1;
        inword = 1;
      }
    }
    else
    {
      inword = 0;
    }
  }
}

int isletter(char c)
{
  if (((c >= ’A’ ) && (c <= ’Z’))
      || ((c >= ’a’ ) && (c <= ’z’)))
  {
    return 1;
  }
  else
  {
    return 0;
  }
}
```

Fig. 2 The word count program

5.1 Known Semantics

The first of the tiny programs, sumprod computes the sum and product of the first ten integers. It is commonly used to illustrate that slicing can separate the computation of the sum from that of the product. The second tiny program, word count, is shown in Figure 2. It computes the number of lines, words, and characters in an input text. Its slices are used in many papers on slicing (Gallagher and Lyle, 1991; Reps and Turnidge, 1996) as trivial examples of static slices. It is implicit in all treatments of this example that the slices are trivial and present few interesting issues, hence its widespread use as an illustrative example. As we shall see, observational slicing reveals that there *are*, in fact, subtleties, even in this simplest of examples.

Third, *the SCAM mug* example, shown in Figure 3, appeared on the souvenir mug given to delegates of the first incarnation of the SCAM conference in Florence, 2001. It has subsequently been used as a ‘challenge’ example for slicing algorithms (Ward, 2003), due to its subtle semantics and the difficulty in obtaining a minimal slice, even using very sophisticated algorithmic techniques.

Finally, the *Montréal Boat Example*, shown in Figure 4, was formulated by Sebastian Danicic and John Howroyd during a boat excursion at the 2^{nd} incarnation of the SCAM

```
1  int mug(int i, int c, int x)
2  {
3      while (p(i))
4      {
5          if (q(c))
6          {
7              x = f();
8              c = g();
9          }
10         i = h(i);
11     }
12     return x;
13 }
```

Fig. 3 The SCAM'01 Mug Example. Predicates p and q, and function h depend only on their single formal parameter while functions f and g return (unknown) constant values. The key point in this code is that in any terminating execution the final value of x is independent of Line 8: if q(c) is initially false, it remains false and thus x retains its initial value. On the other hand, if q(c) is true one or more times then x will have the value assigned at Line 7. In the latter case, it does not matter how often q(c) is true and thus the assignment at Line 8 does not impact the value of x at Line 12.

```
int mbe(int j, int k)
{
    while (p(j))
    {
        if (q(k))
        {
            k = f1(k);
        }
        else
        {
            k = f2(k);
            j = f3(j);
        }
    }
    return j;
}
```

Fig. 4 The Montréal Boat Example. Predicates p and q, and functions f1, f2, and f3 are not shown. They depend only on their single formal parameter. The relevant observation is that in any terminating execution, the computation of k is irrelevant to the computation of j.

conference in Montréal, 2002. It was discussed at length at the conference as an example of the subtleties of producing minimal slices (Danicic and Howroyd, 2002). This example is considered in a C variant (mbe) and a Java variant (jMBE).

5.2 Exhaustively Sliced

In addition to having been used in prior slicing research (Binkley et al, 2014, 2009; Harman et al, 2009; Binkley et al, 2001), the next set of programs was chosen because it is possible to compute all slices for assignments involving basic scalar types (e.g., ints). Doing so supports the comparison over a large number of slices that have a wide range of complexity (from slices taken with respect to variable initializations all the way through to slices taken with respect to final outputs). The programs are either written in C or Java and most of the programs are compiled from a single file (plus necessary header files for the C programs). The only exception is jDaisy, which contains multiple Java files. Only the main Java file Daisy.java is sliced. It includes 631 lines of which 307 are non-blank, non-comment lines. Table 1 includes the size of each program sliced in this category, not including header files.

5.3 Production Systems

The three remaining systems are in production use and comprise the NASA-led General Mission Analysis Tool (GMAT) (NASA, 2017b) (an open source system for space mission analysis that is in its tenth release year), and the two parts of the NASA Detect and AvoID Alerting Logic for Unmanned Systems (DAIDALUS) system (NASA, 2017a) (implementations of detection and alerting logic, and manoeuvre guidance for unmanned aircraft systems). GMAT includes (or downloads) a number of libraries required to build it. From a slicing perspective it is interesting for its large overall size, the size of some of the individual source files, and the fact that, when its required supporting libraries are included for building, it is a system comprising some 13 programming languages along with various other encoding languages like XML and YAML. The C/C++ core of the system without the libraries includes around 360k SLoC; the figures shown in Table 1 are the totals produced by the utility cloc for the complete source package including libraries. Everything cloc describes as code is included. DAIDALUS is interesting because it has two implementations of the same functionality, one in C++ and the other Java (treated as two separate systems in Table 1, DAIDALUS and jDAIDALUS, to separate the C++ and Java implementations respectively). The totals shown for these systems are for the C++ and Java files only.

For the larger systems, computing all possible slices is infeasible. Instead, the simple output-focused version of *generalised observational slicing* (Gold et al, 2017) was used where the sliced program must generate the same output as the original program for the same set of test inputs. For both GMAT and DAIDALUS, the test cases chosen were drawn from example and tutorial code supplied with the systems. The oracle observations considered were the full output of the example code (DAIDALUS) or an excerpt thereof (GMAT: an excerpt was used because the full output includes a timestamp and is thus different with every execution). Thus the slice produced in each case constitutes a one-minimal slice for the target file for the example code used.

While only a single file was sliced for the other two sets (the known semantics set and the exhaustively sliced set), a set of files was sliced for the production systems. For DAIDALUS, all the source files were selected except one whose build included a time/date stamp and thus produced a different binary for every compilation of the same source). For jDAIDALUS, the set of files comprising the largest Java package was selected (except one where the slicer run failed and which is thus excluded from the analysis here). For GMAT, a small selection of C/C++ files (the languages GMAT's core is written in) was selected from the large number of source files.

6 Results

6.1 RQ1: How do ORBS and T-ORBS slices compare quantitatively?

To answer RQ1, the quantitative slice-size comparison looks at two sets of slices. The first set aims to determine if, like ORBS, T-ORBS can produce minimal static slices of the tiny, well-studied benchmark programs. The second set includes the exhaustively sliced programs. Exhaustive slicing avoids potential experimenter bias when selecting which slices to consider. We constructed 1026 slices in total including, for completeness, 63 slices of the known-semantics benchmark programs. Table 2 shows the average slice sizes produced by the two slicers for the 17 exhaustively sliced programs. The average percent reduction ranges from 21% to 84%.

Table 2 Average Slice Sizes (values in bold are better)

Program	Original (SLoC)	Average Slice ORBS (SLoC)	Average Slice T-ORBS (SLoC)	Percent Reduction ORBS	Percent Reduction T-ORBS
Known Semantics					
sumprod	16	9.0	9.0	44%	44%
wc	70	**15.8**	19.8	**77%**	72%
mbe	62	**29.7**	30.5	**52%**	51%
mug	62	19.4	**19.2**	69%	**69%**
jMBE	53	**40.2**	41.8	**24%**	21%
Exhaustively Sliced (Sorted by Original SLoC)					
jPermutation	129	**92.9**	97.5	**28%**	24%
tcas	141	**22.9**	22.9	**84%**	84%
jHanoi	158	**39.1**	44.1	**75%**	72%
jTCAS	165	64.8	**48.1**	61%	**71%**
hanoi	177	**36.1**	40.8	**80%**	77%
schedule2	256	79.8	**73.1**	69%	**71%**
totinfo	274	83.4	**72.4**	70%	**74%**
jDaisy	307	128.1	**122.9**	58%	**60%**
schedule	313	**115.2**	124.8	**63%**	60%
printtokens2	361	122.7	**122.3**	66%	**66%**
printtokens	436	192.7	**191.0**	56%	**56%**
replace	541	**186.5**	202.2	**66%**	63%

Comparing the two slicers, for most systems their performance is similar. Only two programs, wc and jTCAS, show more than a five percentage point difference in terms of the percent reduction. For wc ORBS significantly outperforms T-ORBS, which is caused by T-ORBS maintaining structure when ORBS does not and for jTCAS T-ORBS significantly outperforms ORBS, which is caused by T-ORBS being able to delete large blocks of code that can only be deleted together – something that ORBS is unable to. Both scenarios will be illustrated later in this section.

For two programs with known semantics, sumprod and mug, ORBS and T-ORBS produce almost the same slices with two exceptions. First, there are layout differences (these are removed, for example, by pretty printing). Second, there are some semantic differences because T-ORBS can remove elements within a line, e.g. it removes the int type from C function declarations (functions without a declared type as given the implicit default type int). Moreover, T-ORBS removes parameters from function declarations when the parameter has been removed from the function body, and deletes the corresponding arguments at the function call site. However, none of the above cause the slices to have more than minor differences and thus the slice sizes as measured in SLoC are the same.

A larger difference occurs with two mbe slices where ORBS removes lines of text that are part of an if statement, while T-ORBS retains the predicate and empty true branch. This statement is found on Lines 5-13 in Figure 4. In the slice, k's value does not affect the value of j and thus only the assignment to j in the false branch is semantically necessary. This enables ORBS to delete Lines 5-9. The following is the T-ORBS slice:

```
5        if ((k))
6        {
7
8        }
9        else
10       {

        j = f3();
    }
```

The T-ORBS slice correctly untangles the computations of k and j. However, it retains Lines 5–9 because in the tree of the if statement, the keyword if is part of a parent node that has three subtrees representing the condition, then-part, and else-part of the if statement. Thus its removal is only possible if the entire if statement can be deleted. Research Question 5 considers the possibility of replacing a parent (e.g., the if node) with one of its children (e.g., the else branch in the example). These two slices account for the minor average-slice-size differences seen for mbe and jMBE in Table 2 and for the more significant differences seen for wc.

The significant difference between T-ORBS and ORBS for jTCAS is due to T-ORBS' ability to remove large blocks of code. jTCAS has a few methods which ORBS is not able to delete. These methods contain a return statement that must be retained (otherwise a compile error occurs) and consequently the computation of the returned value must also be retained. In contrast, T-ORBS can delete the methods entirely as soon as they are no longer called.

The same happens in another Java program with significant differences in average slice sizes, jTCAS. Here, T-ORBS is able to remove complete if statements or while loops, while ORBS retains them. This happens for three of the jPermutation slices and for 34 of the jTCAS slices. Interestingly, the same does not occur for jDaisy or jHanoi, where most of the if and while statements are small and can be completely removed by ORBS.

Returning to the complete set of 1026 slices, Figure 5 graphs slice-size differences. Each bar is the size of the ORBS slice minus the size of the T-ORBS slice. Most of the differences hover around zero. For example, 65% (661 of 1026) differ by less than 10 lines, although only 9.6% (98) have the exact same size. T-ORBS produces smaller slices than ORBS 40.0% (410) of the time while ORBS produces smaller slices than T-ORBS 50.0% (518) of the time.

Inspecting a random sample of the slices in the second set, ORBS tends to produce smaller slices when T-ORBS is forced to preserve code structure, as illustrated in the mbe slice above. Other reasons why ORBS produces smaller slices are discussed in Section 6.2. On the other hand, T-ORBS produces smaller slices when ORBS deletes an initialisation because of *fortuitous placement*, which later inhibits the deletion of lines elsewhere in the code. For example, consider two subsequent function calls to functions f and then g each having a single local variable, l_f and l_g, respectively. In C, local variables are not automatically initialized and thus end up with the value found in the memory they are assigned. Unless overwritten, the value of l_g during the call to g will be the final value of l_f from the call to f (assuming that the activation records for the two have a similar layouts). If ORBS deletes the initialization of l_g because it fortuitously has the correct value, it will then be unable to later delete f and the code that gives l_f its final value because it is needed to maintain the initial value of l_g. While T-ORBS is susceptible to the same issue, it deletes components in a different order and thus commonly deletes f before attempting to delete l_g.

In summary, for RQ1 the slices produced by the two algorithms are similar in size for the majority of programs. ORBS was seen to have one representational advantage in that T-ORBS is forced to retain elements to maintain the tree structure with which it represents code. On the other hand, T-ORBS is able to remove large blocks early which sometimes need to be (partially) retained by ORBS.

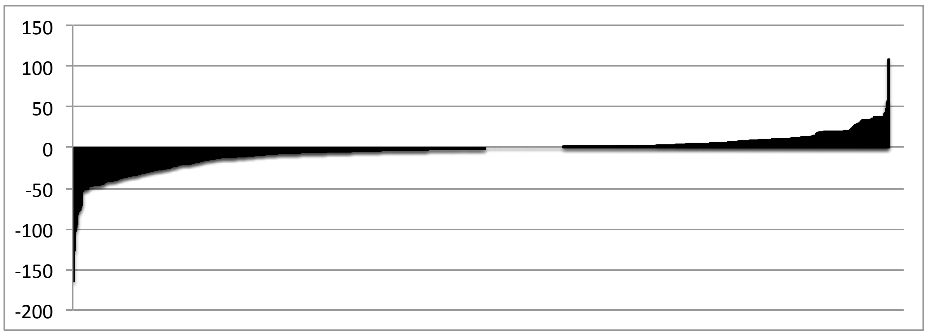

Fig. 5 Slice Size Comparison (positive values indicate that the T-ORBS slice is smaller)

6.2 RQ2: Qualitatively how do the slices produced by ORBS and T-ORBS compare?

RQ2 provides a qualitative look at the slices. The analysis focuses on the tiny programs where knowing the ground truth facilitates comparison. We find that there are four categories of difference between the implementations: *preservation, dissection, capture, and order*. First and foremost, except for the mbe slice described in Section 6.1, like ORBS, T-ORBS computes minimal slices for the challenge problems mug and mbe. And even when not minimal, T-ORBS untangles the complex control and data dependence interactions found in the code. This observation and the representative examples considered in this section point to T-ORBS structure preservation as being the one substantial difference between the two implementations. In the remainder of this section we consider two additional *preservation examples*, four *dissection examples*, two *capture examples*, and two *order examples*.

6.2.1 Preservation

The second preservation example finds T-ORBS structure preservation a detriment in the tcas slice taken with respect to need_downward_RA. It turns out that the test suite includes only tests that make the predicate of the if statement on Line 4 true. ORBS "discovers" this and thus its slice omits the predicate. More importantly, it also omits those definitions upon which the predicate depends. Thus the ORBS slice retains only Line 6 of the following code. In contrast, T-ORBS retains all of the code because it cannot remove the predicate of an if statement without removing both its then and else subtrees. This T-ORBS shortcoming is revisited in Section 6.5.

```
1 #define OLEV 600 /* in feets/minute */
2 ...
3   enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (
         Cur_Vertical_Sep > MAXALTDIFF);
4   if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
5   {
6     need_downward_RA = Non_Crossing_Biased_Descend && Own_Above_Threat();
```

A final preservation example shows T-ORBS' inability to delete the lines #ifdef DEBUG and #endif. In the srcML representation, each of these is a separate (sub)tree and thus T-ORBS cannot remove them, because it attempts to do so one at a time.

6.2.2 Dissection

In the preservation examples, the use of the tree-based structure by T-ORBS causes it to include parent structures (e.g., if statements) when only a child structure (e.g., the else block) is required, as well as preprocessor directives such as #ifdef. However, the use of a tree-base structure also enables T-ORBS to "dissect" individual lines of text. Four dissection examples are considered. The first dissection example involves the replacement of the lines

```
typedef enum Boolean
{ FALSE = 0, TRUE = 1, FAIL = 0, SUCCEED = 1, OK = 1, NO = 0, YES = 1, NOMSG =
      0,
    MSG = 1, OFF = 0, ON = 1 } BOOLEAN;
```

with

```
typedef enum { OK = 1, NO = 0, YES } BOOLEAN;
```

The second dissection example involves the function header int h(int i) where the type int is C's implicit default type. ORBS is unable to delete the text line containing the function header because the function h is part of the slice. However, T-ORBS reduces this line to h(i), because the srcML for a function includes four subtrees: (return) type, (function) name, parameter_list, and block (body):

```
<function>
  <type><name>int</name></type>
  <name>h</name>
  <parameter_list> (<parameter><decl><type><name>int</name></type> <name>i</
       name> </decl></parameter>)</parameter_list>
  <block>{}</block>
</function>
```

A related dissection example occurs when T-ORBS removes parameters because a preceding call has placed the same value at the correct stack location. For example, in the following code (a fragment of the slice for the Montréal Boat Example in Fig. 4), the call q(k) places k on the stack in the first parameter position, thus the call f1() effectively also passes k to f1() (because k is still on the stack). This T-ORBS behaviour can be suppressed using an enhancement discussed in Section 6.5.

```
 if (q(k))
 {
   k = f1();
   printf("\norbs:%d\n", k); //slice here w.r.t. k
```

The final dissection example comes from the SCAM mug example. This program is really a schema (Danicic et al, 2004; Laurence, 2004) as it involves several unspecified constants. In the concrete implementation, these constants are assigned values using command-line arguments that are extracted using code such as x = (int) strtol(argv[3], NULL, 10). Because the actual value chosen is uninteresting (apart from degenerate values such as zero), various constants were chosen. The ORBS test suite, which includes no degenerate values and is thus sufficient to ensure that ORBS produces the expected static slice, was initially used with T-ORBS. The particular value chosen happened to be 10, the base of the conversion used in the call to strtol. T-ORBS replaced the initialization with x = (int) (10), which preserved the behaviour. Updating its test suite to include a value other than 10 caused T-ORBS to generate the expected slice.

6.2.3 Capture

The next two examples illustrate a form of "capture" in which ORBS is able to combine parts of different syntactic units. The first capture example is from sumprod. The first seven lines of which are as follows:

```
1  for(i=1; i<=10; i++)
2  {
3    sum = sum + i;
4    prod = prod * i;
5  }
6  printf("at end i = %d\n", i);
7  printf("\norbs:%d\n", i); //slice here w.r.t. i
```

For the slice taken with respect to i, T-ORBS produces the expected slice by removing the body of the loop and the first call to printf (Lines 3, 4, and 6). In contrast, with a window-size of four, ORBS deletes Lines 2-5. In the resulting code, the first of the two printf calls gets "captured" by the loop header leading to the following code.

```
for(i=1; i<=10; i++)
  printf("at end i = %d\n", i); // indentation added for clarity
printf("\norbs:%d\n", i); //slice here w.r.t. i
```

Because this syntactically correct program computes the correct values for i, it is a slice of the original. Here ORBS produces the smaller slice (of only three lines), while T-ORBS produces the more natural slice (the one that preserves more of the original structure).

The second capture example is one of the more interesting ORBS slices where the slice combines statements from two (adjacent) functions. The following code is from the word count program. The slice was taken with respect to the value of c at the top of the function isletter. It just so happens that the same variable name is used by the caller.

```
1   while (scanf("%c", &c) == 1)
2   {
3     if (isletter(c))
4     {
5   ...
6
7   int isletter(char c)
8   {
9     printf("\norbs:%c\n",c); //slice here w.r.t. c
10  ...
```

In this example, ORBS discovers that it is possible to merge code from these two functions. The resulting slice includes Line 1, the while loop, and Line 9, the call to printf.

```
while (scanf("%c", &c) == 1)
  printf("\norbs:%c\n",c); //slice here w.r.t. c
```

T-ORBS is unable to produce such a slice because it cannot merge subtrees.

6.2.4 Order

The final set of examples are order examples, which occur because of a difference in the order in which deletions are attempted. In the first example, the program tcas initializes the variable alt_sep to zero at the top of the function alt_sep_test. The slices taken with respect to this initialization must preserve the call which ORBS does by retaining the entire line fprintf(stdout, "%d\n", alt_sep_test()). In contrast, T-ORBS reduces this line to (alt_sep_test()), dropping the name fprintf associated with the call and two of the three arguments leaving an expression list with a single entry, the call to alt_sep_test().

The second order example highlights an interesting results of the test suite including a test with insufficient command-line arguments. For this test case no output should be generated. The following is the relevant part of the code.

```
1  if(argc < 13)
2  {
3      fprintf(stdout, "Error: Command line arguments are ...\n");
4      exit(1);
5  }
6  ...
7  Climb_Inhibit = atoi(argv[12]);
8  ...
9  fprintf(stdout, "%d\n", alt_sep_test());
```

Because ORBS and T-ORBS involve different deletion orders, T-ORBS retains the if statement and the call exit(1) (but not the call to fprintf on Line 3). ORBS on the other hand deletes Lines 1-5 including the call exit(1). It thus is forced to retain the call atoi(argv[12]), which causes the program to abort when there are insufficient arguments – effectively preventing the program from calling alt_sep_test(). In this case, again, T-ORBS produces the more natural slice.

In summary for RQ2, the differences in the slices produced by the two slicers fall into four categories. ORBS produces smaller slices when T-ORBS, by its very nature, is forced to retain more of the structure of the underlying code. In contrast, T-ORBS naturally performs "sub-line" deletions, which in one case helped to focus an enum on only those entries relevant to the slice. Third, ORBS is more prone to capture lines. While this can produce smaller slices, they are often harder to comprehend. On the other hand, in the final group T-ORBS produces several *more intuitive* slices. It is clear from these examples that each slicer brings pros and cons to the qualitative comparison.

6.3 RQ3: What impact does implementation have on the time taken to compute a slice?

RQ3 takes a quantitative look at slicing time. In the broad context, the expectation is that T-ORBS will be faster when large chunks of code can be deleted in a single deletion (e.g., an entire function body), but must pay for this as it considers all subtrees. This is particularly costly when a line is required by the slice and has lots of subtrees. For example, T-ORBS attempts the independent deletion of a, =, b, +, and c from the statement a = b + c. T-ORBS also incurs the cost of running srcML, which includes the single execution to create the initial srmML version and then an execution ahead of each compile to convert the tree back into source code. Empirically srcML makes the text of the source about 4.6 times larger ($R^2 = 0.99$). Based on a random sample of the process statistics for T-ORBS, tree manipulations consume from 2% to 50% of the total execution time. It is higher in the last iteration where the cache hit rate is quite high. Finally another expected advantage of ORBS is the speed-up achieved through parallelisation by attempting multiple deletion windows in parallel making the most of multi-core CPUs.

Table 3 shows the CPU (user) and wall-clock times for both ORBS and T-ORBS for the twenty programs of Table 1. The times displayed are average times over all the slice computations for each program. The smaller time (both CPU and wall-clock) for each program is highlighted in bold. With one exception, ORBS is universally faster. This exception is the CPU time for the largest program, which provides greater opportunities to delete large subtrees (some larger than many of the smaller programs in their entirety) in a single deletion. Furthermore, the impact of ORBS parallelism can be seen by separately comparing the user times and the wall-clock times.

It is slightly surprising that ORBS achieves lower CPU times for all but one program. Given that ORBS is attempting multiple deletion windows in parallel but only uses the result of the winning one, as opposed to T-ORBS which is single threaded, it was expected that, barring edge cases, T-ORBS would exhibit lower CPU times than ORBS. Looking ahead to Section 6.5 it turns out that this is largely caused by T-ORBS attempting sub-line deletions. When limited to deleting nodes that represent at least one line of code, T-ORBS runs about three times faster.

ORBS parallelisation can be seen at work in particular for the Java programs, where the CPU utilization is well above 200%. Here, the inherent parallelism coming from the JVM is also evident. This JVM parallelism is even evident in the T-ORBS Java slices when the wall-clock time is less than the CPU time. Numerically, ORBS is between 1.3% and 1116% faster than T-ORBS in terms of CPU time and between 77% and 1089% faster in terms of wall-clock time. On average it is 71% faster in terms of CPU time and 319% faster in terms of wall-clock time, again showing the impact of its parallelisation.

In summary, the investigation into RQ3 involving the impact of slicer implementation strategy on slice time suggests trends related to scalability. Outside the largest system, it seems that the ability of T-ORBS to delete large subtrees does not counter-balance the cost of traversing the trees down to leaves. Overall, ORBS makes much better utilization of the CPU and performs better in terms of wall-clock time for all of the programs. Finally, the slice times hint that T-ORBS would benefit from greater use of parallelism, because the main reason for ORBS' performance is leveraging multi-core CPUs.

6.4 RQ4: What impact does source language have on slicer behaviour?

Four of the systems studied have both C and Java implementations: mbe, hanoi, tcas, and DAIDALUS. The first two, mbe and hanoi, share a common implementation style because both versions of mbe and the C version of hanoi were written by the authors. The two versions of the other two programs were written independently.

The investigation considers first a quantitative look at the data and then a qualitative one. The relevant quantitative data includes both slicing time and the reduction attained by each slicer. For each of the eight programs, Table 4 shows the T-ORBS data, which includes the mean slicing time (measured in seconds) and the mean percent reduction (measured as percentage of SLoC). Each comparison includes the results of Tukey's HSD (Honestly Significant Difference) Tukey (1949), which provides some statistical backing to the numeric trends across all eight programs. This statistical test performs pairwise comparison of a set of treatments while correcting for multiple comparisons and then summarizes the results by labelling each treatment with a letter. Treatments that do not share a letter are statistically different from each other. In addition, we consider head-to-head t-tests comparing the time and reduction percentage for each pair independently.

Table 3 Slice Times (smaller times shown in **bold**)

	ORBS		T-ORBS	
Program	User Time (h:m:s)	Wall Clock (h:m:s)	User Time (h:m:s)	Wall Clock (h:m:s)
sumprod	**0:03**	**0:02**	0:04	0:10
wc	**0:10**	**0:06**	0:10	0:13
mug	**0:10**	**0:14**	0:15	0:30
mbe	**0:16**	**0:22**	0:18	0:40
jMBE	**4:30**	**1:13**	4:43	4:03
jHanoi	**11:38**	**3:26**	13:02	11:46
jTCAS	**15:45**	**5:42**	20:55	20:37
jPermutation	**14:34**	**4:29**	28:55	28:09
tcas	**0:18**	**0:12**	0:38	0:58
hanoi	**0:21**	**0:14**	0:55	2:30
schedule2	**0:49**	**0:47**	3:07	5:30
jDaisy	**59:54**	**17:23**	64:22	38:29
schedule	**1:04**	**0:44**	5:32	8:18
totinfo	**0:51**	**0:33**	2:52	6:36
printtokens2	**1:19**	**2:51**	5:41	12:01
replace	**1:40**	**3:07**	20:20	25:40
printtokens	**3:16**	**8:25**	16:25	29:26
DAIDALUS	**33:18**	**16:43**	39:56	45:57
jDAIDALUS	**2:21:36**	**38:42**	2:47:31	2:51:59
gmat	10:30:06	**5:40:45**	**5:20:50**	7:20:29

Table 4 Per-slice average size reduction and run time (programs are ordered according to the mean time or mean percent reduction attained)

Slicing Time (sec)			Size Reduction		
Program	Time	Tukey	Program	Reduction	Tukey
jDAIDALUS	10191	a	tcas	84%	a
DAIDALUS	2396	b	hanoi	77%	ab
jTCAS	1255	b	DAIDALUS	73%	ab
jHanoi	782	b	jTCAS	73%	ab
jMBE	283	b	jHanoi	71%	abc
hanoi	55	b	jDAIDALUS	69%	bc
tcas	38	b	mbe	51%	c
mbe	18	b	jMBE	21%	d

Considering first the slicing time data, slicing Java typically takes considerably longer (with the exception of the C program DAIDALUS). Because DAIDALUS and jDAIDALUS are so much larger the comparison of all eight slice times is not very informative. For example, Tukey's HSD finds only that jDAIDALUS takes much longer to slice. Furthermore because of its size, it is reasonable to consider whether it is simply the overall size that causes this. In an attempt to account for the relative size of programs, Table 5 shows the effect of 'normalising' the slicing time. This normalisation divides each time by the number of source lines of code in each system. This gives a nominal measure of the slicing time per source line of code. This data more clearly shows that the Java programs take longer to slice than the C/C++ group. It is noticeable that jDAIDALUS and DAIDALUS have much lower ratios. This is perhaps related to T-ORBS ability to remove large subtrees in a single deletion, a behaviour that is impossible in the other programs because of their relatively small size.

Compared head-to-head, the slicing time for the C code is significantly less in each of the four comparisons. The *p*-value for comparing tcas and jTCAS is 0.0008, while the other

Table 5 Nominal slicing time per line of code (ordered from smallest to largest)

Program	Time/SLoC
DAIDALUS	0.11
tcas	0.28
mbe	0.31
hanoi	0.31
jDAIDALUS	0.49
jHanoi	4.73
jMBE	5.20
jTCAS	7.54

three are < 0.0001. Furthermore, the C code shows less variation over the range of programs considered; the range for C is less than a factor of three (2936/19 = 155) while for Java the range is almost double that of the C code (10051/39 = 258).

Moving on to the reduction attained, unlike the time taken by the slicer, the percent reduction attained by T-ORBS is independent of programming language. This can be seen in Table 4 where each pair of programs shares a letter except for mbe and jMBE. In this last case Java's higher overhead (e.g., a four line C program can print argv[0] while printing args[0] requires seven lines of Java) is significant given the small size of many of the slices. Comparing each pair head-to-head brings greater statistical power and finds a difference between the reduction attained between tcas and jTCAS (p-value = 0.0014) as well as mbe and jMBE (p-value = 0.048). Furthermore, it is interesting to note that the artificial Montréal Boat Example sees the smallest reduction where the Java version jMBE sees the least and is statistically less than all the other programs. This is an expected result because mbe is designed to illustrate the subtlety of dependence analysis and thus has tightly knit semantics. Finally, while not statistically significant, in each pairing, the C code shows the larger numeric reduction. This is largely due to C's "looser" semantics allowing type-unsafe deletions.

We repeated the analysis using the ORBS data for the same eight programs. The results are identical except that the size comparison yields slightly sharper statistics. Specifically the smaller percent reduction for jDAIDALUS when compared to DAIDALUS is statistically significant when using the ORBS data.

Finally, taking a qualitative look at some individual slices, the two languages show very little difference. This comparison does provide some insight into the larger relative reduction that hanoi shows when compared to that of jHanoi. The original Java code makes use of the standard Java stack library class. In contrast, the C translation implements a simple stack as part of the code. This stack code is not present in all slices and thus, on average, more code is removed from the C version than from the Java version.

In conclusion when considering RQ4, the source language brings little difference to the actual slices. In contrast, language plays a significant role in the slicing time. While not directly a language issue, one surprising result was that both ORBS and T-ORBS have a relatively lower time per source line for DAIDALUS and jDAIDALUS. This seems to suggest that it is not only T-ORBS' ability to remove large subtrees in a single deletion that accounts for this pattern.

6.5 RQ5: How does the modified T-ORBS algorithm perform?

This section considers the effect of the modifications to the core T-ORBS algorithm presented in Sections 3.1.1 and 3.1.2.

6.5.1 Size threshold

The effect of incorporating a size threshold into the T-ORBS algorithm (Algorithm 4) is investigated to determine what, if any, effect it has on the slicer performance and slice results. The initial investigation considers singleton thresholds such as -st 8, which ignores nodes that represent less than 8 lines of code, and then combinations such as -st 8,4,2,1. The initial experiment considers five different thresholds: 0, 1, 2, 4, and 8. The goal of this experiment is to understand the trade-off between slice size and slice time, with larger thresholds expected to be quicker in exchange for producing larger slices.

The results of this experiment are shown in Figure 6 and Table 6. Because the runtimes for the various programs vary so widely, the values for both time and slice-size are normalized to the time and size when using size threshold zero (which is equivalent to the original T-ORBS slicer). Considering first the slicing time in Table 6, a threshold of one reduces the average slicing time by over 70% to 29.4% of its original value. As can be seen in the top chart shown in Figure 6, this reduction is reasonably consistent across all of the programs. Going a step further, a threshold of two further reduces slicing time by approximately two-thirds of the threshold one value. Said another way, a threshold of one is 3.4 times faster that the original, while a threshold of two is 10.5 times faster (i.e., 3.1 times faster than threshold one). While thresholds four and eight bring further reduction, they are much less dramatic. Statistically, as seen in Table 6, the first two drops are statistically significant, while the shared letters indicate that this is less true for the larger thresholds.

Of course with each speed increase, the slicer is examining less of the tree and thus runs an ever greater risk of retaining larger portions of the code in the slice. Looking at the second chart in Figure 6, larger thresholds clearly lead to larger slices. What is interesting is that the thresholds clearly partition themselves with zero and one performing almost identically and then the other three all having notably worse, but similar, performance. This visual division is born out by the statistics where Tukey's HSD test, shown in Table 6, finds two separate groups. *The practical upshot is that T-ORBS with a threshold of one runs three times faster while leading to an insignificant increase in slice size.*

While the two charts at the top of Figure 6 show per-program summaries of the slices, the lower two charts show per-slice summaries. Because one of the jDAIDALUS slices takes over 18 hours to compute, a log scale is used on the y-axis of the third and fourth charts. The x-axis shows each slice sorted by the average slicing time over all five thresholds considered. Two things are evident in the figure: first, the trend lines show the same basic pattern as seen in the per-program data. Second, as the threshold grows, the variation in performance also grows.

The final chart shows the per-slice size data using the same order on the x-axis and again using a log scale on the y-axis. Here the trend lines clearly show the separation of the thresholds into two distinct bands. While the slice sizes show a much larger variation than the slice times, the summary here is the same as with the per-program data, a threshold of one is clearly the *sweet spot*. However, given the variation, it would be interesting to learn if there were good predictors of when a higher or lower threshold is preferable.

Summarizing the experiments using single thresholds, the "sweet spot" is clearly a threshold of one, which balances reduced slicing time against slightly larger slices. Expanding on this, the next experiment considers the value of an initial pass using a high threshold followed by a pass with a threshold of one. The question being considered is if it is possible to get the best of both worlds by combining an initial quick high-threshold pass to remove "most" of the code with the precision of a threshold one pass. Despite two, four, and eight

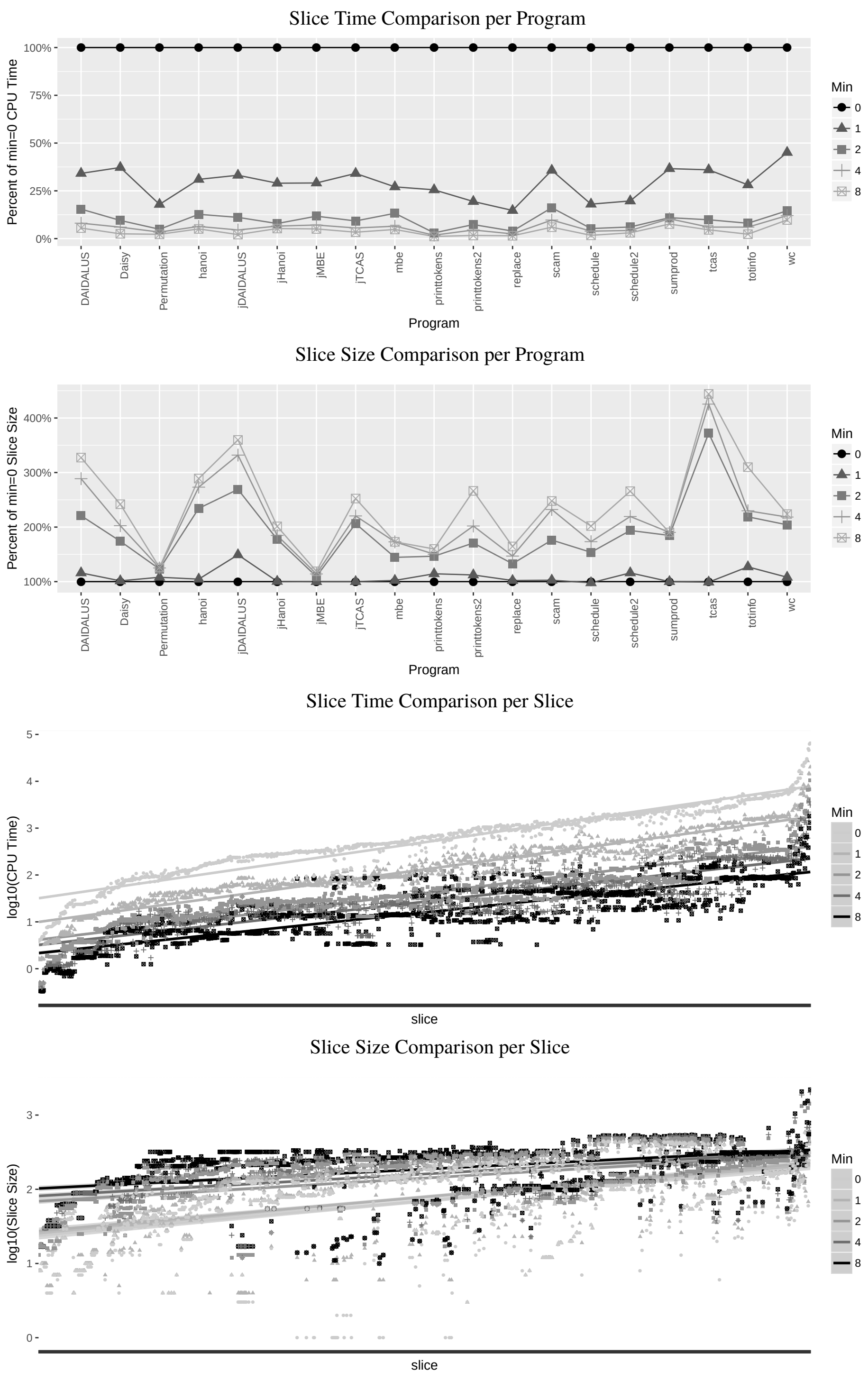

Fig. 6 By program and by slice data comparing the thresholds 8, 4, 2, 1, and 0.

Table 6 RQ5 data CPU time and slice size normalized to the -st 0 value.

Size Threshold	Percent CPU	Tukey HSD	Percent Slice Size	Tukey HSD
0	100.0%	a	100.0%	a
1	29.4%	b	108.5%	a
2	9.5%	c	190.2%	b
4	6.1%	cd	215.9%	b
8	4.0%	d	240.2%	b

Table 7 RQ5 data CPU time and slice size for threshold pairs, again normalized to the -st 0 value. (Note that the order of the size thresholds differs in the two comparisons.)

Size Threshold	Percent CPU	Tukey HSD	Size Threshold	Percent Slice Size	Tukey HSD
8,4,2,1	31.4%	a	1	108.5%	a
2,1	29.7%	a	8,1	108.1%	a
4,1	29.2%	a	4,1	107.9%	a
1	29.0%	a	2,1	107.7%	a
8,1	28.6%	a	8,4,2,1	107.7%	a

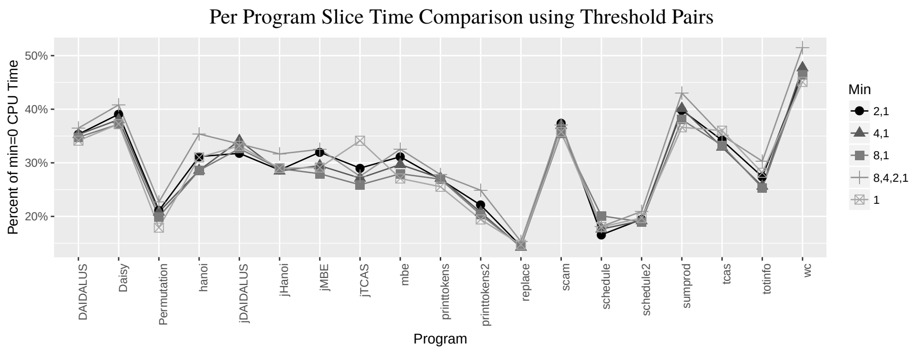

Fig. 7 By program slice size comparison using threshold pairs.

all having similar performance, for completeness, all three are paired with one in the next experiment, which also considers their combination -st 8,4,2,1

The resulting data is shown in Table 7, which, similar to Table 6, compares the slice time and reduction. The values are again normalized to those for a threshold of zero. Unlike Table 6, it does not work well to show the thresholds in the same order and thus the CPU time and the slice size are shown using different threshold orders.

The analysis of this data considers first the time taken and then the size reduction attained. Finally, it considers an illustrative example. Considering first the CPU time, statistically there is no difference between the five thresholds considered. Still, it is comforting to see that numerically, the expected pattern is seen with -st 8,4,2,1 taking the most CPU time, and -st 8,1 taking the least. Numerically, a first pass with a threshold of four or eight brings a time advantage, while a first pass with a threshold of two and all three thresholds brings a disadvantage. Figure 7 compares the times for the various programs. As captured by the statistics, there is no clear fastest option. The chart helps to illustrate why as it shows that the order of performance for a given program varies across the set of programs.

Turning to the slice size data again no statistical difference is seen. This is not unexpected as each slice ends with a threshold of 1. In fact, it is perhaps more surprising that the values are not all the same. While the variation is small (less than 1% from largest to smallest), different threshold values can impact the order that code is deleted and thus can impact the slices.

As an illustrative example consider the following code from printtokens where we have added comments to the two calls to print_token:

```
while(!is_eof_token((token_ptr=get_token(stream_ptr))))
{
    print_token(token_ptr); // call 1
}
print_token(token_ptr); // call 2
exit(0);
```

For a slice taken with respect to the sequence of tokens produced by get_token, the -st 8,1 slice includes

```
while(!is_eof_token((token_ptr=get_token(stream_ptr))))
{
}
```

and omits the definition of print_token. In comparison the -st 2,1 slice includes

```
while(!is_eof_token((token_ptr=get_token(stream_ptr))))
print_token(token_ptr); // call 2
```

The indentation here is important. Similar to the “capture” examples from Section 6.2, the while loop has captured the second call to print_token that was originally after the loop. The capture takes place during the first pass of the -st 2,1 slice, which skips the second call print_token because it is less than two lines of code. It subsequently removes the body of the loop leading to the capture. Once this is done, it cannot delete the captured call and thus must also retain the skeleton of the function print_token.

In contrast, the first pass of the slice where -st 8,1 skips the while loop (and its subtrees) because they represent less than eight lines of code. To understand the behaviour of the -st 8,1 slice, it is helpful to refer to the following srcml excerpt.

```
<while>while
  <condition> ...
  <block>{
    <expr_stmt><expr><call><name>print_token ...
  }</block>
</while>

<expr_stmt><expr><call><name>print_token ...
```

On the second pass of the -st 8,1 slice, again the slicer fails to delete the entire while loop because the condition is part of the slice. It thus places the subtrees rooted at <condition> and <block> on the worklist and moves on to the next worklist element, the call labelled // call 2, which T-ORBS can delete. Eventually the subtree rooted at <condition> reaches the font of the queue. It cannot be deleted. Next is the subtree rooted at <block>. Because there is no longer a capture-able statement following the call, this subtree cannot be deleted. So its (only) subtree, the <expr_stmt>, is placed on the worklist. Eventually, this subtree reaches the front of the queue. It can be successfully deleted yielding the final slice.

Table 8 The impact of subtree replacement

Size Threshold	Percent CPU	Tukey HSD	Size Threshold	Percent Slice Size	Tukey HSD
-1,1	33.8%	a	1	108.1%	a
1,-1	33.3%	a	-1,1	93.2%	b
1	29.0%	a	1,-1	87.8%	b

In the end the -st 8,1 slice includes two lines not in the -st 2,1 slice (the braces), while the -st 2,1 slice includes a call, the function header, and an empty body (two lines for the two braces). This leaves the -st 2,1 slice two lines longer.

In summary, using a threshold size of one leads to a dramatic improvement in slicing time without a significant increase in slice size. However, even the 5% improvement that comes from pairing thresholds of eight and one, does not yield a statistically significant improvement.

6.5.2 Subtree replacement

The effect of incorporating subtree replacement into the T-ORBS algorithm (Algorithm 5) is investigated to determine its effect. The replacement experiment considers the impact of an initial subtree replacement pass, denoted -1,1, and a final subtree replacement pass, denoted 1,-1. Note that the latter of these is more correctly labelled 1*,-1 as it slices with a threshold of one until no further reduction is possible and then it makes a subtree replacement pass. The difference between these two uncovers the impact of the subtree replacements enabled by slicing.

Table 8 shows the CPU time and size impact of the subtree replacement. As done in the other tables, the values are all relative to the original T-ORBS slicer. Considering, first the CPU time, it is possible that an initial subtree replacement would remove considerable code and thus speed the subsequent slicing. On the flip side, it is possible that the subtree replacement is so time consuming that any speed-up in the subsequent slicing is dwarfed by the cost of the replacement. As can be seen by comparing the first and the third lines of Table 8, the cost of the replacement dominates the overall cost. It is slightly more efficient to apply the subtree replacement at the end of the computation (compare lines two and three) because the sliced code is smaller and thus the expense less.

Turning to the reduction achieved by the slicer, the subtree replacement brings a statistically significant reduction in slice size when applied *after* slicing. Compared directly to slicing with a threshold of one, an initial subtree replacement pass reduces slice size by 13.8%. Furthermore, because the slicing enables additional replacements, when the subtree replacement is run after slicing, it yields a 18.8% reduction; thus slicing has enabled an additional 5.0% reduction. Given that running the replacement at the end of the slice also means not having to consider deleted parts of the code, the data supports the use of subtree replacement as a post-slicing pass. Finally, while it is possible to consider subtree replacement in the "middle" given the CPU cost, on average doing so never led to an improvement.

Figure 8 shows the slice sizes for the three subtree-replacement options studied. While the overall order seen in Table 8 is evident in the chart, it is interesting that for certain programs (e.g., jTCAS) performing an initial subtree replacement produces noticeably larger slices. Finally, applied after slicing, subtree replacement always reduces slice size.

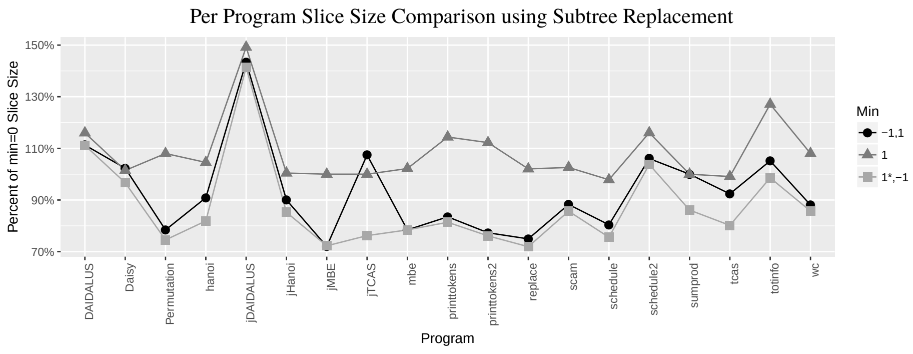

Fig. 8 By program slice size comparison using threshold pairs.

Finally, to gain some intuition for the interplay between the subtree replacement and slicing, we consider a representative example from mbe. The relevant code is from the main function:

```
while (p(j))
{
  if (q(k))
  {
    k = f1(k);
  }
  else
  {
    k = f2(k);
    j = f3(j);
  }
}
```

The example considers the slice taken with respect to the value of j right after the assignment j = f3(j) and hinges on the amount of code in the true branch. If there is sufficiently little code in this branch that an initial subtree replacement is possible, it succeeds yielding

```
while (p(j))
{
  k = f2(k);
  j = f3(j);
}
```

Subsequent slicing removes the computation of k. This slice is interesting because applying subtree replacement at the end produces the same main function, but retains the definition of the function q. Applying subtree replacement at the end removes the call found in if (q(k)), but it would require a subsequent slicing pass to remove q's definition. Such a pass is possible, but comparatively expensive relative to the reduction achieved. Thus in this case, performing subtree elimination before slicing yields the greater reduction.

In contrast, if in the original program there was more code in the true branch, then the initial subtree replacement would not be applicable. However, if this code was removed by the slicer, then slicing would have enabled subtree replacement if the replacement was applied after slicing. This second situation is the more common and thus, as seen in Table 8, 1,-1 yields a greater reduction than -1,1.

While mostly cosmetic, it is interesting that the subtree replacement enables T-ORBS to produce what might be considered *more natural* slices. For example, slicing on the value of j after the loop, T-ORBS when using 1,-1 produces the final slice

```
while (p(j))
  j = f3(j);
```

while ORBS produces the slightly odd looking

```
while (p(j))
{
  {
    j = f3(j);
  }
}
```

Several other engineering improvements have been incorporated into T-ORBS. We describe one of them, the *parent trap* as a representative example. It is not uncommon for several nodes in the tree to have a single descendant. For example, consider the srcml for a function call: <expr_stmt><expr><call> ... </call></expr>;</expr_stmt>. If omitting the <expr_stmt> node fails, the slicer places all its children on the worklist (in this case, the <expr> node). Since these two nodes lead to the same source code, the parent's failure predicts that of the child and consequently, there is no need to attempt the deletion of the child.

In summary, RQ5 investigates the effect of two modifications to the T-ORBS algorithm: the addition of a size threshold, and subtree replacement. While neither of these improvements led to clearly faster slicing **and** smaller slices, a size threshold of one captures an excellent engineering trade-off where the slicer runs over 70% faster while producing slices that are only about 10% larger. Likewise, the subtree replacement actually led to (statistically insignificant) time increase, in exchange for a statistically significant reduction in slice size. On the qualitative front, the subtree replacement also leads to more natural looking slices.

6.6 Summary

From the examples presented to study RQ2 and the data considered to address RQ1 and RQ3, is clear that each slicer has its own pros and cons. In general, the two produce similar slices where T-ORBS slices can be slightly larger because they must maintain the XML tree structure. However, the larger slices produced by T-ORBS are often the more intuitive. On the other hand, T-ORBS can perform "sub-line" deletion, which, as shown in Section 6.2, can be both a blessing and a curse.

The experiments performed for RQ4, comparing of Java and C programs, showed that the source language brings little difference to the actual slices. In contrast, language plays a significant role in terms of the slicing time. As slicing time for T-ORBS is usually larger, RQ5 investigated two improvements to the tree-based slicer, the addition of a size threshold and subtree replacement, and showed that, among other things, the improvements cause the slicer to run 3.4 times faster while producing more natural slices that have the same or even slightly smaller size.

6.7 Threats to Validity

Threats to internal validity concern the factors that may have incorrectly biased conclusions claimed by this study. The primary threat to internal validity is the correctness of implementations including the conversion to tree structures and program instrumentation for both ORBS and T-ORBS. For the conversion, we rely on srcML (Collard, 2005) that is actively being maintained and has stood against the scrutiny of many users and researchers. We have manually inspected program instrumentation to ensure their correctness.

Threats to external validity concern any factor that may limit the extent we can generalise our findings. To increase the representativeness of studied programs, we use a wide range of programs ranging from toy examples for which complete analysis is possible, to production systems with millions of lines of code. Another threat to external validity is the extent to which the same programs written in different languages share the same structural properties. For example, jHanoi and hanoi, written by the same author, use different stack implementations: standard Java Stack for the former, while a lightweight independent implementation for the latter. We posit that it is possible to capture the *typical* properties of different languages via examples, and have tried to avoid any further bias by using examples written by programmers other than authors whenever possible. In case of alternative programs in different languages written by authors, two versions have been written independently. Finally, we tried to avoid experimental bias in the selection of slicing criteria by performing exhaustive slicing using all possible criteria.

Threats to construct validity concern the question of whether the experimental results are based on observation of factors that actually reflect our claims. Both lines of code and number of tree nodes are straightforward counting metrics that precisely reflect the effectiveness of slicing, and have been used in the literature extensively. To prevent line counts being affected by layouts, we automatically formatted the source code of all programs (except for the production systems) in the same style. Consequently, differences in layout do not impact our results. While there are various line counting schemes (such as counting only executable lines, or including/excluding whitespace only lines or comments), we do not think the choice of line counting scheme can affect our results, as software size related metrics are highly correlated with each other (Mamun et al, 2017).

7 Related Work

Static slicing was first introduced by Weiser (1981). Subsequently, Ottenstein and Ottenstein (1984) proposed that program slicing can be formulated as a graph reachability problem on the Program Dependence Graph (PDG). Horwitz et al (1988) introduced an algorithm which extended program slicing to be interprocedural using the System Dependence Graph (SDG) as the representation. Horwitz et al. also introduced a two-pass static slicing (Horwitz et al, 1990), an algorithm that remains as the most predominantly used and whose variants are widely studied.

Many flavours of static slicing algorithm attempted to reduce the size of the slice. Incremental Slicing (Orso et al, 2001) allows selection of the type of data dependencies that are to be included in a slice. Stop-list slicing (Gallagher et al, 2006) allows the programmer to define variables that are out 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 PDG that cannot be passed during the graph traversal, also resulting in focused and smaller slices. Use of path-sensitivity analysis (Jaffar et al, 2012) with static slicing is another approach to reduce slice sizes by removing infeasible paths. However, such techniques suffer from their combinatorial nature and can only work precisely in the absence of certain constructs that lead to combinatorial explosion, such as loops.

Amorphous Slicing (Harman and Danicic, 1997) is an approach that aims to preserve the semantics of the program during slicing, but not the syntax. Amorphous slices use program transformation to simplify programs, preserving the semantics of the program with respect to the slicing criterion. While ORBS only transforms programs using deletions, the end result may be merging between remaining program elements, which could be regarded as a form of (very slightly) amorphous slicing.

To our knowledge no other slicing approach follows the observational statement-deletion approach used by ORBS. The ORBS algorithm (Binkley et al, 2014) is a dynamic form of slicing but it constructs slices using dynamically *observed* dependencies, rather than dynamically *occurring* yet statically determined dependencies. Note that all other dynamic slicing approaches rely on the statically determined dependencies.

Dynamic slicing is a concept introduced by Korel and Laski (1988, 1990). They 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. For example, WebSlice (Nguyen et al, 2015) attempts to perform program slicing across different languages for web applications, by identifying data-flow dependencies among data entities for PHP code based on symbolic execution and connecting them to data flows in embedded language: SQL, HTML, and JavaScript. Despite being multi-language, it is specific to these languages. The observational nature of ORBS, on the other hand, allows it to slice programs constructed from an unspecified set of multiple program languages (Binkley et al, 2014). Of all previous dynamic slicing formulations, the closest to our observation-based slicing is Critical Slicing (DeMillo et al, 1996). However, our previous empirical study has shown that critical slices are not only significantly larger than observation-based slices, but also often incorrect (Binkley et al, 2014).

The idea of deleting parts of a program or test input also prominently features 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 for a tree-based Delta Debugging approach: its relationship to the original Delta Debugging is similar to that of T-ORBS to the original ORBS. 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. They also introduce the concept of *generalised delta debugging* that allows any iterative optimization strategy, including other transformations than deletion. Their C-Reduce tool uses 30 source-to-source transformations for C code. 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.

Perses (Sun et al, 2018) exploits the formal syntax to guide the reduction of programs. It uses deletion and subtree replacement as operations and applies Delta Debugging to sequence of child nodes. Similarly to observational slicing, it allows any program property to be defined as a criterion. T-ORBS is more general than Perses as it is not restricted to formal languages and can reduce any tree representation expressed in XML.

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. However, their forward dynamic slicing suffers from low recall of what they call *dynamic semantic dependencies*, which can have serious effects on impact analysis.

Finally, union slicing (Beszédes et al, 2002) is also related to ORBS. Union slicing approximates a static slice by unioning dynamic slices obtained with a set of test 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 all criteria without unioning.

8 Conclusion

Observational slicing is a new slicing technique that constructs slices using the dependencies observed during execution. Previous work includes the comparison of observation-based slicing to traditional static (Binkley et al, 2015) and to dynamic (Binkley et al, 2014) slicing techniques, as well as application of observation-based slicing to domains with non-traditional semantics such as visual languages (Yoo et al, 2014) and modelling languages (Gold et al, 2017). In particular, the development of a slicer for modelling languages led to the creation of an observational slicer for XML trees. Closing the loop, this paper applies the XML tree slicer to source code that has been transformed into XML using srcML. The original ORBS uses physical lines as the unit of speculative deletion, while T-ORBS uses subtree deletion: both use observation of program execution to validate slicing criteria. Our aim is to compare the two slicers to better understand the pros and cons of each representation and approach.

Our empirical comparison using twenty target programs shows that, while ORBS and T-ORBS produce largely comparable slices, there are subtle differences due to both structural constraints and opportunities imposed and revealed by each representation (i.e., lines versus trees). Based on our analysis, we investigate some of the opportunities in the form of size threshold and subtree replacement for T-ORBS. Overall, the results of our investigation hint at the rich diversity of possible *language-independent* slicing strategies. Furthermore, they open the door for the future study into the impact that variations might have as complements to existing slicing techniques.

Future work will consider the hybridisation between ORBS and T-ORBS. For example, our results suggest that T-ORBS tends to be more effective at deleting large blocks of code. It is natural to imagine a hybrid slicer that initially applies T-ORBS to only subtrees that represent "large" blocks, followed by one or more ORBS passes to handle code elements that T-ORBS cannot, such as #ifdef (because directives are each in their own subtree, T-ORBS cannot delete matching pairs of #ifdef / #endif). A final T-ORBS pass that might only consider subtrees that represent only "small amounts" of code would serve to simplifying existing lines as illustrated in Section 6.2.

9 Acknowledgements

A special thanks to Mark Harman for many interesting conversations on the use of observational slicing. Dave Binkley is supported by NSF grant 1626262. Shin Yoo is 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). The authors acknowledge the use of the UCL Legion High Performance Computing Facility (Legion@UCL), and associated support services, in the completion of this work. They also thank the Software Infrastructure Repository for the provision of subject programs.

References

Agrawal H, Horgan JR (1990) Dynamic program slicing. In: Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation (PLDI)

Barpanda SS, Mohapatra DP (2011) Dynamic slicing of distributed object-oriented programs. IET software 5(5)

Beszedes A, Gergely T, Szabó ZM, 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

Beszédes Á, Faragó C, Szabó ZM, Csirik J, Gyimóthy T (2002) Union slices for program maintenance. In: Proc. of the 18th Intl. Conf. on Software Maintenance (ICSM)

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)

Binkley D, Capellini R, Raszewski L, Smith C (2001) An implementation of and experiment with semantic differencing. In: Proceedings of the 2001 IEEE International Conference on Software Maintenance, pp 82–91

Binkley D, Harman M, Hassoun Y, Islam S, Li Z (2009) Assessing the impact of global variables on program dependence and dependence clusters. Journal of Systems and Software 83(1)

Binkley D, Gold N, Harman M, Islam S, Krinke J, Yoo S (2014) ORBS: Language-independent program slicing. In: Proc. 22nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering

Binkley D, Gold N, Harman M, Islam S, Krinke J, Yoo S (2015) ORBS and the limits of static slicing. In: Intl. Working Conference on Source Code Analysis and Manipulation (SCAM)

Binkley D, Gold N, Islam S, Krinke J, Yoo S (2017) Tree-oriented vs. line-oriented observation-based slicing. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM)

Cleve H, Zeller A (2000) Finding failure causes through automated testing. In: Intl. Workshop on Automated Debugging

Collard M (2005) Addressing source code using srcml. In: IEEE International Workshop on Program Comprehension Working Session (IWPC'05)

Danicic S, Howroyd J (2002) Montréal boat example. In: Source Code Analysis and Manipulation (SCAM 2002) conference resources website, URL `http://www.ieee-scam.org/2002/Slides_ct.html`

Danicic S, Harman M, Hierons R, Howroyd J, Laurence M (2004) Applications of linear program schematology in dependence analysis. In: $1^{st\cdot}$ International Workshop on Programming Language Interference and Dependence, Verona, Italy, URL `http://profs.sci.univr.it/~mastroen/noninterference.html`

DeMillo RA, Pan H, Spafford EH (1996) Critical slicing for software fault localization. In: Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA)

Gallagher KB, Lyle JR (1991) Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17(8)

Gallagher KB, Binkley D, Harman M (2006) Stop-list slicing. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM)

Gold NE, Binkley D, Harman M, Islam S, Krinke J, Yoo S (2017) Generalized observational slicing for tree-represented modelling languages. In: Proc. 25nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering

Harman M, Danicic S (1997) Amorphous program slicing. In: 5^{th} IEEE International Workshop on Program Comprenhesion (IWPC)

Harman M, Binkley D, Gallagher K, Gold N, Krinke J (2009) Dependence clusters in source code. ACM Transactions on Programming Languages and Systems 32(1):1:1–1:33

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 DW (1988) Interprocedural slicing using dependence graphs. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation

Horwitz S, Reps T, Binkley DW (1990) Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12(1)

Islam S, Binkley D (2016) PORBS: A parallel observation-based slicer. In: 24th International Conference on Program Comprehension (ICPC), IEEE, pp 1–3

Jaffar J, Murali V, Navas J, Santosa AE (2012) Path-sensitive backward slicing. In: Proc. SAS'12, Springer, vol 7460

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)

Korel B, Laski J (1988) Dynamic program slicing. Information Processing Letters 29(3)

Korel B, Laski J (1990) Dynamic slicing in computer programs. Journal of Systems and Software 13(3)

Krinke J (2003) Barrier slicing and chopping. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM)

Laurence MR (2004) Equivalence of linear, free, liberal program schemas is decidable in polynomial time. PhD thesis, Goldsmiths College, University of London

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

Mamun MAA, Berger C, Hansson J (2017) Correlations of software code metrics: An empirical study. In: Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, pp 255–266

McPeak S, Wilkerson DS, Goldsmith S (2006) Delta (`http://delta.tigris.org`). URL `http://delta.tigris.org`

Misherghi G, Su Z (2006) HDD: hierarchical delta debugging. In: Proc. of the 28th Intl. Conf. on Software Engineering (ICSE)

Mund G, Mall R (2006) An efficient interprocedural dynamic slicing method. Journal of Systems and Software 79(6)

NASA (2017a) DAIDALUS: Detect and avoid alerting logic for unmanned systems. https://github.com/nasa/WellClear

NASA (2017b) GMAT: Generalised mission analysis tool R2017a. https://sourceforge.net/projects/gmat/files/GMAT/GMAT-R2017a/

Nguyen HV, Kästner C, Nguyen TN (2015) Cross-language program slicing for dynamic web applications. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp 369–380

Orso A, Sinha S, Harrold MJ (2001) Incremental slicing based on data-dependences types. In: Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM)

Ottenstein KJ, Ottenstein LM (1984) The program dependence graph in software development environments. In: Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment

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)

Reps T, Turnidge T (1996) Program specialization via program slicing. In: Danvy O, Glück R, Thiemann P (eds) Dagstuhl Seminar on Partial Evaluation, vol 1110

Sun C, Li Y, Zhang Q, Gu T, Su Z (2018) Perses: Syntax-guided program reduction. In: Proc. of the 40th Intl. Conf. on Software Engineering (ICSE)

Szegedi A, Gyimóthy T (2005) Dynamic slicing of Java bytecode programs. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM)

Tukey JW (1949) Comparing indirect means in the analysis of variance. Biometrics 5(99)

Ward M (2003) Slicing the SCAM mug: A case study in semantic slicing. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM)

Weiser M (1981) Program slicing. In: Proc. of the 5th Intl. Conf. on Software Engineering

Weiser M (1982) Programmers use slices when debugging. Communications of the ACM 25(7)

Yoo S, Binkley D, Eastman RD (2014) Seeing is slicing: Observation based slicing of picture description languages. In: Intl. Workshop on Source Code Analysis and Manipulation (SCAM), pp 175–184

Zeller A (1999) Yesterday, my program worked. today, it does not. Why? In: European Software Engineering Conf. and Foundations of Software Engineering

Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering 28(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

Zhang X, Gupta N, Gupta R (2007) A study of effectiveness of dynamic slicing in locating real faults. Empirical Software Engineering 12(2)