Acta Cybernetica **20** (2012) 483–497.

Understanding Program Slices

Ákos Hajnal* and István Forgács†

Abstract

Program slicing is a useful analysis for aiding different software engineering activities. In the past decades, various notions of program slices have been evolved as well as a number of methods to compute them. By now program slicing has numerous applications in software maintenance, program comprehension, reverse engineering, program integration, and software testing. Usability of program slicing for real world programs depends on many factors such as precision, speed, and scalability, which have already been addressed in the literature. However, only a little attention has been brought to the practical demand: when the slices are large or difficult to understand, which often occur in the case of larger programs, how to give an explanation for the user why a particular element has been included in the resulting slice. This paper describes a reasoning method about elements of static program slices.

Keywords: data flow analysis, static program slicing, reasoning

1 Introduction

Program slicing is a source code analysis technique proposed by Mark Weiser [35] capable of automatically identifying the set of program statements, called the *slice*, which may affect the values of the selected variables at a program point of interest, called the *slicing criterion*. Program slicing uses dependence analysis that examines the source code to trace control- and data flow to determine the statements that belong to the slice.

Weiser's original method – motivated to aid debugging activities – has been classified later as a "backward static" program slicing technique. *Backward*, because in constructing the slice, statements affecting the selected statement are traced backwards, and *static*, because the analysis is made without having specified any particular program execution, i.e. all possible program executions are taken into account. Forward static program slicing determines the part of the program that is directly or indirectly affected by the selected statement.

*Computer and Automation Research Institute Hungarian Academy of Sciences, E-mail: ahajnal@sztaki.hu

†4D Soft Ltd., E-mail: forgacs@4dsoft.hu

DOI: 10.14232/actacyb.20.4.2012.2

Since Weiser's method other variants of program slicing have been evolved such as dynamic slicing [30, 2], quasi-static slicing [34], conditioned slicing [8], amorphous slicing [22], hybrid slicing [19], and relevant slicing [20]. In the past decades, numerous applications of program slicing have been proposed in different areas of software engineering, including software maintenance [16, 15, 10, 11], program comprehension [12, 24], reverse engineering [9], program integration [27, 7], and software testing [18, 23, 4, 5, 14, 25, 26].

Program slicing allows the users to focus on the selected aspects of semantics by breaking the whole program into smaller pieces, and when these slices are small they can be more easily maintained. However, lager program slices, but even slices containing only some tens of program instructions can be very difficult to understand. As William Griswold [17] pointed out in his talk: *Making Slicing Practical: The Final Mile*, one of the problems why slicers are not widely used is that it is not enough to dump the results onto the screen without explanation.

Slices computed based on execution traces (dynamic) are typically smaller than the ones that consider all possible program executions (static). Furthermore, as a particular execution history is available during dynamic slicing, the chain of dependences caused a given program statement to be included in the slice can be more easily discovered. This is not the case in static slicing, where neither a particular dependence chain nor an execution trace covering these dependences are presented. Some applications such as program comprehension, re- and reverse engineering rely on static slicing, and it may occur that code under analysis cannot be even compiled and run (legacy systems, program under development).

Static program slicing gives a wider view to the connected parts of the program code, which is essential in program comprehension or at extracting reusable functions from legacy systems – considering all possible program executions. Note that without an automated slicing tool revealing dependences in the program text is very labor-intensive, tedious, and time consuming task. These techniques calculate the set of statements that directly or indirectly affect (or affected by) the slicing criterion. However, beyond claiming that there is dependence between the slicing criterion and the computed slice element, no explanation of the result is provided, which could help in understanding the effects between different parts of the program code by the human users.

For example, in regression testing, one can use static program slicing to determine those parts of the code that are affected by the program modification. It can occur that one or more slice elements fall out of the software component that the change supposed to be influenced, so the user may be curious how the effect has reached that point. By showing a particular chain of dependences from the slicing criterion to the selected slice element the user could be convinced that the influence indeed exists, and either there is an unforeseen, undesired side effect of the modification, or this effect has not been taken into consideration at determining the impact of the change.

The more precise the applied slicing technique the less the resulting slice sizes are. There are no fully precise static slicing methods for real programming languages, so *false positives*, i.e. slice elements identified on dependences that actually

cannot occur during real program executions are unavoidable. One source of such imprecision is due to following non-realizable program paths during the analysis (paths along which procedure calls and returns are incorrectly nested). By applying *context-sensitive* techniques, these false positives can be filtered out. Other sources of imprecision are due to infeasible program paths (no such program input that results in the execution of the traversed conditional branches) and programming language constructs that make impossible to recover the precise flow of data statically (use of pointers, dynamic constructs). The latter two problems are not solvable in general; static slicing techniques typically use a conservative approach to provide safe results (consider all potential but not necessarily "real" effects).

In this case, reasoning about slice elements could help programmers to recognize false positives. In regression testing, for example, an unexpected impact of a program change may be proven to be false, when the presented chain of dependences is infeasible (it can be realized along infeasible paths only), and it is rejected by a human user. This is a manual process, but it can still be less expensive than retesting all the slicer indicated parts of the code.

This paper concerns with the token propagation-based context-sensitive, static program slicing technique [21], and proposes a method to reason about the computed slice elements. Reasoning means showing a specific *dependence chain* – along with control-flow information – from the slicing criterion to the selected slice element.

The rest of the paper is organized as follows. Section 2 provides an overview of the necessary concepts, and summarizes the basic rules of the token propagation-based slicing method. Section 3 describes how a dependence trace for the slice elements can be derived by computing a particular dependence chain. Section 4 discusses the related work. Finally, Section 5 concludes the paper.

2 Background

Computer programs can be represented by directed graphs called *control flow graphs* (CFGs). Control flow graphs are constructed by assigning a directed graph to each procedure (*intra*procedural control flow graph) with unique entry and exit nodes, in which nodes correspond to the statements and predicates in the procedure, and edges represent the possible flow of control. A call statement is represented by two nodes, a *call site* and a *return site*, which are linked to the entry and exit node of the called procedure, respectively (*inter*procedural control flow graph). We refer to the related call site c and return site r using the *callSiteOf*, *returnSiteOf* operators, such that, c=*callSiteOf(r)* and r=*returnSiteOf(c)*.

Variable references and assignments are referred to as *uses* and *definitions* in nodes. A definition is *influenced* by a use in the same node if the assigned value is dependent on the value of the referenced variable. A path containing no definition for a variable v (excluding start and end nodes) is a *definition-clear path* with respect to variable v. The definition of v in node n and the use of the same variable in node m form a *definition-use pair* if there is a definition-clear path with respect

to v from n to m. Node m is said to be (directly) *data dependent* on node n.

A statement S is *control dependent* on predicate P if the outcome of P determines whether S executes. Intuitively it means that statements contained by conditionally executed branches are control dependent on the predicate. Control dependent procedure calls extend the control scope to statements in the called procedure(s). There are different definitions and computations of control dependence, the particular notion is however orthogonal to how to compute the slice. We assume that (intraprocedural) control dependences are available in the program graph, represented by *control edges*. We treat interprocedural control dependences indirectly, by introducing control edges from call sites to procedure entry nodes, and from entry to other nodes in the procedure – except entry- and exit nodes, as shown in Figure 1.

The transitive flow of data and control dependences form a *dependence chain*, which is a sequence of nodes n_1, n_2, ..., n_k, where node n_{i+1} is either directly data- or control dependent on node n_i for all i, $1 \leq i \leq k-1$. Nodes n_2, n_3, ..., n_k are said to be *affected* by node n_1. A path p *covers* the dependence chain if it goes through chain nodes n_1, n_2, ..., n_k, and each subpath of p between nodes n_i and n_{i+1} is either definition-clear with respect to the variable defined at n_i (data dependence), or all the nodes of the subpath are control dependent on n_i (control dependence), respectively. The dependence chain is realizable if it can be covered by a realizable path.

A slicing criterion is a pair $C=<I,\ V>$, where I is a program point and V is a subset of program variables. The backward static program slice S with respect to slicing criterion C consists of all the parts of the program that have direct or indirect effect on the values computed for variables V at I. In forward static program slicing, statements depending on the slicing criterion C are computed, where V is a set of variables defined at I. Computing a program slice requires determining the nodes of possible dependence chains that end (backward slicing), or start (forward slicing) at the slicing criterion, respectively. The program slicing method is considered to be precise up to realizable program paths if the slice is computed upon realizable dependence chains.

2.1 Program Slicing via Token Propagation

The token propagation-based static program slicing method has been presented in [21]. The idea of the approach is to discover possible dependence chains by propagating tokens of the control flow graph starting from the slicing criterion. Tokens contain a token index corresponding to a defined variable (initially the variable of the slicing criterion) and a backtrack index used to control token propagations from procedure exit nodes (considering realizable program paths). Tokens are propagated along definition-clear paths wrt. variable corresponding to the token index; tokens propagated to affected nodes (containing use of the token index) causing these nodes marked as in the slice. Influenced definitions induce new token propagations from the affected nodes. A special $\emptyset$ backtrack index value is used to distinguish tokens having no previous "calling context", otherwise backtrack indices

correspond to variable identifiers.

The token propagation rules of forward data-flow slice computation (data-dependences are considered only) can be summarized as follows:

Rule 0. A token $RD_x^{\emptyset}$ is created for slicing criterion C=<n, $\{x\}$>, which is propagated to the successor node of n. Node n is marked as in the slice.

Rule 1. If a token RD_x^y is propagated to a node n that does not (re)define variable x, the token is propagated to the successor node(s) of n unchanged.

Rule 2. If a token RD_x^y is propagated to a node n that uses variable x, n is marked as in the slice. A new token RD_z^y is created for definition of variable z influenced by use of x, which is propagated to the successor node of n.

Rule 3. If a token RD_x^y is propagated to a call site, token RD_x^x is propagated to the entry node of the called procedure.

Rule 4. Any call site c that contains a token RD_x^y and exit node e (of the called procedure) that contains a token RD_z^x induce the propagation of a token RD_z^y from return site *returnSiteOf(c)*. Token $RD_z^{\emptyset}$ is propagated from an exit node to all return sites unchanged.

The token propagation stops when no more propagation is possible (a given token is propagated to a given node once), and the slice is given by the set of nodes marked as in the slice. Notice that a token RD_z^x propagated to a procedure exit node directly corresponds to procedure summary edge: $x \rightarrow z$ of Horwitz et al. [28]. (Summary edges represent the transitive dependences due to the procedure call.)

Control tokens are created at affected predicate nodes (using a special token index value C) and propagated along control edges to accommodate control dependences. Nodes reached by control tokens are marked as in the slice; definitions in control dependent nodes start new (data) token propagations. The rules of the full forward slicing are shown below:

Rule 5. If a token RD_x^y is propagated to a predicate node n that uses variable x, a new token RD_C^y is created and propagated to the nodes that are control dependent on n.

Rule 6. If a token RD_C^y is propagated to a predicate node n, token RD_C^y is propagated to the nodes that are control dependent on n.

Rule 7. If a token RD_C^y is propagated to a node n, n is marked as in the slice. A new token RD_z^y is created for definition of variable z, which is propagated to the successor node of n.

Rule 8. If a token RD_C^C is propagated to an entry node, token RD_C^C is propagated to all the nodes of the procedure (except entry and exit nodes).

Backward slicing can be obtained by reversing the token propagation rules of forward slicing, where tokens are propagated to predecessor nodes, backwards.

3 The Reason-why Algorithm

This section presents a method capable of reasoning about an arbitrarily selected element of the resulting slice, called the "*reason-why algorithm*". First, we restrict to forward data-flow slices; then, we extended to full forward slices. Reasoning about backward slices is just the dual of the presented method, which is hence omitted to save space. For clarity of the presentation we consider programs containing global and scalar variables. Local variables and parameter passing can be treated as described in [21].

3.1 Reasoning Data-flow Slices

We assume that we are given a slicing criterion $C=<n, \{x\}>$ for which the data-flow slice has been computed using the token propagation method. We also assume all the tokens propagated during slicing are available, and the resulting slice contains a node m to be explained; m contains a use of variable y and a token RD_y^z caused m to be marked as in the slice (Rule 2). To justify why m is included in the slice our goal is to present a definition-use chain from n to m – along with a potential execution trace that covers it. The pair (n, $RD_x^{\emptyset}$) will be referred to as the *source*; the pair (m, RD_y^z) is referred to as the *target*. We note that we provide one single, any of the possible definition-use chains between the source and the target, which is not necessarily the shortest one.

To our experiences providing a complete CFG path covering a definition-use chain contains too much detail (instructions) to overview by a human user; providing merely the nodes of the chain is not enough to see how this dependence chain can be covered by a potential program execution. The path to be constructed, called the "*reason-why path*", will hence be a definition-use chain augmented with procedure calls and returns (intraprocedural path segments between the use-definition nodes and the procedure boundaries are omitted).

To reveal a definition-use chain between n and m we trace back the token propagation performed during slicing. We start from target node m, and investigate the tokens propagated to the predecessor nodes. Based on this information we can deduce to the previously applied token propagation rule(s), and determine the node(s) from where the token propagated to m may have been originated. The predecessor node and the (possibly) new token propagated to the predecessor node become the new target. Then, we continue finding such predecessors as far as we reach the source. From procedure entry nodes we "return" to call sites, and from return sites we enter procedure exit nodes, respectively. The traversed definition-use chain nodes, as well as procedure call- and return sites are recorded; finally, this node sequence is reversed. We bypass recovering applications of Rule 1 (which propagates tokens unchanged to successors iteratively) by identifying reachable nodes along definition-clear paths backwards.

The construction of the reason-why path is performed in two passes: in Pass 1 we traverse intraprocedural-, summary- and call edges backwards (to callers), whereas in Pass 2 we traverse intraprocedural-, summary- and return edges (to

called procedures). As procedure summary edges – represented by exit node tokens in the called procedures – are available, we can cross procedure calls without ascending into the called procedures. Exploited summary edges are resolved in a subsequent step. Finally, the path is reversed to get a forward path. Note that using the two-pass method procedure calls and returns are correctly nested, i.e. the resulting reason-why path is realizable.

Pass 1

Pass 1 (as well as Pass 2) consists of a sequence of intra- and interprocedural path search steps. In the intraprocedural step our goal is to get to the entry node of the current procedure, whereas in the interprocedural step we select one of the potential callers of the current procedure from where the token propagation had been originated.

First, we consider the initial target: node m and token RD_y^z, where $z \neq \emptyset$. (If $z = \emptyset$, we skip Pass 1.) To determine the node from where RD_y^z had been propagated to m, we determine the set of nodes in the current procedure reachable along definition-clear path wrt. y backwards. The possible source(s) of RD_y^z among these nodes is either (a) the procedure entry node if $z = y$ and the entry node contains RD_y^y, (b) a node containing a definition of variable y, a use of a variable v, and a token RD_v^z (RD_y^z had been started by Rule 2), or (c) a return site of a called procedure P such that the related call site of P contains a token RD_v^z and there is a summary edge $v \rightarrow y$ (Rule 4 had been applied to RD_y^v in the called procedure's exit node). Note that as the backtrack index is not $\emptyset$, slicing criterion node n cannot be the source of RD_y^z. In either case, we record- and set the new node and the new token as the new *target*. In the case of (b) and (c), we continue searching for the next predecessor of the current target as far as we reach the entry. In the case of (c), we record the call- and the return site, as well as the summary edge used to cross the call (resolved later). To avoid infinite loop we traverse each node-token pair at most once, and use backtracking if necessary.

In the interprocedural step, we select one of the potential callers that resulted in the propagation of RD_y^y to the entry node. These call sites contain a token RD_y^v (Rule 3 had been applied). We select one of them, and apply the above intraprocedural path search for the new target (call site and RD_y^v) to get to the entry node of the caller procedure.

We continue the above procedure as far as any of the call sites contains a token $RD_y^\emptyset$, when we turn to Pass 2. In the presence of strongly-connected components (SCCs), we visit each call site and call site token at most once, which avoids infinite cycle.

As an example, let us consider the program shown in Figure 1. For slicing criterion C=(*a2*, {x}), we obtain the data-flow slice: S={*a2*, *a4*, *b2*, *m6*, *c5*}. (The related instructions are highlighted in boldface characters; tokens propagated during slicing are indicated next to the nodes in the figure). Assume that we choose node *c5* to be explained.

In Pass 1, we start from target (*c5*, RD_y^y). After identifying the set of nodes reachable (backwards) along definition-clear paths wrt. y we find return site *c3*,

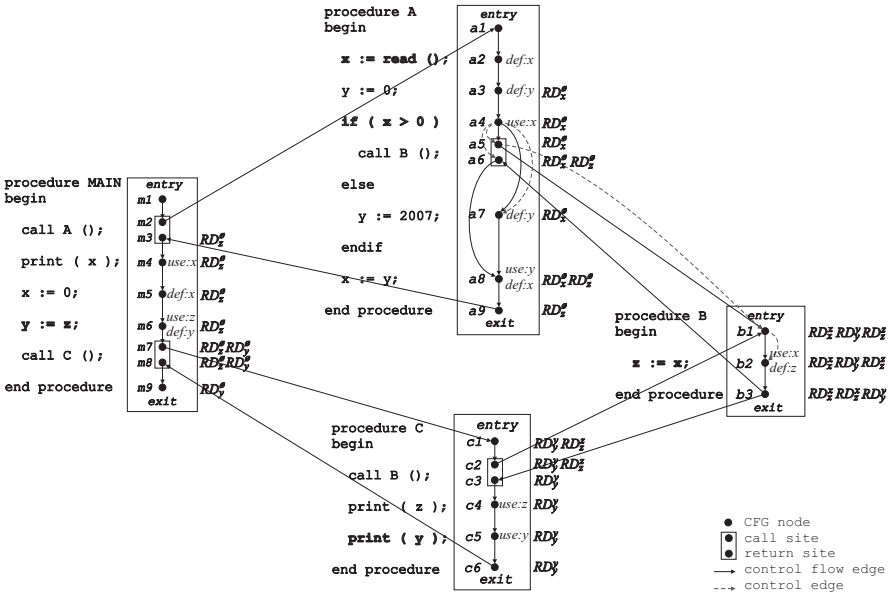

Figure 1: Example program graph and the tokens propagated during data-flow slicing

whose call site contains a token RD_y^y and the called procedure contains summary edge $y \to y$ (exit node token RD_y^y in procedure B; case c). The new target is set as node *c2* and token RD_y^y. In the next step, we reach procedure entry node *c1* (case a).

In the interprocedural step, we return to call site *m7*, as it contains a token $RD_y^\emptyset$, so we finish Pass 1. The reason-why path constructed during Pass 1 is shown below:

1. (c5, RD_y^y) -- use of y
2. (c3, RD_y^y) -- return from B, summary edge: $y \to y$
3. (c2, RD_y^y) -- call B
4. (c1, RD_y^y) -- entry C
5. (m7, $RD_y^\emptyset$) -- call C

Pass 2

During Pass 2 we traverse intraprocedural- and return edges, and trace back the propagation of $RD_y^\emptyset$ towards the slicing criterion.

The intraprocedural path search starts from a call site (following Pass 2), or from node m, respectively (m contains a token $RD_y^\emptyset$). The potential source of this token is a node reachable from the current target along definition clear-path wrt. y backwards, which is either (a) node n if $y = x$, (b) a node containing a definition

of variable y, a use of a variable v, and a token $RD_v^{\emptyset}$ (Rule 2), (c) a return site such that the related call site contains a token $RD_v^{\emptyset}$ and there is summary edge RD_v^y (Rule 4), or (d) a return site such that the called procedure's exit node contains the token $RD_y^{\emptyset}$ (Rule 4 is applied to a token with $\emptyset$ backtrack index). In the case of (a), we finish Pass 2; in the case of (b) or (c), we continue the intraprocedural search; in the case (d), we set the exit node of the called procedure and $RD_y^{\emptyset}$ as the new target (interprocedural step). We continue the above procedure as far as we reach n.

In the example, in Pass 2, we start from node *m7* and token $RD_y^{\emptyset}$. The only reachable node is node *m6*, which defines y, uses z, and contains a token $RD_z^{\emptyset}$ (case b). The new target is set as (*m6*, $RD_z^{\emptyset}$). In the next steps, we select return site *m3* and exit node *a9* of procedure A, which contains $RD_z^{\emptyset}$ (case d). The source of token $RD_z^{\emptyset}$ propagated to *a9* is return site *a6*, since there is a token $RD_x^{\emptyset}$ in *a5*, and the called procedure contains summary edge $x \to z$. From target (*a5*, $RD_x^{\emptyset}$) slicing criterion node *a2* is reachable, and token index x corresponds to the variable of the slicing criterion (case a), so Pass 2 finishes too.

The path constructed in Pass 2 is as follows:

```
6.  (m6, RD_z^∅) -- use of z, definition of y
7.  (m3, RD_z^∅) -- return from A
8.  (a9, RD_z^∅) -- exit A
9.  (a6, RD_z^∅) -- return from B, summary edge:  x → z
10. (a5, RD_x^∅) -- call B
11. (a2, RD_x^∅) -- definition of x
```

Resolving Summary Edges

The reason-why path potentially contains "jumps" from return- to call sites via summary edges that need to be resolved. It requires constructing a coverage path for a dependence-chain realizing the procedure summary. We iterate over each adjacent call- and return sites contained in the reason-why path, resolve them one-by-one, and insert the related summary edge coverage path into the original reason-why path between the related call- and return site pair.

The construction of the coverage path for a summary edge $v \to y$ is performed correspondingly to the intraprocedural path search applied in Pass 1: for a given call site c and return site r we construct a reason-why path from the exit node of the called procedure and token RD_y^v (target) to the entry node of the called procedure and token RD_v^v (source). Once this path has been constructed, it is inserted between the call- and return site pair.

Resolving a summary edge may introduce new summary edges (case c), which also need to be resolved, recursively. In the presence of SCCs, during resolving a summary edge the same summary edge could potentially be reused. As during resolving a summary edge there must exists a path that does not reuse itself (otherwise, it would mean an infinite loop in the code, so the summary edge would have

never been computed), excluding the reuse of the same summary edge currently being resolved, the infinite loop can be avoided.

By reversing the resulting path we obtain the required definition-use chain containing a proper sequence of call- and return sites.

Continuing with the example, the reason-why path contains two summary edges, at positions 2 and 9, which need to be resolved. The first summary edge $y \rightarrow y$ is resolved by starting from exit node *b3* in procedure B and token RD_y^y (target). Since the entry node is reachable from the exit, and the entry node contains RD_y^y (source), the path search finishes. The path to be inserted between positions 2 and 3 is as follows:

```
1.  (b3, RD_y^y) -- exit B
2.  (b1, RD_y^y) -- entry B
```

During resolving summary edge $x \rightarrow z$ of procedure B we have to traverse node *b2* as well, which results in the following path to be inserted between positions 9 and 10:

```
1.  (b3, RD_z^x) -- exit B
2.  (b2, RD_x^x) -- use of x, definition of z
3.  (b1, RD_x^x) -- entry B
```

The resulting reason-why path is then reversed. The reason-why path from *a2* to *c5* is shown below (only target token indices are indicated – corresponding to the most recently defined variable; procedure calls and returns are tabbed; comments are substituted by actual program instructions):

```
1.    a2, x  -- x := read()
2.    a5, x  -- call B ()
3.     b1, x -- entry B
4.     b2, x -- z := x
5.     b3, z -- exit B
6.    a6, z  -- return from B
7.    a9, z  -- exit A
8.   m3, z   -- return from A
9.   m6, z   -- y := z
10.  m7, y   -- call C ()
11.   c1, y  -- entry C
12.   c2, y  -- call B ()
13.    b1, y -- entry B
14.    b3, y -- exit B
15.   c3, y  -- return from B
16.   c5, y  -- print (y)
```

3.2 Reasoning Full Slices

In full slicing, data dependent predicates induce propagations of control tokens along control edges, which also need to be considered at constructing the reason-why path.

If target node m contains control token only, the initial target is of the form (m, RD_C^z). During the intraprocedural path search we determine the set of *controlling* nodes, i.e. the nodes from where there is a control edge to m. The possible source(s) of token RD_C^z among these nodes is either (a) a predicate node containing a use of a variable v and a token RD_v^z (Rule 5), (b) a predicate node containing RD_C^z (Rule 6), or (c) the procedure entry node if $z = C$ and the entry node contains RD_C^C (Rule 8). The new node and the new token are set as the new target. This intraprocedural search step is applied in passes 1 and 2 each time the origin of a control token needs to be determined.

Another change in reasoning full slices is that control tokens induce data-tokens at definition nodes (Rule 7); hence, at determining the possible sources of a data token RD_y^z, nodes containing definition of variable y and token RD_C^z need to be investigated as well. If it holds for some node, this node and RD_C^z are also a potential new target during the intraprocedural path search.

When the target token is a control token, the interprocedural step in Pass 1 requires determining the set of call sites containing control token. In Pass 2, as no control token can be propagated to a procedure exit node, the interprocedural traversal is unchanged.

Using the above extensions, a reason-why path can also be calculated for elements of full slices.

4 Related Work

Various algorithms for calculating interprocedural slices exist. The first method published by Weiser [35] is not context-sensitive. There are studies [1, 32, 6, 31] investigating whether considering calling-context has significant affect on the size of the slices. It may occur that inaccurate slices due to following non-realizable paths are several times larger than precise ones – what is more, the computation of these extra large slices may take more time.

There are a number of context-sensitive static slicing methods. Most of them are based on system dependence graphs published first by Horwitz et al. [28]. By computing transitive dependences due to procedure calls (summary edges), slicing is reduced to a graph reachability problem. Agrawal and Guo [1] have presented an explicitly context-sensitive slicing method over the SDG (without summary edges), in which the call stack is maintained during the propagation. Krinke [32] presented a corrected explicitly context-sensitive algorithm. Atkinson and Griswold [3] used CFGs and the invocation graph approach [13] for context-sensitive slicing. Liang and Harrold [33] proposed a precise slice computation method also based on data-flow information propagation over the CFG.

To our knowledge no reasoning technique has been proposed to justify slice elements computed by these methods.

Hajnal and Forgács [21] presented a context-sensitive static slicing method which combines the demand-driven nature of the CFG-based slicing and the efficiency of the SDG-based slicing, using token propagation. The reason-why algorithm proposed in this paper makes it possible to justify slice elements computed that method but also applicable to reason about slice elements computed by any other technique which is at least as precise as our token propagation-based method. For example, our method can be applied for SDG-based slicing considered the most wide-spread method nowadays

Chopping [29] is a variant of program slicing capable of revealing statements involved in a transitive dependence from one specific statement (source criterion) to another one (target criterion). A chop is basically the intersection of the forward slice of the forward criterion and the backward slice of the backward criterion, which provides a more focused approach to investigating how one statement affects the other. Considering a chop, which gives a set of nodes composed of (all) the dependence chains between the source and the target, it can be still very difficult to construct a dependence chain from source to target – and, if given that, an appropriate calling sequence that covers these nodes. The solution proposed in this paper answers both questions. We are aware of no other similar techniques for this problem.

5 Conclusions

To our knowledge no automated reasoning technique about the computed slice elements has been proposed in the literature so far. Without such a tool verification or understanding of the resulting program slices requires considerable expertise and time. This paper proposes a solution to "explain" slice elements by computing a specific dependence chain from the slicing criterion to the chosen slice element. This definition-use chain, augmented with control information, is more easily overviewed or analyzed by a human user.

We implemented the presented reason-why algorithm in the Java programming language and integrated with the slicing tool presented in [21]. We carried out several experiments on the same COBOL systems and slices computed in programs of different sizes. The results showed that in all the cases the slice computation time dominates the time of the reason-why path computation (it took only a few seconds in the worst case). It is because the reason-why algorithm only reads the available token information and performs no compute-intensive operations (such as slicing). Note that slice computation has to be performed once; then several reasoning tasks can be initiated on the resulting slice elements.

In the presented method, the dependence chain is determined arbitrarily – tracing back any of the possible token propagations performed during previous slicing. Shorter chains are however easier to understand, therefore we plan to investigate how to provide shorter paths from source to target (e.g., by also introducing a

kind of distance information from source into tokens). As the number of tokens may be huge in the case of large programs, it is an interesting question how the increased memory requirements and its maintenance cost affect the overall slicing performance, and in what extent the reason-why chains can be shortened. Also, to our experiences data-dependences are easier to follow mentally, hence, we should be able to give option for selecting data-dependences (priorize or let the user decide interactively) where both types of dependences arise. It would also be worth investigating that if the provided path has been found infeasible by the user, how the algorithm can search for alternative path. The latter issues imply further possibilities for improvement: how to visualize or represent the reason-why path (which is currently plain XML), and how to make the path search interactive, respectively. These serve as the basis of our future work.

References

[1] Agrawal, G., and Guo, L. Evaluating explicitly context-sensitive program slicing. *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 6–12, 2001.

[2] Agrawal, H., and Horgan, J. Dynamic program slicing. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI). ACM, New York, USA, pages 246–256, 1990.

[3] Atkinson, D.C., and Griswold, W.G. The design of whole-program analysis tools. *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, 1996.

[4] Binkley, D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8): 498–516, 1997.

[5] Binkley, D. The application of program slicing to regression testing. *Information and Software Technology Special Issue on Program Slicing*, 40(11–12): 583–594, 1998.

[6] Binkley, D., and Harman, M. A large-scale empirical study of forward and backward static slice size and context sensitivity. *Proceedings of the International Conference on Software Maintenance*, pages 44–53, 2003.

[7] Binkley, D., Horwitz, S., and Reps, T. Program integration for languages with procedure calls. *Transactions on Programming Languages and Systems*, 4(1): 3–35, 1995.

[8] Canfora, G., Cimitile, A., and De Lucia, A. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11–12): 595–607, 1998.

[9] Canfora, G., Cimitile, A., and Munro, M. RE2: Reverse engineering and reuse reengineering. *Journal of Software Maintenance: Research and Practice*, 6(2): 53–72, 1994.

[10] Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G.A. Software salvaging based on conditions. *Proceedings of the International Conference on Software Maintenance*, pages 424–433, 1994.

[11] Cimitile, A., De Lucia, A., and Munro, M. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3): 145–178, 1996.

[12] De Lucia, A., Fasolino, A.R., and Munro, M. Understanding function behaviours through program slicing. *Proceedings of the 4th IEEE Workshop on Program Comprehension*, pages 9–18, 1996.

[13] Emami, M., Ghiwya, R., and Hendren, L.J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 20–24, 1994.

[14] Forgács, I., Takács, É., and Hajnal, Á. Regression slicing and its use in regression testing. *Proceedings of IEEE International Computer Software and Applications Conference*, pages 464–469, 1998.

[15] Gallagher, K.B. Evaluating the surgeon's assistant: Results of a pilot study. *Proceedings of the Conference on Software Maintenance*, pages 236–244, 1992.

[16] Gallagher, K.B., and Lyle, J.R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8): 751–761, 1991.

[17] Griswold, W.G. Making slicing practical: the final mile. *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, page 1, 2001.

[18] Gupta, R., Harrold, M.J., and Soffa, M.L. An approach to regression testing using slicing. *Proceedings of the Conference on Software Maintenance*, pages 299–308, 1992.

[19] Gupta, R., Soffa, M.L., and Howard, J. Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Transactions on Software Engineering and Methodology*, 6(4): 370–397, 1997.

[20] Gyimóthy, T., Beszédes, Á., and Forgács, I. An efficient relevant slicing method for debugging. *Lecture Notes in Computer Science*, pages 303–321, 1999.

[21] Hajnal, Á., and Forgács, I. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software: Evolution and Process*, 24(1): 67–82, 2012.

[22] Harman, M., and Danicic, S. Amorphous program slicing. *Proceedings of the 5th International Workshop on Program Comprehension*, pages 70–79, 1997.

[23] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3): 143–162, 1995.

[24] Harman, M., Hierons, R.M., Fox, C., Danicic, S., and Howroyd, J. Pre/Post conditioned slicing. *Proceedings of the IEEE International Conference on Software Maintenance*, pages 138–147, 2001.

[25] Hierons, R., Harman, M., and Danicic, S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4): 233–262, 1999.

[26] Hierons, R., Harman, M., Fox, C., Ouarbya, L., and Daoudi, M. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12(1): 23–28, 2002.

[27] Horwitz, S., Prins, J., and Reps, T. Integrating non-interfering versions of programs. *Transactions on Programming Languages and Systems*, 11(3): 345–387, 1989.

[28] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1): 26–60, 1990.

[29] Jackson, D. and Rollins, E.J. A new model of program dependences for reverse engineering. *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 2–10, 1994.

[30] Korel, B., and Laski, J. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[31] Krinke, J. Effects of context on program slicing. *Journal of Systems and Software*, 79(9): 1249–1260, 2006.

[32] Krinke, J. Evaluating context-sensitive slicing and chopping. *Proceedings of the International Conference on Software Maintenance*, pages 22–31, 2002.

[33] Liang, D., and Harrold, M. J. Reuse-Driven Interprocedural Slicing in the Presence of Pointers and Recursion. *Proceedings of the IEEE International Conference on Software Maintenance*, pages 421–430, 1999.

[34] Venkatesh, G.A. The semantic approach to program slicing. *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 107–119, 1991.

[35] Weiser, M. Program slicing. *IEEE Trans. Software Eng.*, 10(4): 352–357, 1984.

Received 10th April 2012