Hindawi
Scientific Programming
Volume 2019, Article ID 4108652, 15 pages
https://doi.org/10.1155/2019/4108652

Research Article

Automatic Testing of Program Slicers

Sergio Pérez, Josep Silva, and Salvador Tamarit

Universitat Politècnica de València, Camí de Vera s/n, E-46022 València, Spain

Correspondence should be addressed to Salvador Tamarit; tamarit27@gmail.com

Received 4 November 2018; Accepted 22 January 2019; Published 25 February 2019

Academic Editor: Michele Risi

Program slicing is a technique to extract the part of a program (the slice) that influences or is influenced by a set of variables at a given point (the slicing criterion). Computing minimal slices is undecidable in the general case, and obtaining the minimal slice of a given program is normally computationally prohibitive even for very small programs. Therefore, no matter what program slicer we use, in general, we cannot be sure that our slices are minimal. This is probably the fundamental reason why no benchmark collection of minimal program slices exists. In this work, we present a method to *automatically* produce quasi-minimal slices. Using our method, we have produced a suite of quasi-minimal slices for Erlang that we have later *manually* proved they are minimal. We explain the process of constructing the suite, the methodology and tools that were used, and the results obtained. The suite comes with a collection of Erlang benchmarks together with different slicing criteria and the associated minimal slices.

1. Introduction

In the areas of scientific and engineering computing, it is common the use of different program transformations to change an algorithm several times until certain performance requirements are met. One of these transformations is *program slicing*. Program slicing is a technique for program analysis and transformation whose main objective is to extract from a program those statements (the *slice*) that influence or are influenced by the values of one or more variables at some point of interest, often called the *slicing criterion* [1–4]. This technique has been adapted to practically all programming languages, and it has many applications such as debugging [5], program specialization [6], software maintenance [7], and code obfuscation [8].

In the general case, determining the minimal slice is undecidable [1]. For this reason, almost all program slicing techniques guarantee that their computed slices are *complete* (i.e., they contain all statements that do influence the slicing criterion), but, in general, they do not guarantee that their computed slices are *sound* (i.e., they probably contain statements that do not influence the slicing criterion).

Example 1. Consider the programs in Figure. We can define the slicing criterion $\langle 7, x\rangle$ in the original program. This means that we are interested in all statements that are needed to compute the value of x in line 7. The original code is a slice of itself, but there exist smaller slices. For instance, the code in the middle is the slice computed by almost all current static program slicers (e.g., this is the output of the Indus Java slicer [9] and CodeSurfer [10]). However, the slice in the middle is not minimal. The minimal slice of the original program is the code on the right. It would be difficult for a slicer to compute this slice because line 7 is reachable via a control-flow path from line 6 and line 6 defines variable x, which is used in line 7. Thus, most slicers consider that line 6 does influence line 7. This reasoning is transitively applied to lines 4 and 6. Hence, program slicers produce the code in the middle.

Example 1 illustrates how a tiny program without method calls and even without loops, cannot be handled precisely by current program slicers. The fact that computing minimal slices is undecidable in the general case does not, however, prevent us from defining a procedure to compute minimal slices for a given concrete program. Nevertheless,

	Original Program	Slice	Minimal Slice
1	*read (z)*	*read (z)*	*read (z)*
2	*x = 42;*	*x = 42;*	*x = 42;*
3	*y = 1;*		
4	*x++;*	*x++;*	
5	*if (z>0) {*	*if (z>0) {*	*if (z>0) {*
6	*x--;*	*x--;*	
7	*print (x);*	*print (x);*	*print (x);*
8	*...*		
9	*}*	*}*	*}*
10	*...*		

Figure 1

normally, even for very small programs, this procedure would be computationally intractable [11, 12]. Unfortunately, human intervention is often needed to produce minimal slices, and this is only practical for small programs.

1.1. Motivation. Being able to compute minimal slices would speed up many software processes. For instance, compilers use program slicing to remove dead code, and many analyses use program slicing as a preprocessing stage to detect variable dependencies. Therefore, making slicing more accurate would also improve the later analyses based on it.

Because computing minimal static slices is undecidable, in this work, we propose a method to compute quasi-minimal slices, which, roughly, are minimal slices for a given set of inputs (this means that quasi-minimal slices may not be sound static slices, i.e., for all possible sets of inputs). In many cases, we are interested in producing a slice with respect to a given computation (known as *minimal dynamic slice*). For instance, in debugging, we are often interested in producing a slice of a program that produced an error *for a particular input* because the slice produced is a reduced version of the program that reproduces the wrong computation (and that contains the error). In regression testing, after we test a new release of a program with the regression tests, many different errors can show up. In this situation, we can be interested in producing a slice *for a given set of test cases* (known as *minimal simultaneous dynamic slice*).

Our method produces minimal dynamic slices and simultaneous dynamic slices, and it can also produce static minimal slices in many cases. On the one hand, if the input domain of a program is finite, we automatically produce all possible input values, thus, producing a minimal static slice. On the other hand, if the input domain is infinite, we provide an instrumentation based on concolic testing to produce test cases that explore all possible branches (100% branch coverage) of the program. This ensures in many cases that the produced static slice is also minimal.

We have used our method to produce a suite of benchmarks with minimal slices. This has shown that quasi-minimal slices are often minimal slices. In fact, we have produced a suite of 23 quasi-minimal slices, and we have proven that all of them are actually minimal slices.

From the best of our knowledge, there does not exist any public repository of benchmarks with minimal slices, and this is surprising because a suite of minimal slices is very useful for slicer developers. In particular, we have implemented several program slicers for different languages, including Petri nets [13], XQuery [14], Erlang [15], and CSP [16]. Every time we improved our program slicer (e.g., with a new technique or feature or just to correct some bug), we found the same problem: we could not measure the improvement achieved with that change. What we often do is to implement some benchmarks and compare our previous results with the new ones. This gives a measure of improvement. But, it would be much more useful to start a battery of tests that automatically compare the new slices produced by our released code with a gold standard (i.e., the minimal slices). This would allow us not only to objectively measure the improvement of the new release but also to detect possible introduced problems in other parts of the slicer and, e.g., to fairly compare our tool with other tools.

As an application of our method to produce quasi-minimal slices, in this work, we also present the first fully automated system to evaluate and compare program slicers. This system inputs a program slicer and outputs a report about precision and recall of this slicer with respect to a suite of minimal slices that have been already computed. The system can also input two slicers and automatically compare them. If the two slicers are two releases of the same slicer, then the system can not only measure the improvement achieved but also identify errors introduced (or solved) in the new release.

1.2. Contributions. The main contribution of this work is a method for generating quasi-minimal slices, which has been later instantiated for Erlang and used to generate a suite of benchmarks composed of programs together with their minimal slices. The contributions of this work are summarized below:

(i) A method to obtain quasi-minimal slices.

(ii) An adaptation of *observation-based slicing* (ORBS) [12] to work with abstract syntax trees (ASTs). This maximizes precision, allowing us to slice at the level of literals.

(iii) A generalization of ORBS. ORBS is not correct in all cases. This problem is identified and solved in our approach.

(iv) An implementation of the proposed method for Erlang, producing a new program slicer for Erlang.

(v) A suite of benchmarks with challenging program slicing problems together with their minimal slices. The suite includes a tool that can be used to evaluate a program slicer against the suite.

2. Preliminaries and Notation

This section introduces some preliminary definitions and notation that are used along the paper. Because there exist several different notions of slice and minimal slice in the literature, to make things concrete, we need to provide a formal definition on which we will base the rest of the paper.

Program slicing is based on a slicing criterion over which the slice is obtained. This slicing criterion traditionally corresponds to a statement in the code and a variable within

that statement. However, if we use statements in our definitions, we could not be as precise as we want to be. Therefore, we base our slicing criterion on expressions, which do not impose that precision barrier.

Definition 1 (slicing criterion). Let P be a program. A slicing criterion C of P is an expression in P whose evaluation produces a value.

Note that even though most program slicers are based on statements, we do not have to restrict ourselves to that precision level. Moreover, any variable v in P can be considered a slicing criterion in our definition because variables are expressions, but we also allow for defining other slicing criteria such as results of operations (e.g., an addition), values to be assigned, values returned by procedure calls, and values of literals.

Our method combines static and dynamic slicing and, thus, we also need a definition of dynamic slicing criterion based on expressions, as well as a definition for the sequence of values the dynamic slicing criterion is evaluated to.

Definition 2 (dynamic slicing criterion). Let P be a program. A dynamic slicing criterion of P is a tuple $\langle C, I\rangle$ such that C is a slicing criterion and I is an input for P.

Definition 3 (sequence of values). Let P be a program and $\langle C, I\rangle$ be a dynamic slicing criterion of P. $seq(P, C, I)$ is the sequence of values the slicing criterion C is evaluated to during the execution of P with I.

First of all, it is important to remark that we use the standard definition of slice, which excludes nonterminating and nondeterministic programs. A justification of the necessity of these exclusions can be found in the seminal paper by Weiser [1]. Another important property is that we want our slices to be executable so that the execution of the slice for any given input must evaluate the slicing criterion as many times (or more) as the original code, and the sequence of values the slicing criterion is evaluated to when executing the original code must be equal to (or a prefix of) the sequence obtained at the slice. Formally, the definition is given as follows:

Definition 4 (static executable program slice (based on [3, 12])). A static executable program slice S of program P with respect to a slicing criterion C is any executable program with the following properties:

(1) S can be obtained by deleting code from P (denoted $S \subseteq P$).

(2) For all input I, $seq(P, C, I)$ is a prefix of $seq(S, C, I)$.

We define a dynamic executable program slice as an executable program slice for a given set of inputs. Formally, the definition is given as follows:

Definition 5 (dynamic executable program slice). A dynamic executable program slice S of a program P on a dynamic slicing criterion $\langle C, I\rangle$ is any executable program that fulfils the two properties of Definition 4 for P with respect to C and for only one specific input I.

According to Definitions 4 and 5, every program itself has as an executable program slice under any criteria.

From here on, given a program P and a slicing criterion C for P, we use the domain $\mathcal{S}lices_C^P$ to denote the finite set containing all possible slices of P with respect to C. We also use the notation $slice_X(P, C)$ to refer to the slice of P with respect to C computed with a specific slicer X.

Definition 6 (minimal slice). A minimal slice of program P with respect to a slicing criterion C is any $S \in \mathcal{S}lices_C^P$ such that $\nexists S' \in \mathcal{S}lices_C^P \wedge S' \subset S$.

Note that a minimal slice, according to this definition, is not necessarily unique and is not necessarily a slice with the smallest number of expressions (e.g., [17]). Because computing minimal slices is undecidable, similarly to [12], we can relax its definition to be minimal with respect to a finite set of inputs. Formally, the definition is given as follows:

Definition 7 (quasi-minimal slice). Let $\mathcal{I}$ be a set of possible inputs for a program P and C be a slicing criterion for P. A quasi-minimal slice (QM-slice) $qm\text{-}slice(P, C, \mathcal{I})$ of P with respect to C and $\mathcal{I}$ is a dynamic executable program slice of P that is minimal for all $I \in \mathcal{I}$ on a dynamic slicing criterion $\langle C, I\rangle$. If $\mathcal{I}$ contains all possible inputs of P, then $qm\text{-}slice(P, C, \mathcal{I})$ is a minimal slice of P with respect to C.

In the method proposed in this paper, besides a program P, a slicing criterion C, and a set of inputs $\mathcal{I}$, we also associate slices with the AST of P. This is particularly useful to allow us to reason about the accuracy of slices (Section 3). Therefore, we need to adapt the notion of slicing criterion to ASTs. This can be easily done by redefining a slicing criterion in such a way that the point of interest is not an expression but the AST node whose subtree represents that expression. We define the slicing criterion and the dynamic slicing criterion in terms of ASTs as follows:

Definition 8 (AST-adapted slicing criterion). Let P be a program and C be a slicing criterion of P. Let $AST(P) = (N, E)$ be an AST of P where N is the set of nodes and E is the set of edges. n is the AST-adapted slicing criterion of C such that $n \in N$ and n is the root of the subtree of $AST(P)$ that represents C.

Definition 9 (AST-adapted dynamic slicing criterion). Let P be a program and $\langle C, I\rangle$ be a dynamic slicing criterion of P. An AST-adapted dynamic slicing criterion of $\langle C, I\rangle$ is a tuple $\langle n, I\rangle$ such that n is the AST-adapted slicing criterion of C.

3. Focussing on Fine-Grained Slices

Program slices are often measured in code lines. The reason is that most program slicing techniques consider lines of code as atomic elements and, thus, they remove a whole line

or nothing [1, 5, 12]. For this reason, most of the work that compares the precision of different program slicing techniques just compares the retrieved number of lines (e.g., [12, 18]). Unfortunately, this is very sensitive to the programming style, and moreover, it can be very imprecise, especially in functional languages.

Example 2. Consider the Erlang program in Figure 2(a) and its minimal slice 2(b) with respect to the slicing criterion $\langle 10, B\rangle$. Observe that some expressions have been replaced by _ or by the fresh atom sliced ([15, 19]). This is needed to make the slice executable. Clearly, all methods based on lines would not be able to remove the subexpressions that are not needed in lines 1, 2, 3, and 6. For instance, in line 2, $C = B$ can be removed, but the programmer initialized A, B, and C in a single line, and thus the whole line cannot be removed. One can argue that a preprocessing phase could be used to refactor the code and place all statements in different lines whenever it is possible. But, this cannot solve the second problem: sometimes only a subexpression can be removed in a line. This is the case of variables Z, Y, and C in line 3.

To overcome these limitations, as already done by, e.g., CodeSurfer [10] or in [15], in our method, we propose to use expressions as the slicing criterion, so precision can be increased. We also reason about slices at the AST level so that instead of counting lines of code, we can measure the number of AST nodes that belong to the slices, thus producing a more precise measure. However, note that this is a generalization, i.e., those program slicers that work directly on lines or statements (e.g., [9]) are an instance of our model because they remove subtrees of the AST that corresponds to lines or statements. That is, a line/statement is represented in the AST with a single node (and its subtree). This reasoning is also applicable to those program slicers that base their slices over other elements such as procedures and expressions or even AST nodes (e.g., [10, 15, 20]).

4. A Method to Produce Quasi-Minimal Slices

Given a program and a slicing criterion, our method computes its QM-slice (Definition 7) following two sequential phases. The first phase produces a static slice of the original program, which is the input of the second phase. The second phase further slices this slice, producing the final QM-slice. Figure 3 summarizes the method, which is explained in the following subsections.

4.1. Phase 1: Combining Static Program Slicers. In the first phase, we use a set of static program slicers to repeatedly slice the original program until a fix point is reached. Different program slicers usually implement different techniques and optimizations to reduce the size of the slice. Therefore, we can use any program slicer to produce a first slice that we can use as the starting point to further reduce its size with another program slicer because the slice of a slice is a slice provided that the same slicing criterion is used.

Theorem 1. *Let P be a program and $S = slice_{X_1}(P, C)$ be a program slice. Then, $S' = slice_{X_2}(S, C) \in \mathcal{S}lices_C^P$ for any P, C, X_1, and X_2.*

Proof. By point 4 in Definition 4, we know that $S' \subseteq S \subseteq P$. By point 4 in Definition 4, we know that $\forall I : seq(P, C, I)$ is a prefix of $seq(S, C, I)$ and that $seq(S, C, I)$ is a prefix of $seq(S', C, I)$. Therefore, $seq(P, C, I)$ is also a prefix of $seq(S', C, I)$. Hence, $S' \in \mathcal{S}lices_C^P$. □

Therefore, given a program P and a slicing criterion C, *slicer B* can use the slice provided by *slicer A* as its input and take advantage of the code removed by A. However, A may also take advantage of the code removed by B and thus remove code it did not remove the first time, which would imply that A can take further advantage of the new code removed. Therefore, a loop between all the slicers is needed until none of them can further remove any additional code, thus reaching a fix point.

One important property of the slicers, which is a requirement of the method, is that the slices produced by all of them must be complete (Note that, in our context (according to Definitions 4 and 5), a slice is always complete. However, not all program slicers produce complete slices. Some slicers such as [21] only ensure soundness. Therefore, in this paper, *"complete slice"* should be read as just *"static slice"*). Therefore, the output of *Phase* 1 is always a *complete slice*, because the sequential composition of *complete slicers* produces a *complete slicer*.

Theorem 2 (completeness). *Let P be a program, and let C be a slicing criterion for P. Given two complete program slicers X_1 and X_2, then $slice_{X_2}(slice_{X_1}(P, C), C)$ is a complete slice with respect to P and C.*

Proof. First, because X_1 is complete, we know that $S_1 = slice_{X_1}(P, C)$ is a complete slice with respect to P and C. We prove the theorem by contradiction assuming that the slice $S_2 = slice_{X_2}(slice_{X_1}(P, C), C)$ is not complete with respect to P and C. This is only possible if either X_2 is not complete and thus $slice_{X_2}(S_1, C)$ is not a complete slice with respect to S_1 and C, or if X_2 is complete, but S_1 is not complete with respect to P and C. However, both cases lead to a contradiction because both S_1 and X_2 are complete. Moreover, because S_1 and S_2 are complete, then $S_2 \subseteq S_1 \subseteq P$, and thus, S_2 is also a complete slice with respect to P and C. □

While it is mathematically correct to say that the slicing criterion C is common for all program slicers (because C is a reference to a piece of code in P), in practice C is normally provided in a text mode (e.g., $\langle 5, v\rangle$ meaning line 5, variable v), so it is not a reference anymore. Therefore, if a line before C (e.g., *line* 2) is sliced off from P by the first program slicer obtaining S, then C needs to be updated (e.g., to $\langle 4, v\rangle$) so the subsequent program slicers can locate the slicing criterion in S. Figure 4 shows how the slicing criterion is updated. The process consists of four steps: first, an *AST* of the code and of its slice are obtained; second, a mapping ([22, 23]) over

```
   Original Program                    Minimal Slice for (10, B)

1  main (X, Y) ->                      main (X, _) ->
2      A = 1, B = A, C = B,                A = 1, B = A,
3      Z = foo (X, {Y, B, C}),             _ = foo (X, {sliced, B, sliced}).
4      Z.
5
6  foo (X, {Y, B, C}) ->               foo (X, {_, B, _}) ->
7      case X of                           case X of
8          123456789 -> Z = X/Y,
9                       Z + C;
10         2         -> B;                 2 -> B
11         _         -> X/Y
12     end.                                end.

          (a)                                 (b)
```

Figure 2: An Erlang program and its minimal slice for a slicing criterion.

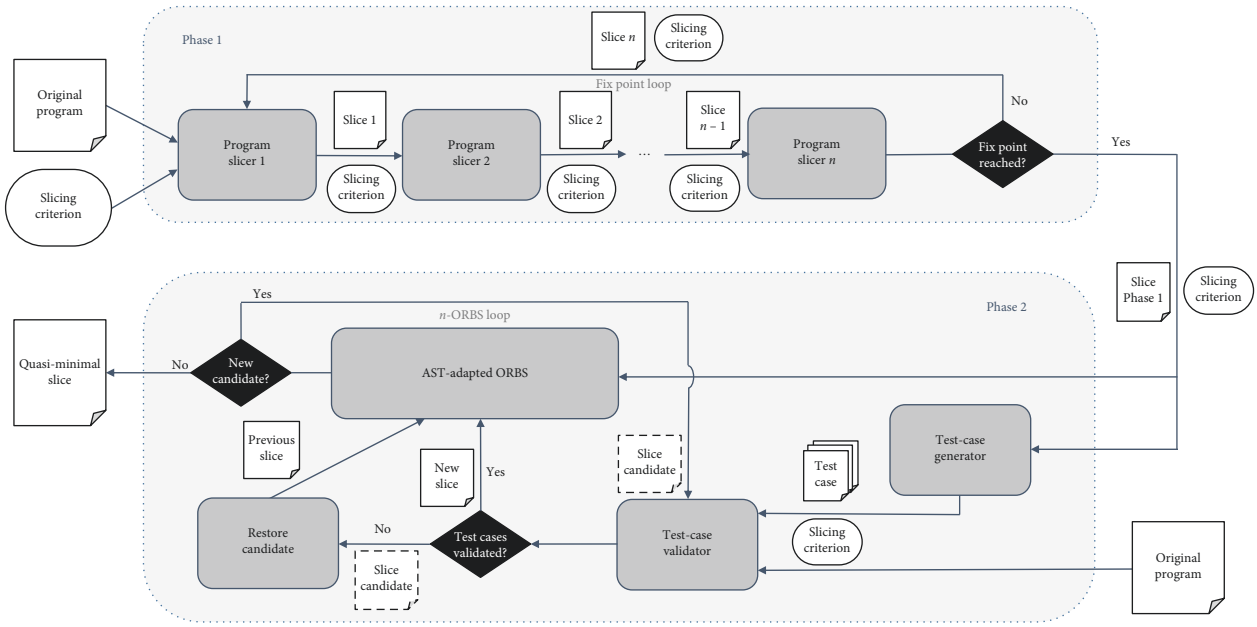

Figure 3: A method to produce quasi-minimal slices.

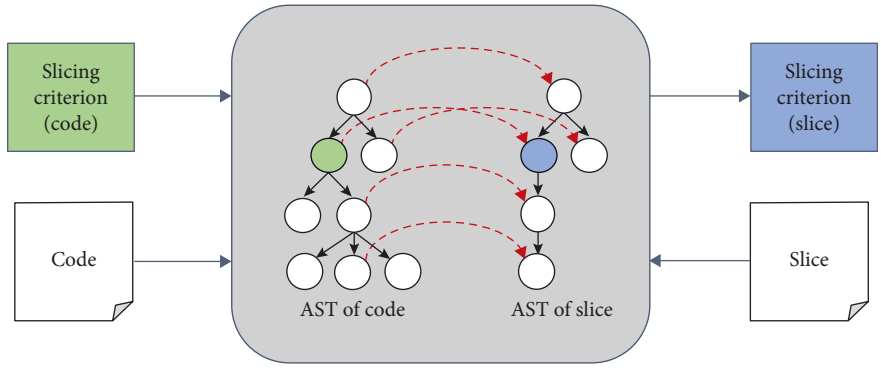

Figure 4: Slicing criterion mapper.

both *AST*s is calculated (dashed lines in the figure); third, the node that represents the slicing criterion (Definitions 8 and 9) is located within the *AST* of the code; finally, the mapping is used to find the node in the *AST* of the slice.

4.2. Phase 2: Increasing Precision via an AST-Adapted ORBS Algorithm. Phase 2 comprises three main modules: ORBS, test-case generation, and test-case validation. We explain the modules hereafter. Before delving into the details, it is worth

to remark that Phase 1 is optional because Phase 2 obtains the same result working alone as when it is combined with Phase 1. However, Phase 1 significantly reduces the number of AST nodes that Phase 2 has to work with, which speeds up the process (e.g., in our implementation of the method, Phase 1 reduces the time of Phase 2 by 64.99%).

4.2.1. ORBS. We have implemented a variant of observation-based slicing (ORBS) [12]. ORBS is a technique that iteratively removes lines from a program and checks whether the observable behaviour is the desired one. This is checked for a particular set of test cases. If the observable behaviour is the desired one, then the line can be effectively removed and the system can try again with a different line until no more lines can be removed. When the system has finished with one line at a time, it can repeat the process removing two lines at each iteration, and so on.

Our variant of ORBS iterates over the AST of the program (instead of iterating over its lines). In particular, it iterates over the AST of "$Slice_{Phase1}$" (Figure 3). Roughly, this variant iteratively tries to remove from the AST each subtree. Each removal attempt of a subtree produces a *"Slice candidate"* (see Figure 3). For each slice candidate, its behaviour is compared with the behaviour of the original program according to Definition 7. If they show the same behaviour, then that part is permanently removed from the AST producing a *"New slice"* (see Figure 3), and ORBS is restarted with this new slice as input. Otherwise, the *"Previous slice"* is restored and used in a new iteration of ORBS. This iterative process is incremental (first, it removes one node at a time, then, two nodes at a time, and so on) and continues until no more nodes can be removed. This ORBS-based technique is described in Algorithm 1, where we use E^* to denote the reflexive and transitive closure of E. Note that the algorithm is parametric with respect to *MN*, which denotes the maximum number of nodes that can be removed to produce a slice candidate (all possible combinations could be checked when $MN = -N-$.) Roughly, the algorithm loops *currMN* from 1 to *MN*. It proceeds by removing every combination of *currMN* nodes and then testing them. For example, first take out each one node (and its subtree) and run tests to check whether the sequences of values at the slicing criterion are preserved. Second, take out combinations of two nodes (and their subtrees), test those, and so on. Always building on the previous result.

For this, the algorithm uses function $seq(P, C, t)$, which executes program P with the test case t and records the sequences of values computed at the slicing criterion C. The recursive function (*ORBSAST*) iterates top-down over the *AST* removing subtrees and checking whether the sequences of values computed with *seq* for the original program are a prefix of the sequences of values computed for the new program with the subtrees removed. This is done with a battery of tests (also called inputs in our context). Function *ORBSAST* is called until a fix-point is reached (**repeat-until** loop), for each number of removed nodes from 1 to *MN* (**for** loop).

This adaptation of *ORBS* to *AST*s works top-down. This is more efficient because it works in a concretization fashion, trying to remove first entire functions, clauses, and data structures before trying with their components. If a bottom-up traversal was used instead, whenever a function could be removed, each of its statements would be removed beforehand. This is probably not a problem in other contexts, but in our context, each time a subtree (e.g., a statement) is removed from the AST, all generated test cases are run to validate that removal. Clearly, these validations are a waste of time in case the whole function is going to be removed.

The only functions that must be provided by the user in Algorithm 1 are *seq* and *generateTestCases*, which is described in the next subsection.

It is important to remark that our algorithm is a generalization of ORBS in two ways. First, because it can slice any expression and not only lines of code. If we consider that the nodes removed can only be those subtrees that correspond to lines in the code, then our algorithm is equivalent to ORBS. However, there is a second generalization. ORBS uses a window of size δ that represent the lines that can be removed all together. Therefore, ORBS can delete various lines at a time, but it imposes the restriction that all of them must be together (inside the window). This means that ORBS cannot produce the minimal slice of Example 1 because lines 4 and 6 must be deleted together without deleting line 5. Our approach allows for deleting different (not necessarily adjacent) subtrees of the AST, thus solving this problem and producing the minimal slice in Example 1.

4.2.2. Test-Case Generation and Validation. The second module used in this phase is in charge of the test-case generation, which is implemented by function *generateTestCases* in Algorithm 1. The goal is to generate test cases that execute different paths of the slice and that evaluate the slicing criterion. Every *"Slice candidate"* produced by ORBS is tested by comparing its behaviour with the one of the original program. If they show the same behaviour, then the missing code in the slice candidate is definitely removed. Otherwise, it is restored. Clearly, the quality of this phase depends on the generated test cases. An important remark is that our architecture takes advantage of Phase 1 not only to produce the refined slice "$Slice_{Phase1}$" but also to improve the generation of test cases. In particular, we can observe in Figure 3 that module *"Test-Case Generator"* inputs "$Slice_{Phase1}$" (instead of the "*Original program*"). Generating the test cases from "$Slice_{Phase1}$" produces better test cases because this avoids generating test cases that explore the removed code in the slice (and,thus that cannot affect the slicing criterion). Observe, however, that "$Slice_{Phase1}$" is not used as input for the module *"Test-Case Validator"* because the output of *seq* for this slice and for the *"Original program"* differs according to property 4 in Definition 4. This is explained in Example 3.

Example 3. Consider the following sequences of values produced in a slicing criterion *SC* when executing a concrete input *I* over the original program (*Original*), the output slice

Input: A program P, an executable program slice S of P, a slicing criterion C for S, and the maximum number of nodes MN to be removed at a time.
Output: A quasi-minimal slice of P.

```
tests = generateTestCases(S, C)
testsSeq = {(t, seq(P, C, t)) | t ∈ tests}
A = getAST(S)
for (currMN ∈ 1 ... MN)
  repeat
    A' = A
    A = ORBSAST(A', 1, currMN, ∅)
  until A = A'
end for
return getProgram(A)
function ORBS_AST(A, currNode, currMN, treatedNodes)
  (N, E) = A
  remNodes = {n ∈ N | ∄n' ∈ treatedNodes . (n', n) ∈ E*}
  while (remNodes ≠ ∅)
    node = n ∈ remNodes | ∄n' ∈ remNodes . (n', n) ∈ E
    remNodes = remNodes\{node}
    N' = N\{n ∈ N | (node, n) ∈ E*}
    E'_s = {(n, n') ∈ E | n, n' ∈ N'}
    A' = (N', E'_s)
    if (currNode < currMN)
      A'' = ORBS_AST(A', currNode + 1, currMN,
                     treatedNodes ∪ {node})
      if (A' ≠ A'')
        return A''
      end if
    else
      if (∀(t, seq_P) ∈ testsSeq.
            seq_P is a prefix of seq(getProgram(A'), C, t))
        return A'
      end if
    end if
  end while
  return A
end function
```

Algorithm 1: ORBS-based *AST* pruning algorithm

of phase 1 ($Slice_{Phase1}$), and a slice candidate (*SliceCandidate*):

$$\begin{aligned} &\text{(a) } seq(Original, I, SC) = [1, 2, 3], \\ &\text{(b) } seq(Slice_{Phase1}, I, SC) = [1, 2, 3, 5], \\ &\text{(c) } seq(SliceCandidate, I, SC) = [1, 2, 3, 7]. \end{aligned} \tag{1}$$

In this scenario, if we validate (i.e., decide whether it is a valid slice) *SliceCandidate* with respect to *Original*, then, according to property 4 of Definition 4, *SliceCandidate* is an executable program slice of *Original* ((a) is a prefix of (c)). Nevertheless, if we validate *SliceCandidate* with respect to $Slice_{Phase1}$, then the validation fails because (b) is not a prefix of (c). This happens because a slice can produce more values than the original program in the slicing criterion. Therefore, module *Test-Case Validator* inputs *Original* to prevent these kinds of false negatives.

Figure 3 summarizes the described phases. In the figure, the phases are enclosed inside light grey boxes; the slicers and the other processes are represented with dark grey boxes; the slices and the test cases are represented with white files; the slice candidates (not validated yet or not valid) are represented with dashed-border white files; and decision points are represented with dark rhombuses. The intermediate and output slices of the first phase must be static executable program slices of the original program (Definition 4), whereas the intermediate and output slices of the second phase are dynamic executable program slices (see Definition 5).

Note that this is a general scheme that can be adapted to any language. For this, we only need to instantiate some of the dark grey components: the program slicers, the test-case validator, and the test-case generator (the ORBS technique is already paradigm-independent and works for any language).

5. Implementation of the Method for Erlang

We describe in this section how we have instantiated the method for Erlang. The method follows the schema shown in

Figure 3, where we use two program slicers in Phase 1 called *Slicerl* [15] and *e-Knife*; *CutEr* [24] as a test-case generator; *SecEr* [25] as a test-case validator; and *Cover* [26] as a coverage meter to decide when to stop generating test cases.

5.1. Phase 1: Slicerl and e-Knife. In our setting, we used two slicers: *Slicerl* [15] and *e-Knife*. We selected Slicerl for four reasons: First, because it is based on a data structure called Erlang Dependence Graph (EDG) whose granularity level is minimal (i.e., tokens). This allows for removing expressions even inside a line of code. Second, because it is open source, and thus, we have been able to access its internal behaviour and analyses, extend it, and use it in our implementation. Third, because it implements some novel optimization techniques that make it very precise. And fourth, because it is interprocedural. Other slicers such as the Wrangler's slicer [27] were quickly discarded because they are only intraprocedural, and thus, they cannot handle with precision any of the benchmarks in the suite (note that this does not mean that the suite is useless for intraprocedural slicers. It just means that intraprocedural slicers are less useful to construct the suite.)

The second slicer is called e-Knife. It is a static slicer for Erlang on which we have been working for the last few years. e-Knife is also based on the EDG and thus, it has the same granularity level as Slicerl tokens (every token is represented in the EDG with a different node that is susceptible of being sliced off). Moreover, e-Knife incorporates a new technique to precisely slice composite data structures, which complement the static analyses made by Slicerl.

Example 4. Given the program on the left in Figure 5, with the slicing criterion $\langle 3, X\rangle$, Wrangler and Slicerl produce the slice in the middle, whereas e-Knife produces the slice on the right.

Note that, even though X depends on A and A depends on 2, X does not depend on 2. Only e-Knife is able to detect intransitive data dependencies.

5.2. Phase 2: CutEr, Cover, and SecEr. In this section, we explain how we have instantiated for Erlang the test-case generation and validation tasks needed for ORBS.

5.2.1. Test-Case Generation. We ensure high quality test cases using concolic testing. We did two sequential steps to ensure a 100% branch and statement coverage:

(i) *Concolic test-case generation.* This technique analyses the branching conditions in the source code and generates constraints that the input must satisfy to visit all branches. Then, a constraint solver is used to produce the test cases. We used a concolic testing tool for Erlang called CutEr [24]. The following example shows that white-box testing can generate test cases that execute very unusual branches.

Original Program	Wrangler's Slice	e-Knife's Slice
1 *main () ->*	1 *main () ->*	1 *main () ->*
2 *A = {1,2},*	2 *A = {1,2},*	2 *A = {1, sliced},*
3 *{X,Y} = A.*	3 *{X,_} = A.*	3 *{X,_} = A.*

Figure 5

Example 5. Consider again the program in Figure 2(a). The case branch in line 8 will be hardly executed with random test-case generation. 100% branch coverage can only be achieved if a test case exists with $X = 123456789$. However, this does not guarantee the evaluation of all expressions in the branch. 100% statement coverage in the first branch can only be achieved if a test case exists with $X = 123456789$ and $Y<>0$.

(ii) *Semirandom test-case generation.* We complemented our white-box testing with black-box testing. We implemented random generators for all possible data types in Erlang.

The maximum number of test cases to be generated is a parameter of our method. This number depends on the concrete code to be processed. The number also has a direct impact on the run time and on the precision of the final slices produced. In the default configuration, our implementation generates test cases until all the codes are tested (i.e., 100% statement and branch coverage). For this, it generates 10 test cases at a time, accumulating the test cases produced and measuring the coverage at each step with a tool called Cover [26]. Cover is a coverage analysis library for Erlang that can determine the coverage achieved when executing a program with several invocations (in our setting, test cases) and that can also identify the uncovered branches. It basically instruments the code so that every line is augmented with a new function call. Therefore, by counting the calls performed during the execution of the test cases we can know exactly what lines were executed and how many times. When Cover reports that 100% statement and branch coverage is reached, the test-case generation finishes. We want to note that these coverages are only metrics but not objectives. 100% coverage does not necessarily imply high slicing precision.

Example 6. Consider again the program in Figure 2(a). A test case with input $X = 1$ and $Y = 1$ does execute all expressions in line 11—100% branch and statement coverage in this line— but it does not trigger the division-by-zero exception. Finding this situation would require to generate more test cases (e.g., $X = 1$ and $Y = 0$).

5.2.2. Test-Case Validation. Our test-case generation obtains inputs that ensure a 100% statement and branch coverage. However, in our case, these inputs must be complemented with very specific outputs to form the test cases: the sequences of values the slicing criterion is evaluated to. In our case, this is done by a tool called *SecEr* [25] (which implements function *seq* in Algorithm 1). Given a

slicing criterion, SecEr instruments the source code in such a way that the execution of the instrumented code obtains as a side effect the sequence of values it is evaluated to.

6. Evaluation of the Method

We identified a collection of slicing problems and challenges and applied our method to obtain 23 benchmarks for Erlang (23 slicing criteria defined over 18 different Erlang programs) that (combined) implement all of the problems. These benchmarks form a suite that contains triples *program–slicing criterion–minimal slice*. The slices produced in our implementation are QM-slices (Definition 7) and they are fine-grained slices because they have been obtained working over AST nodes (Definitions 8 and 9). In this section, we show the behaviour of each component of the method.

6.1. Phase 1: Behaviour of Slicerl and e-Knife. The fix point of Phase 1 was reached in only one iteration (the slice produced by e-Knife (*Slice 2*) could not be further reduced by Slicerl). The first slicer needed 12820 milliseconds to slice all the benchmarks except for nine of them whose syntax is not supported by Slicerl. This produces an average of 916 milliseconds per benchmark. Slicerl was able to remove 619 nodes from *"Original program"* in total (an average reduction of 31.89%). The second slicer needed 48361 milliseconds to slice all the benchmarks (an average of 2103 milliseconds per benchmark) (e-Knife is a multiparadigm slicer implemented in Java. For this reason, it needs extra time to access Erlang). e-Knife further reduced the slices produced by Slicerl by 59 nodes in total (an average extra reduction of 2.48% over the original program). If we also consider those benchmarks that Slicerl cannot handle, then the extra reduction is 14.67%.

6.2. Phase 2: Behaviour of ORBS and CutEr

6.2.1. ORBS. The execution of Algorithm 1 with $Slice_{Phase1}$ and removing one node at a time ($MN = 1$) reduce the original program to 50.02% (as an average). This is an extra reduction of 15.84% over the result of phase 1. Afterwards, Algorithm 1 was executed again but this time removing two nodes instead of one. The slice remained unchanged in all cases (0% reduction). Then, three nodes were removed in each iteration, and 0% reduction was achieved. Finally, four nodes were removed in each iteration for some benchmarks (according to our estimations, the evaluation of the other benchmarks would have taken around 8 months). Again, in all cases, 0% reduction was achieved when four nodes were removed in each iteration. Due to the combinatorial explosion, we did not run any of the benchmarks with five nodes because its run time was estimated in years.

We compare the four iterations performed with ORBS in Table 1. The columns labelled with i nodes, where $i \in \{1, 2, 3, 4\}$, represent each of the iterations of the **for** loop in Algorithm 1 (the first removing 1 node in each iteration, the second removing 2 nodes in each iteration, etc.). In these columns, Iter is the number of different iterations performed by the algorithm (i.e., the number of configurations that were checked, where each configuration is the result of removing i nodes from the AST), Time is the total time used to check the configurations, and % is the percentage of nodes that remain from the original code. Note that the algorithm only removed nodes when trying to remove single nodes (1 node).

This whole exhaustive process (with $MN = 4$) took nine days, thirteen hours, and fifty-one minutes. However, the ORBS loop with 2, 3, and 4 nodes did not produce any reduction (and consumed most of the time). Therefore, unless one is specially interested in producing minimal slices (as we are), it is a good design decision to configure ORBS to only remove one node at a time. This nearly always produces exactly the same results, but the time is significantly reduced. With this configuration ($MN = 1$), the whole suite of benchmarks was sliced in 14 minutes and 25 seconds, producing the same results.

6.2.2. CutEr. The coverage achieved by the test cases generated with CutEr for each benchmark is listed in column CutEr of Table 2. In 14 out of 23 benchmarks, CutEr produced a 100% branch coverage. In 4 out of 23 benchmarks, CutEr produced a branch coverage $< 100\%$. In 5 benchmarks (*b16_s58C*, *b12_s40BS*, *b12_s92A*, *b15_s65Shown*, and *b18_s50J*), CutEr returned an error or was unable to generate any test case.

In all those benchmarks where CutEr did not produce a 100% branch and statement coverage, a second phase of semirandom test-case generation was activated to reach 100%. Column Random of Table 2 shows this second phase where a 100% statement and branch coverage was achieved in only 0.12 seconds on average.

6.3. Empirical Evaluation. Prior to the design and application of our method, we first produced the slices of the benchmarks with Slicerl and with e-Knife, separately. This enables evaluating how precise QM-slices (obtained with our method) are compared to standard slices (obtained with two program slicers).

6.3.1. Executable Program Slices. We sliced all the benchmarks with two Erlang program slicers (*Slicerl* and *e-Knife*) that produced an interesting result: the empirical evaluation of (and a comparison between) each slicer. Slicerl could not handle nine of the benchmarks (it crashed due to unhandled syntax constructs). If we omit these benchmarks, then their precision was similar. As an average, Slicerl reduced the original programs ($\overline{X} = 31.90\%, \sigma = 21.29\%$), while *e-Knife* reduced them ($\overline{X} = 33.03\%, \sigma = 25.18\%$). However, because the analyses performed by both slicers are different, Slicerl was better twice and *e-Knife* was better thirteen times. This clearly justifies the combination of program slicers in the first phase of our method. We also compared the following three slices for all benchmarks:

Table 1: Comparison of the different iterations of ORBS.

Benchmark	1 node			2 nodes			3 nodes			4 nodes		
	Iter	Time (s)	%	Iter	Time (s)	%	Iter	Time	%	Iter	Time (s)	%
b1_s56Year	10430	441.48	28.29	23155	649.08	0	692795	269172.36	0	—	—	—
b2_s38C	253	10.76	26.56	264	6.08	0	1072	23.00	0	2714	53.34	0
b2_s40D	82	3.51	29.69	467	10.49	0	3460	73.66	0	16110	315.32	0
b3_s28C	133	5.50	45.28	136	3.69	0	395	8.76	0	621	15.35	0
b4_s32Abb	193	8.36	72.41	1453	36.41	0	21319	488.77	0	211141	4437.24	0
b5_s30C	42	2.21	94.44	758	18.64	0	7909	165.45	0	54567	1134.96	0
b6_s35C	222	9.07	28.57	414	10.00	0	2923	61.19	0	13063	257.84	0
b6_s36D	126	5.33	20.30	206	4.82	0	836	17.90	0	2039	39.64	0
b7_s27C	18	0.87	78.57	105	3.38	0	234	5.33	0	285	7.54	0
b8_s29Deposits	244	23.14	72.50	1165	51.24	0	15468	540.83	0	140135	4134.79	0
b9_s59A	112	4.96	88.14	799	18.83	0	8039	177.06	0	51658	1353.71	0
b10_s34DB	253	43.48	76.43	4383	225.84	0	123884	4651.84	0	2477094	81536.00	0
b11_s28C	28	1.18	80.00	293	7.00	0	1588	44.10	0	5509	133.16	0
b12_s40BS	138	12.56	25.77	4872	257.64	0	145830	5510.94	0	3085914	102958.79	0
b12_s92A	83	7.17	20.70	3168	166.99	0	74649	2625.25	0	1227042	37374.00	0
b13_s38NewI	41	0.65	69.70	700	29.19	0	6649	208.36	0	39929	1112.95	0
b14_s44V	214	6.78	26.32	983	22.33	0	11418	224.08	0	84958	1689.93	0
b14_s45W	137	156.76	23.92	808	465.78	0	8169	4563.61	0	54519	30204.37	0
b14_s46Z	127	5.35	18.18	357	8.16	0	2402	47.41	0	10629	215.53	0
b15_s65Shown	657	34.97	81.33	13208	384.85	0	661963	16267.98	0	23716885	560087.23	0
b16_s58C	25	1.37	27.78	241	13.07	0	1195	75.92	0	3465	399.11	0
b17_s54X	408	38.39	68.35	848	69.63	0	9230	3263.62	0	—	—	—
b18_s50J	341	41.25	48.42	588	50.01	0	4965	3761.92	0	—	—	—
Average	622.04	37.61	50.08	2581.35	109.27	0	78538.78	13564.32	0	1356446.83	35976.58	0
Median	137	7.17	45.28	758	22.33	0	7909	208.36	0	45793.50	1123.95	0
Total	14307	865.09	1151.67	59371	2513.15	0	1806392	311979.34	0	31198277	798490.53	0

$$S_1 = slice_{Slicerl}\left(slice_{e-Knife}(B,C),C\right),$$
$$S_2 = slice_{e-Knife}\left(slice_{Slicerl}(B,C),C\right), \quad (2)$$
$$S_3 = slice_{Slicerl}(B,C) \cap slice_{e-Knife}(B,C),$$

where B is a benchmark and C is a slicing criterion (note that, theoretically, unions and intersections of slices are not necessarily slices [17], but in practice (e.g., with all our benchmarks), they usually are). We discovered that, for all benchmarks, $S_1 = S_2 \subseteq S_3$. Hence, (i) the order in which the slicers were executed was not relevant and (ii) it is better composing slicers sequentially (i.e., slicing slices) than composing them in parallel and get the intersection. The reason is that one slicer can take advantage of the parts removed by the other slicer. This justifies the need for a fix-point loop in Phase 1 of the method.

6.3.2. Quasi-Minimal Slices. Table 2 summarizes the empirical evaluation of our particular implementation of the proposed method. Concretely, it compares the size of the successive refinements of all the slices, and the time needed by all processes of the two phases. Each row represents a different benchmark. For each benchmark, column Nodes represents its number of AST nodes, which corresponds to the size of the programs/slices. In the case of the slices, we also include the percentage of nodes that remain in the slice with respect to the original program. Column Time shows the time expended in each phase measured in seconds (s). Phase 2 is divided into two different processes: test-case generation and ORBS limited to only one iteration (see Section 4.2 for a justification of this decision). Finally, column Iterations shows the number of configurations checked by ORBS (that is, the number of different nodes removed to produce slice candidates).

It is important to compare the data of the different rows taking into account that the columns provide complementary information. For instance, if we compare the reduction % achieved by *Slicerl* for benchmarks b5_s30C and b14_s44V, one can think that the slice produced for b14_s44V is much better (it was reduced to 37.8%, while b5_s30C was only reduced to 94.44%). However, if we observe the % in the ORBS column, we can see that the conclusion could be the opposite: *Slicerl* produced a minimal slice for b5_s30C, while the slice produced for b14_s44V was not minimal.

6.3.3. Lessons Learnt. Our implementation of the proposed method and its empirical evaluation has answered several research questions in the process:

(1) *Is Phase 1 really needed?* The final slice produced in Phase 2 is the same with independence of whether Phase 1 is used or not. However, the use of Phase 1 reduced the time of Phase 2 by 64.99%.

(2) *Run time: How long does each process last?* The whole suite was sliced in 1054 s. (Phase 1: 62 s, ORBS: 865 s, and test case generation: 127 s). This provides an idea of the relative costs of the phases.

Table 2: Empirical evaluation of the method instantiated for Erlang.

	Original	Phase 1				Phase 2					Total
		Slicerl		*e-Knife*		Test-case generation		*ORBS* (one node at a time)			
	Nodes	Time (s)	Nodes (%)	Time (s)	Nodes (%)	CutEr	Random	Iterations	Time (s)	Nodes (%)	Time (s)
b1_s56Year	820	—	—	3.21	649 (79.15)	100% (4.93 s)	—	10430	441.48	181 (28.54)	449.62
b2_s38C	128	0.87	95 (74.22)	2.22	95 (74.22)	100% (16.03 s)	—	253	10.76	34 (26.56)	29.88
b2_s40D	128	0.88	98 (76.56)	2.32	98 (76.56)	100% (16.01 s)	—	82	3.51	38 (29.69)	22.72
b3_s28C	53	0.82	35 (66.04)	2.12	35 (66.04)	100% (3.15 s)	—	133	5.50	24 (45.28)	11.59
b4_s32Abb	87	—	—	2.19	74 (85.06)	100% (10.45 s)	—	193	8.36	63 (72.41)	21.00
b5_s30C	54	0.99	51 (94.44)	2.17	51 (94.44)	78% (5.03 s)	100% (0.02 s)	42	2.21	51 (94.44)	10.42
b6_s35C	133	1.02	62 (46.62)	2.18	57 (42.86)	100% (10.87 s)	—	222	9.07	38 (28.57)	23.14
b6_s36D	133	1.01	49 (36.84)	2.16	49 (36.84)	100% (3.29 s)	—	126	5.33	27 (20.30)	11.80
b7_s27C	28	0.95	22 (78.57)	2.12	22 (78.57)	100% (2.61 s)	—	18	0.87	22 (78.57)	6.55
b8_s29Deposits	80	0.95	78 (97.50)	2.23	76 (95.00)	86% (12.69 s)	100% (0.03 s)	244	23.14	58 (72.50)	39.03
b9_s59A	59	0.81	57 (96.61)	2.18	54 (91.53)	100% (14.07 s)	—	112	4.96	52 (88.14)	22.02
b10_s34DB	140	—	—	2.25	111 (79.29)	82% (4.96 s)	100% (0.02 s)	253	43.48	107 (76.43)	50.71
b11_s28C	40	0.83	32 (80.00)	2.15	32 (80.00)	100% (3.96 s)	—	28	1.18	32 (80.00)	8.12
b12_s40BS	454	—	—	2.52	118 (25.99)	—	100% (0.03 s)	138	12.56	117 (25.77)	15.11
b12_s92A	454	—	—	2.18	94 (20.70)	—	100% (0.03 s)	83	7.17	94 (20.70)	9.38
b13_s38NewI	66	0.94	46 (69.70)	2.15	46 (69.70)	100% (1.65 s)	—	41	0.65	46 (69.70)	5.38
b14_s44V	209	0.93	79 (37.80)	2.18	75 (35.89)	100% (1.69 s)	—	214	6.78	55 (26.32)	11.58
b14_s45W	209	0.96	89 (42.58)	2.20	64 (30.62)	67% (2.99 s)	100% (0.83 s)	137	156.76	49 (23.92)	163.74
b14_s46Z	209	0.86	117 (55.98)	2.23	97 (46.41)	100% (4.54 s)	—	127	5.35	38 (18.18)	12.98
b15_s65Shown	225	—	—	2.40	195 (86.67)	—	100% (0.04 s)	657	34.97	183 (81.33)	37.41
b16_s58C	108	—	—	1.67	30 (27.78)	—	100% (0.04 s)	25	1.37	30 (27.78)	3.09
b17_s54X	79	—	—	1.01	76 (96.20)	100% (7.38 s)	—	408	38.39	54 (68.35)	46.79
b18_s50J	95	—	—	1.02	92 (96.84)	—	100% (0.04 s)	341	41.25	46 (48.42)	42.32
Average	173.52	0.56	146.61 (80.59)	2.13	99.57 (65.93)	95.17% (5.49 s)	100% (0.12 s)	622.04	37.61	64.91 (50.08)	45.84
Median	128	0.93	87 (94.44)	2.18	75 (76.56)	100% (4.95 s.)	100% (0.03 s.)	137	7.17	12 (50.00)	45.28
Total	3991	12.82	3372 (80.59)	49.08	2290 (65.93)	(126.30)	(1.08 s)	14307	865.09	1493 (50.08)	1054.36

(3) *Accuracy: How accurate is each phase (on average)?* Phase 1 reduced the original program 34.07%, and Phase 2 further reduced it 15.85% (producing the minimal slice).

(4) *Concolic vs. random test cases: Is concolic testing enough?* No. CutEr was able to produce the desired coverage 60.87% of the times. In the other 39.13%, random test-case generation was needed.

(5) *Slicerl vs. e-Knife: Which is better (on average)?* When they were run independently, e-Knife was better in 13/23 benchmarks. Table 3 shows the comparison of both slicers.

(6) *Sequential vs. parallel composition (intersection) of slicers: Which is better?* Sequential composition of slicers provides the best results.

7. A Suite of Minimal Slices

Following the method presented, we have generated a suite of minimal slices for Erlang. In Erlang, this suite is especially useful because it presents special challenges for program slicing (higher order, anonymous functions, pattern matching, etc.) and, moreover, in this language, no studies evaluating current program slicers existed yet.

7.1. Selection of Benchmarks. The suite of benchmarks has been designed to contain small to medium programs that contain well-known program slicing challenging problems described in the literature (e.g., dead code, unreachable clauses [21], pattern matching [15], and collapse and expansion of composite data structures [28]). For instance, the suite includes classical slicing programs used in different papers such as word count, the SCAM mug, the Montréal boat example, and the Horwitz et al. interprocedural slice [29]. The objective is to challenge program slicers to check how many of these programs are they able to slice. In order to test different syntax constructs in Erlang that are also challenging for program slicing (e.g., list comprehensions, block structures, chars, and remote function calls), various benchmarks have been taken from the *github* repository and the *rosetta code* programming chrestomathy website (http://rosettacode.org/). For each benchmark, we defined different slicing criteria so that their slices can be used to test slicers that work at the function, clause, line, or expression level. The suite of benchmarks has been designed to contain small to medium programs that

(i) Require interprocedural techniques. Interprocedural slicing is a challenge in functional languages. For instance, the program slicer of Wrangler [20], one of the most advanced Erlang refactoring tools, is still intraprocedural.

(ii) Can be sliced by slicers of different performance. The main goal of the suite is not performance but precision. Therefore, we prefer small to medium programs for which we can systematically produce minimal slices rather than large programs for which reasoning about minimality is impossible due to its prohibitive cost.

(iii) Contain different slicing problems. In fact, each benchmark defines concisely one specific slicing challenge.

This suite can be used to evaluate and compare program slicers, but it is also particularly useful to develop slicers. To help in this last task, we have implemented a tool that inputs a program slicer and it slices all the benchmarks in the suite with this program slicer. Then, the slices the program slicer obtains are compared with the minimal slices in the suite to calculate the accuracy in terms of preserved AST nodes (i.e., using the minimum granularity). Finally, a report indicating the recall, precision, and F1 is provided to the user as well as the variation of these metrics with respect to the best results the program slicer has achieved so far. The suite and the tool are publicly available at http://personales.upv.es/josilga/slicing/bencher/.

7.2. Structure of the Suite. All benchmarks are labelled so that their purposes and properties can be identified by just looking at their labels. The labels classify the benchmarks depending on the slicing challenges they include and on the syntax constructs they use.

Example 7. All benchmarks are identified with a code. For instance, benchmark b15_s65Shown refers to program 15 with slicing criterion $\langle 65, Shown \rangle$. The code of program 15 was originally extracted from *rosettacode*. Then, the code was augmented and redesigned to include challenging problems for slicing. Finally, this benchmark has been labelled with IP, LC, AF, Rem. Their meanings are as follows:

IP: the benchmark requires interprocedural slicing

LC: the benchmark uses list comprehensions

AF: the benchmark defines and uses anonymous functions

Rem: the benchmark contains remote procedure calls to external functions (nonavailable code)

All the information about the meaning of the labels and about the classification of benchmarks can be found on the public website of the suite.

7.3. Minimality. Our method/slicer produces quasi-minimal slices. Ensuring minimality is undecidable because not all possible test cases can be executed (they are potentially infinite). However, our method palliates this problem with a test-case generation phase that ensures 100% branch and statement coverage combining white-box and black-box testing. Thanks to this phase, the quasi-minimal slices produced are actually minimal in many cases. In particular, we have manually proved that all 23 quasi-minimal slices generated with our tool (with MN = 1 and generating random test cases until 100% statement and branch coverage is achieved) are in fact minimal slices.

Table 3: Empirical evaluation and comparison of Slicerl and e-Knife.

	Original	*Slicerl*		*e-Knife*	
	Nodes	Time (s)	Nodes (%)	Time (s)	Nodes (%)
b1_s56Year	820	—	—	3.21	647 (78.90)
b2_s38C	128	0.87	95 (74.22)	2.07	95 (74.22)
b2_s40D	128	0.88	98 (76.56)	2.08	98 (76.56)
b3_s28C	53	0.82	35 (66.04)	2.08	35 (66.04)
b4_s32Abb	87	—	—	2.19	74 (85.06)
b5_s30C	54	0.99	51 (94.44)	2.09	51 (94.44)
b6_s35C	133	1.02	62 (46.62)	2.10	72 (54.14)
b6_s36D	133	1.01	49 (36.84)	2.09	49 (36.84)
b7_s27C	28	0.95	22 (78.57)	2.16	22 (78.57)
b8_s29Deposits	80	0.95	78 (97.50)	2.08	76 (95.00)
b9_s59A	59	0.81	57 (96.61)	2.06	54 (91.53)
b10_s34DB	142	—	—	2.78	113 (79.58)
b11_s28C	40	0.83	32 (80.00)	2.09	32 (80.00)
b12_s40BS	454	—	—	2.52	118 (25.99)
b12_s92A	454	—	—	2.18	94 (20.70)
b13_s38NewI	66	0.94	46 (69.70)	2.17	46 (69.70)
b14_s44V	209	0.93	79 (37.80)	2.27	99 (47.37)
b14_s45W	209	0.96	89 (42.58)	2.28	64 (30.62)
b14_s46Z	209	0.86	117 (55.98)	2.28	97 (46.41)
b15_s65Shown	225	—	—	2.39	195 (86.67)
b16_s58C	108	—	—	1.67	30 (27.78)
b17_s54X	79	—	—	1.01	76 (96.20)
b18_s50J	95	—	—	1.02	92 (96.84)
Average	173.52	0.92	65 (68.10)	2.12	101.26 (66.97)
Average total	173.52	0.56	146.61 (80.59)	2.12	101.26 (66.97)

Concretely, we have proven minimality for each single pair ⟨benchmark, slicing criterion⟩ in the suite, proving that each single node of the sliced AST is actually needed and that all required nodes are part of the slice. Each benchmark of the suite is thus accompanied with a proof of minimality.

8. Related Work

One approach similar to ours is dynamic program dicing, proposed by Chen and Cheung [30] as an alternative to static program dicing, which was originally proposed by Lyle and Weiser [31]. This approach obtains a program slice formed by the statements contained in the traces of a set of failed executions and not in the traces of a set of correct executions. In most cases, the remaining statements would contain the source of the error. Nevertheless, this approach presents two differences with respect to our approach. 1) It is incomplete. The slices produced may not contain the errors that produced the discrepancies. (2) The slices produced may not be executable, and thus, they cannot be used to check the discrepancies.

In our approach, we use a technique that can be considered a variant of ORBS [12]. ORBS is a language-independent technique, and thus it removes lines without parsing them. Hence, if two statements are placed in the same line, they are removed together. Of course, this can also produce compilation errors if a part of one syntax construct is removed. Instead of removing lines of the code, we use a mechanism to remove expressions or replace them by a fresh constant sliced; thus, the obtained precision is higher and, moreover, this enables us to remove expressions with independence of how they were coded. Our proposed ORBS algorithm is similar to the one proposed in [32] that removes nodes one by one. Specifically, the algorithm proposed in this work is a generalization of [32], because we also allow to iteratively remove N nodes (instead of one) by computing all possible combinations with an efficient top-down pruning algorithm.

Our technique is also similar to Delta Debugging (DD) [11, 33]. DD was originally defined for debugging, but it can also be used to compute slices. The way in which DD and our technique compute slices differ. DD relies on the use of a trace, which is cut in the middle first, in a quarter next, and so on. This process is too expensive compared to our approach (and also compared to ORBS). Moreover, DD can produce slices that are not correct, in the sense that their behaviour differs from the one of the original program. Clearly, this is useless for our purposes because we need to ensure that the slices of the suite are correct.

Another related approach is Critical Slicing [5]. The idea behind Critical Slicing is the same as ORBS: they both remove lines and check whether the slice produced by removing each line preserves the original behaviour. The difference is that Critical Slicing removes lines one at a time, while ORBS removes them incrementally. As a consequence, (i) contrarily to ORBS, Critical Slicing needs a fixed number of compilations (one per line), and (ii) critical slices can be incorrect because two lines individually removed without changing the behaviour at the slicing criterion may produce

a program with a different behaviour when they are removed together. Hence, as DD, Critical Slicing also produces incorrect slices.

Comparing and evaluating the performance of program slicers and slicing-based techniques has been traditionally of wide interest, not only because this enables developers to select the best slicer or technique for their purposes but also because it provides information about how precise the slicer is. For this reason, many surveys and works exist (e.g., [18, 34, 35]) that evaluate and compare the size of the slices produced by different techniques. Unfortunately, due to the lack of a standard suite of benchmarks, in most cases, the benchmarks are implemented from scratch to make the experiments [34, 35], they are taken from different papers and projects [12], or they belong to suites of programs not specific for slicing [18]. Moreover, often, the benchmarks that are used in the experiments are not publicly available or accessible (e.g., in [18, 34, 35]), which makes them impossible to replicate and/or validate the study. Furthermore, the unavailability of the benchmarks prevents other researchers and developers from comparing their techniques with the reported results. In consequence, these reports are just a fixed picture of the state of the art, but they are not usable to measure and compare future techniques.

A suite of program slicing benchmarks would solve these problems, but we are not aware of any suite of benchmarks prepared for slicing, i.e., with specific challenging problems for slicing and with solutions (minimal slices) for each benchmark. The construction of this suite is completely novel. Unfortunately, computing the minimal slices of each benchmark is not trivial at all. In fact, it is undecidable in the general case, so we had to manually prove minimality. The techniques used in this system are very related to other existing techniques and methods. In particular, we use semirandom test-case generation similar to the one implemented by SmallCheck [36]. We also prevent duplicated test cases but, contrarily to SmallCheck, our test-case generation is not based on properties.

9. Conclusions

This work presents a method to produce a new type of slice that we call quasi-minimal slice. This method has been used to obtain a suite of minimal slices. In all our benchmarks, we have proved that the quasi-minimal slices obtained with the method are indeed minimal slices.

The method includes the use of several tools, including program slicers, white-box and black-box test-case generators, and coverage tools that are combined in such a way that they minimize their global computation effort and maximize their performance.

In the process of designing the method, we had to define new algorithms to further reduce the size of our slices. In particular, we have implemented a new interprocedural slicer for Erlang, e-Knife, and we have adapted the ORBS technique to work with AST nodes instead of lines. Of course, the methodology can be perfectly used with the standard ORBS technique but reimplementing it to use as an AST instead of lines has increased its precision.

We have instantiated the proposed methodology and produced the first program slicing benchmark suite for Erlang. As a result, we have developed a collection of benchmarks with specific and challenging problems for program slicing. Each benchmark in the suite is composed of its slicing criteria, their associated minimal slices (accompanied with a proof), and metainformation to ease its use and classification.

The evaluation of the methodology has produced interesting residual results. In particular, we have empirically evaluated and compared two Erlang slicers, proving that they are complementary and should be combined if reducing the size of the slice is critical. We have also evaluated three combinations of the slicers (two sequential and one in parallel) showing that the sequential combinations produce better results. We have also evaluated the ORBS technique with our suite of benchmarks. This has revealed that removing one node at a time often (always in our experiments) produces the same results as removing 2, 3, and 4 nodes at a time. This justifies skipping these expensive configurations.

It is also interesting to remark that our implementation of the method is fully automatic and can be used itself as a very precise slicer because it takes a program and produces a QM-slice. Moreover, this new slicer is not only precise but also scalable if we parameterize Algorithm 1 with $MN = 1$. According to our experiments (Table 2), this significantly reduces the run time at no cost (precision is not reduced according to our experiments because $MN > 1$ never reduced the size of any slice).

Data Availability

The data used to support the findings of this study are included within the article.

Disclosure

A preliminary version of this paper was presented at the XVI edition of the *Spanish Workshop on Programming Languages* (PROLE 2016).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work has been partially supported by MINECO/AEI/FEDER (EU) under grant TIN2016-76843-C4-1-R and by the Generalitat Valenciana under grant PROMETEO-II/2015/013 (SmartLogic).

References

[1] M. Weiser, "Program slicing," in *Proceedings of 5th international conference on Software engineering (ICSE '81)*, pp. 439–449, IEEE Press, San Diego, CA, USA, March 1981.

[2] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[3] D. W. Binkley and K. B. Gallagher, "Program slicing," *Advances in Computers*, vol. 43, no. 2, pp. 1–50, 1996.

[4] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Computing Surveys*, vol. 44, no. 3, pp. 1–41, 2012.

[5] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 121–134, 1996.

[6] C. Ochoa, J. Silva, and G. Vidal, "Lightweight program specialization via dynamic slicing," in *Proceedings of 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP '05*, pp. 1–7, ACM, Tallinn, Estonia, September 2005.

[7] Á. Hajnal and I. Forgács, "A demand-driven approach to slicing legacy COBOL systems," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 67–82, 2011.

[8] A. Majumdar, S. J. Drape, and C. D. Thomborson, "Slicing obfuscations: design, correctness, and evaluation," in *Proceedings of 2007 ACM Workshop on Digital Rights Management, DRM '07*, pp. 70–81, ACM, Alexandria, VA, USA, October 2007.

[9] V.-P. R.. Indus, "A toolkit to customize and adapt Java programs," http://indus.projects.cis.ksu.edu.

[10] P. Anderson, T. Reps, and T. Teitelbaum, "Design and implementation of a fine-grained software inspection tool," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 721–733, 2003.

[11] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

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

[13] M. Llorens, J. Oliver, J. Silva, S. Tamarit, and G. Vidal, "Dynamic Slicing techniques for petri nets," *Electronic Notes in Theoretical Computer Science*, vol. 223, pp. 153–165, 2006.

[14] J. M. Almendros-Jimenez, J. Silva, and S. Tamarit, "Xquery optimization based on program slicing," in *Proceedings of 20th ACM International Conference on Information and Knowledge Management, CIKM '11*, pp. 1525–1534, ACM, Glasgow, UK, October 2011.

[15] J. Silva, S. Tamarit, and C. Tomás, "System dependence graphs in sequential Erlang," in *Proceedings of 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012), volume 7212 of Lecture Notes in Computer Science (LNCS)*, pp. 486–500, Springer, Tallinn, Estonia, April 2012.

[16] M. Llorens, J. Oliver, J. Silva, and S. Tamarit, "Dynamic slicing of concurrent specification languages," *Parallel Computing*, vol. 53, pp. 1–22, 2016.

[17] A. De Lucia, M. Harman, R. Hierons, and J. Krinke, "Unions of slices are not slices," in *Proceedings of Seventh European Conference on Software Maintenance and Reengineering, CSMR '03*, p. 363, March 2003.

[18] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 2, p. 8, 2007.

[19] D. Binkley, "Precise executable interprocedural slices," *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1-4, pp. 31–45, 1993.

[20] H. Li, S. Thompson, L. László et al., "Refactoring erlang programs," in *Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Germany, November 2006.

[21] K. Sagonas, J. Silva, and S. Tamarit, "Precise explanation of success typing errors," in *Proceedings of ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM '13*, pp. 33–42, ACM, Rome, Italy, January 2013.

[22] K.-C. Tai, "The tree-to-tree correction problem," *Journal of the ACM*, vol. 26, no. 3, pp. 422–433, July 1979.

[23] D. d. C. Reis, P. B. Golgher, A. S. Silva, and A. H. F. Laender, "Automatic web news extraction using tree edit distance," in *Proceedings of 13th International Conference on World Wide Web (WWW'04)*, pp. 502–511, ACM, New York, NY, USA, May 2004.

[24] A. Giantsios, N. Papaspyrou, and K. Sagonas, "Concolic testing for functional languages," in *Proceedings of 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*, pp. 137–148, ACM, Siena, Italy, July 2015.

[25] D. Insa, S. Pérez, J. Silva, and S. Tamarit, "Erlang code evolution control," in *Proceedings of 27th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2017)*, Namur, Belgium, October 2017.

[26] Erlang-cover, 1997, http://www.erlang.org/doc/apps/tools/cover_chapter.html.

[27] H. Li, S. Thompson, G. Orosz, and M. Tóth, "Refactoring with wrangler, updated: data and process refactorings, and integration with eclipse," in *Proceedings of 7th ACM SIGPLAN Workshop on ERLANG, ERLANG '08*, pp. 61–72, ACMd, Victoria, BC, Canada, September 2008.

[28] M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, and T. Kozsik, "Impact analysis of erlang programs using behaviour dependency graphs," in *Proceedings of Third Summer School Conference on Central European Functional Programming School, CEFP'09*, pp. 372–390, Springer-Verlag, Budapest, Hungary, May 2010.

[29] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88*, pp. 35–46, ACM, Atlanta, GA, USA, June 1988.

[30] T. Y. Chen and Y. Y. Cheung, "Dynamic program dicing," in *Proceedings of 1993 Conference on Software Maintenance*, pp. 378–385, Montréal, Canada, September 1993.

[31] W. M. Lyle, "Automatic program bug location by program slicing," in *Proceedings of 2nd International Conference, Computers and Applications*, vol. 2, pp. 877–883, Peking, China, 1987.

[32] D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo, "Tree-oriented vs. line-oriented observation-based slicing," in *Proceedings of 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 21–30, Shanghai, China, September 2017.

[33] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *Proceedings of Fourth International Workshop on Automated Debugging*, Munich, Germany, August 2000.

[34] T. Hoffner, "Evaluation and comparison of program slicing tools," Technical Report, LiTH-IDA-R-95-01 Department of Computer and Information Science, University of Kent, Linkping University, Sweden, Sweden, 1995.

[35] D. Giffhorn and C. Hammer, "An evaluation of slicing algorithms for concurrent programs," in *Proceedings of 7th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'07)*, pp. 17–26, Maison Internationale, Paris, France, September 2007.

[36] C. Runciman, M. Naylor, F. L.. Smallcheck, and L. Smallcheck, "Automatic exhaustive testing for small values," *SIGPLAN Not*, vol. 44, no. 2, pp. 37–48, sep 2008.