# Practical Application of Program Slicing and Its Use in Software Testing and Maintenance

Ph.D. Thesis

**Ákos Hajnal**

Research Supervisors:

István Forgács, Ph.D.

László Zsolt Varga, Ph.D.

Eötvös Loránd University (ELTE), Faculty of Informatics (IK)

Doctoral School of Informatics (András Benczúr, D.Sc.)

Foundations and Methodology of Informatics (János Demetrovics, D.Sc.)

Computer and Automation Research Institute

Hungarian Academy of Sciences (MTA SZTAKI)

Budapest, Hungary, 2012

# Declaration

Herewith I confirm that all of the research described in this dissertation is my own original work and expressed in my own words. Any use made within it of works of other authors in any form, e.g., ideas, figures, text, tables, are properly indicated through the application of citations and references. I also declare that no part of the dissertation has been submitted for any other degree – either from the Eötvös Loránd University or another institution.

*Ákos Hajnal*

*Budapest, December 2012*

# Acknowledgements

First of all, I wish to thank my supervisor István Forgács for introducing me to the science of software testing. Without his constant support and guidance this dissertation would have never been completed. I am also grateful to László Zsolt Varga for his support and suggestions.

I greatly acknowledge the contribution of the Computer and Automation Research Institute of the Hungarian Academy of Sciences (MTA SZTAKI), where I performed the research presented.

Last but not least, I am very thankful for the encouragement of my family.

*To my grandmother*

# Contents

Introduction ..... 1

1.1 Program Slicing ..... 2

1.2 Program Slicing in Practice ..... 5

1.3 Precision ..... 6

1.4 Scalability ..... 9

1.5 System Issues ..... 10

1.6 Motivation ..... 11

1.7 Overview ..... 12

1.8 Accomplishments ..... 13

Slicing via Token Propagation ..... 14

2.1 Definitions ..... 15

2.2 Forward Data-flow Slicing ..... 22

2.3 Forward Slicing ..... 40

2.4 Backward Slicing ..... 43

2.5 Local Variables, Parameter Passing ..... 45

2.6 Related Work ..... 47

2.7 Conclusions ..... 50

Evaluation ..... 51

3.1 Prototype Implementation ..... 52

3.2 Subject Programs ..... 57

3.3 Empirical Results ..... 59

3.4 Comparison with Other Works ..... 62

3.5 Conclusions ..... 63

Understanding Program Slices ..... 64

4.1 The Reason-why Algorithm ..... 65

4.2 Related Work ..... 72

4.3 Conclusions ... 73
Further Enhancements, Applications ... 74
5.1 Reducing Token Storage via Postorder Processing ... 75
5.2 Reducing Propagations via GREF-GMOD-KILL ... 76
5.3 Reuse of Summary Edges ... 78
5.4 On-demand Computation of Flow Edges ... 79
5.5 On-demand Construction of the Du-graph ... 79
5.6 Dicing, Chopping ... 80
5.7 Conclusions ... 81
Summary ... 83
6.1 New Scientific Results ... 84
6.2 Further Research Directions ... 85
Abbreviations ... 87
Appendix: Author's Publications ... 88
Bibliography ... 92
Abstract ... 99
Kivonat ... 100

# List of Figures

Fig. 1. The slice of a program fragment wrt. slicing criterion *C*=(9, {*fact*}) ..... 3
Fig. 2. Context-insensitive and context-sensitive *forward* static program slices of a program fragment with respect to slicing criterion C = (3, {*x*}) ..... 7
Fig. 3. Example program graph ..... 25
Fig. 4. Pseudo-code of the forward slicing algorithm ..... 27
Fig. 5. XML fragment of a CFG, a slicing criterion, and the slice ..... 55
Fig. 6. Slice sizes and computation times (full slicing) ..... 61
Fig. 7. Tokens propagated during data flow slicing ..... 69

# List of Tables

Table 1. Details of the investigated systems..... 59
Table 2. Execution results of the slicing algorithm on 1000 random slicing criteria..... 60

# Chapter 1

# Introduction

The key concepts of data-flow analysis were developed in the late 1960s, which technique had become an important means of program analysis. During data-flow analysis information is gathered about the computer code how instructions affect each other through variable values calculated at various program points. Data-flow analysis is often used by compilers when optimizing computer programs (loop-invariant code motion, common subexpression elimination, simplified arithmetic expression evaluation). Over the past decades, however, the majority of new applications have focused on software quality.

The concept of program slicing proposed by Mark Weiser in 1979 extends data-flow analysis by accommodating *control dependences* (effects of data-flow on control). Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics. The original idea comes from the observation that programmers are often interested in only a portion of the program's code. The process of slicing "deletes" those parts of the program that can be determined to have no effect upon the semantics of interest. Thus program slices are typically much smaller than the whole program which can be more easily understood or maintained.

Program slicing was originally motivated to aid debugging activities. In the past three decades, various notions of program slices have been proposed as well as a number of methods to compute them. By now program slicing has numerous applications in software engineering, including software testing and maintenance, program comprehension, re- and reverse engineering, and program integration.

Slicing industrial-scale programs raises new requirements that a practical slicer must take into account. This chapter investigates the applicability of existing methods to large-size legacy COBOL codes.

Section 1.1 introduces the basic concept of program slicing as well as its main forms and applications. Previous implementations are overviewed in Section 1.2. Sections 1.3, 1.4, and 1.5 discuss the barriers of application of existing techniques on industrial-scale COBOL codes. Section 1.6 presents the motivation of this work. Section 1.7 overviews the

structure of the remainder of the thesis. The main contributions are introduced in Section 1.8.

## 1.1 Program Slicing

The concept of *program slicing* was first introduced by Mark Weiser in his Ph.D. thesis in 1979. His work was also presented at a conference [Weiser 1981] and in a software engineering journal in 1984 [Weiser 1984]. The rapid admission of his idea reflected the growing demand for such analysis and its high potential for application in different software engineering areas. The motivation for slicing derives from the observation that large computer programs are more easily understood or maintained when broken into smaller pieces. Unlike design-time decomposition techniques, slicing is applied to programs after they are written and allows slicing to be performed automatically on the actual program text.

Weiser summarized program slicing as follows:

> *Program slicing is a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior.*
>
> Mark Weiser [1981]

Finding such “ideal slices” however proved to be unsolvable in general: there cannot be a slicing method that can guarantee the minimality of the slices or behaviour equivalence, respectively. Syntactic restriction, namely, the slice is an independent program that can be compiled and run is also relaxed in some techniques later. Weiser therefore proposed a more practical definition for slicing based on *data* and *control flow* [Weiser 84] to enable exact slicing algorithms.

The motivation of program slicing was to answer the question: “Which are the statements that (potentially) *affect* the variable values computed at some program point?” Weiser presented experimental evidence that programmers already use slicing during debugging – mentally. Having picked a statement and a variable (or a set of variables) at

which the error becomes visible, the program slice can show the statements that may participate in the computation of the erroneous value. The selected program point *I* and the set of variables of interest *V* are called a *slicing criterion*, denoted as *C*=(*I*, *V*). A slice is typically much smaller than the whole program, thus the bug can be found easier and faster.

For illustration, consider the following program fragment shown in Figure 1 and the related program slice with respect to slicing criterion *C* = (9, {`fact`}) (statement in line 9 and variable `fact`). The slice contains statements in lines 2, 4, and 6, in addition to the print statement of the slicing criterion. The assignment statement in line 6 has *direct* effect on the variable value printed in line 9 (*data dependence*), furthermore, the value assigned in line 6 is *dependent* on the initialization statement in line 2. The outcome of the conditional statement in line 4 determines the execution of statement 6 (*control dependence*), therefore it is also included in the slice. Other statements have no effect on the slicing criterion. The defect, which is due to the erroneous initialization of variable `fact`, can be found easier, as the slice contains the relevant statements only.

*Program fragment:*

```
...
1  var sum := 0;
2  var fact := 0;
3  var qsum := 0;
4  for (var i := 1; i <= 10; i++)
   {
5    sum := sum + i;
6    fact := fact * i;
7    qsum := qsum + i * i;
   }
8  print(sum);
9  print(fact);
10 print(qsum);
...
```

*Program slice:*

```
2  var fact := 0;

4  for(var i := 1; i <= 10; i++)
   {

6    fact := fact * i;

   }

9  print(fact);
```

**Fig. 1. The slice of a program fragment wrt. slicing criterion *C*=(9, {*fact*})**

Weiser's method 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 (in the opposite direction of the program execution); and *static*, because the analysis is made without having specified any particular program execution (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.

Since Weiser's method, other forms of program slicing have been evolved such as dynamic slicing [Korel and Laski 1988; Agrawal and Horgan 1990], quasi-static slicing [Venkatesh 1991], conditioned slicing [Canfora et al. 1998], amorphous slicing [Harman and Danicic 1997], hybrid slicing [Gupta et al. 1997], and relevant slicing [Gyimóthy et al. 1999].

While a static slice represents the original program's behaviour for any of the program inputs with respect to the slicing criterion, a dynamic slice discovers effects along a given execution trace only. *Dynamic slices* can therefore be much "thinner" than their static counterparts. *Quasi-static slicing* achieves smaller slices by fixing some of the input variables while others may vary. *Conditioned slicing* is a generalization of quasi-static slicing in the sense that it enables specifying any set of input variables by a first order logic formula (used by a symbolic executor). *Amorphous slicing* removes the limitation related to traditional syntax preserving slicing (i.e. simplification via statement deletion), so it can also obtain smaller, and sometimes more meaningful slices, retaining the semantic property of the original program. *Hybrid slicing* integrates dynamic information into static slicing to more accurately estimate the potential paths taken by the program. *Relevant slicing* extends dynamic slicing by including potentially affecting statements as well, which actually did not affect the variable of interest but could have affected it had they been evaluated differently.

Weiser presented two main applications of slicing. One is to aid program debugging and maintenance; the other is to derive slicing-based program metrics about structuring of the program (*coverage*, *overlap*, *clustering*, *parallelism*, *tightness*). Since then, program slicing has found its applications in other areas of software engineering as well, including software testing [Gupta et al. 1992; Harman and Danicic 1995; Binkley 1997; Binkley 1998; Forgács et al. 1998; Hierons et al. 1999; Hierons et al. 2002], software maintenance [Gallagher et al. 1991; Gallagher 1992; Canfora et al. 1994a; Cimitile et al. 1996], program

comprehension [De Lucia et al. 1996; Harman et al. 2001], reverse engineering [Canfora et al. 1994b], and program integration [Horwitz et al. 1989; Binkley et al. 1995].

Static program slicing can support software maintenance and testing in determining that the modification of a component does not interfere with unmodified components, in dividing the program into smaller parts for test case creation, and in focusing regression testing effort on the part of the code that is really affected by a change. Dynamic slicing can be especially useful in debugging by narrowing the focus on the statements potentially containing the bug. Conditioned and amorphous slicing can be an efficient means of program comprehension, reverse engineering, program integration, function isolation, and reusable component extraction.

Among the supporters of the papers published on this topic we can find several leading IT companies such as AT&T, DEC, Hewlett Packard, IBM, Intel, Xerox, as well as military related companies and organizations such as Lockheed Martin, Air Force Office of Scientific Research, Defense Advanced Research Projects Agency, Office of Naval Research, and U.S. Army Research Office. It shows that program slicing is an important means of aiding development and maintenance of highly reliable, safety-critical systems.

This thesis concerns with the fundamental issues related to static program slicing; other forms of program slicing and particular applications are not investigated in details.

## 1.2 Program Slicing in Practice

In the past decades there has been a substantial research effort devoted to program slicing, resulting in over five hundred papers on this topic [Xu et al. 2005]. A number of static slicers have been implemented for C programs such as Aristotle [Harrold and Rothermel 1997], CANTO [Antoniol et al. 1997], ChopShop [Jackson and Rollins 1994a], CodeSurfer [CodeSurfer], Ghinsu [Livadas and Alden 1993], Sprite [Atkinson and Griswold 1996], Spyder [Agrawal et al. 1993], Surgeon's Assistant [Gallagher 1990], Unravel [Lyle and Wallace 1997], ValSoft [Krinke and Snelting 1998]. Furthermore, there are slicers for Java [Indus], FORTRAN (FOCUS [Lyle 1984]), Pascal (Osaka [Nishimatsu et al. 1999]), and Oberon (Steindl's slicer [1998])[1]. To our knowledge, only CodeSurfer has become a commercial product.

[1] A more detailed description and evaluation of these tools can be found in: [Hoffner et al. 1995; Krinke 2003].

Why program slicing tools are not widely used today? William Griswold pointed out some of the possible reasons in his talk: *Making Slicing Practical: The Final Mile* [Griswold 2001]:

- precise algorithms are too expensive in practice,
- algorithms that lack scalability are impractical for real-world programs,
- system issues play a considerable role in the performance,
- slices without explanation are often too difficult to understand.

The last point is discussed in Chapter 4 in details; the first three problems are investigated in the following sections.

## 1.3 Precision

Imprecision in the computed slices may derive from different sources. Some of them are not avoidable, as they relate to problems that cannot be solved in general. One of such a problem is the undecidability of whether a statically selected program path (a potential execution trace) is *feasible* or not, i.e., there exists a program input that forces the actual execution of that path. A solution to this problem would require solving the system of conditions represented by predicates along the path, which can be arbitrary in general. The same problem arises in domain testing, where only heuristics can be used to find input inside predicate borders [Forgács and Hajnal 1998a].

Another problem is with the use of pointers (pointers to data, function pointers, references in object-oriented programming, etc.) that take their particular values at run-time, since statically it is not possible to determine which data they actually point to. Static program slicing techniques, hence, typically use safe approximations, heuristics to narrow the set of potential pointer values as much as possible, or apply a conservative approach.

Unlike these problems, it can be decided whether a program path is *realizable*. The problem derives from the most fundamental program structuring principle to extract commonly used computations: the decomposition of the program into procedures (subroutines, functions, methods). Procedures can be called from different sites, but when the execution of a procedure body finishes, the program execution must continue after the site of the procedure's most recent call. In other words, realizable paths correctly nest call and return sites. Omitting the calling context during the analysis, i.e., returning to all

calling procedures at procedure exits, however, increases the number of paths to be investigated by involving those ones that cannot occur during real program execution. In this way, a large amount of unnecessary statements may be included in the slice due to "false" effects along non-realizable paths which reduces the usefulness of the resulting slice.

Program fragment:

```
1  var x;

2  proc main () {
3    x := 1;
4    call A();
5    print(x);
6    call B();
   }

7  proc A () {
8    x++;
   }

9  proc B () {
10   x := 0;
11   call A ();
12   print(x);
   }
```

Context-*insensitive* slice:

```
1  var x;

2  proc main () {
3    x := 1;
4    call A();
5    print(x);
6    call B();
   }

7  proc A () {
8    x++;
   }

9  proc B () {

11   call A ();
12   print(x);
   }
```

Context-*sensitive* slice:

```
1  var x;

2  proc main () {
3    x := 1;
4    call A();
5    print(x);

   }

7  proc A () {
8    x++;
   }
```

**Fig. 2. Context-insensitive and context-sensitive *forward* static program slices of a program fragment with respect to slicing criterion C = (3, {*x*})**

In Figure 2, a program fragment and the related context-*in*sensitive and context-sensitive forward static program slices are shown with respect to slicing criterion $C$ = (3, {x}). The assignment statement in line 3 affects the increment statement in line 8 (through the call in line 4), which affects the print statement in line 5 after return. If the calling-context is ignored, the effect of the increment statement in line 8 is analyzed in both callers: `main` and `B`, therefore the print statement in line 12 will also be included in the resulting slice. It is incorrect, since the effect of assignment in line 3 is *invalidated* in line 10 prior to line 12 along all realizable paths.

There are studies [Agrawal and Guo 2001; Krinke 2002; Binkley and Harman 2003; Krinke 2006] investigating whether considering the calling-context has significant affect on the slice sizes. These studies showed that inaccurate slices due to following non-realizable paths can be several times larger than the ones that consider realizable program

paths. What is more, the computation of these extra large slices may take more time that makes imprecise solutions impractical for slicing large-size programs.

There are two fundamental approaches to accounting for the calling-context problem. One is based on explicitly maintaining the call stack [Atkinson and Griswold 1996; Agrawal and Guo 2001; Krinke 2002]. The other is based on a two-pass traversal over the *system dependence graphs* (SDGs) [Horwitz et al. 1990; Reps et al. 1994].

The first approach however may cause the re-analysis of procedures several times for different call stacks; moreover, it suffers combinatorial explosion in the case of recursion due to the infinite number of possible call stacks. (The limitation of the considered context depth results in reduced precision [Krinke 2002].) Experimental investigations showed that full precision (unbounded context depth) is unaffordable even at slicing medium-size applications; therefore the approach is impractical for slicing industrial-scale programs.

The other approach is based on SDGs. A program dependence graph (PDG) [Ottenstein and Ottenstein 1984; Ferrante et al. 1987] is a directed graph in which nodes represent statements, and there are two types of edges between nodes: *data flow* and *control*. Data flow edges represent *data* dependences, whereas control edges represent *control dependences* between statements. An SDG is a collection of PDGs assigned to the procedures of a program. Parameter passing between procedures is represented by a collection of vertices associated with each call site corresponding to in and out actual parameters, and a collection of formal-in and formal-out vertices corresponding to the formal parameters at each procedure entry. Global variables are treated as "extra" parameters. Call vertices are connected to the entry vertex of the called procedure's PDG by a *call edge*, actual-in vertices are connected to their matching formal-in vertices via *parameter-in edges*, and actual-out vertices are connected to their matching formal-out vertices via *parameter-out edges*. Summary edges represent the transitive dependences due to procedure calls that are computed in advance for each procedure (between formal in and out parameters) and applied at call sites (edges are added between the corresponding actual in and out parameters).

Using SDGs, precise backward static slices (up to realizable paths) can be calculated by performing graph reachability in two passes, where each pass traverses only certain kinds of edges. In Pass 1, the traversal starts from the node of the slicing criterion and goes backwards along data flow edges, control edges, call edges, summary edges, and parameter-in edges, but not along parameter-out edges. Pass 2 starts from all actual-out vertices reached in Pass 1, and goes backwards along data flow edges, control edges,

summary edges, and parameter-out edges, but not along call or parameter-in edges. The two-pass traversal of the graph traverses all but only the realizable paths.

The slice computation of the SDG-based approach is efficient. It is due to summary edges computed in advance by which we can move across calls without descending into the called procedure. The method is often considered as the quasi-standard technique for precise interprocedural slicing. However, as we will see in the next section, the cost of preprocessing requirements of the approach may be prohibitive in the case of industrial-scale programs.

## 1.4 Scalability

*Scalability* of the slicing technique to be applied is crucial in the case of real-world programs. A slicing method is considered to be scalable if the slice computation time is more or less proportional with the size of the resulting slice – rather than the size of the program. As it is described earlier, when the call stack is recorded explicitly, the computation time may become exponential because of the infinite number of possible call stacks. Therefore the slice computation time does not merely depend on the slice size. Using SDGs, the disproportion is due to the preprocessing requirements of the approach, i.e., SDG construction, summary edge computation. (In some papers, the cost of the slice computation is considered to be the cost of the SDG traversal, which is incorrect, since they omit SDG construction cost.)

The construction of the SDG can be very expensive. Atkinson and Griswold [1996] reported that the application of SDGs for larger C programs may require prohibitive space and time. In the case of COBOL programs, we had similar experiences: the construction of the SDG may take several hours (occasionally, even days) for large-size programs. It is because to build the SDG we need to perform exhaustive data flow analysis in every procedure to discover all data dependences between statements. It is independent of what data dependences will be used at determining the program slice.

Because of the large number of global variables, which is common in COBOL programs, we need to add extra parameter vertices (in and out) at each procedure entry node and call site, and we have to compute their dependences as well (formal-in parameter vertices and actual parameter-out parameter vertices are considered as assignments).

The computation of all the summary edges can be very expensive. Even at using the improved summary edge computation technique proposed by Reps et al. [1994] – which is

currently the most efficient known global technique, the cost can be very high because of the potentially large number of summary edges. It can reach several hundred millions in real-world systems.

These factors make slice computation time of the SDG-based approach dependent on the program size rather than the size of the slice. If we wish to compute only a few slices, or the resulting slices are small, respectively, the cost of preprocessing dominates the cost of slice computation. The situation is even worse when the subject program changes frequently (in interactive contexts such as debugging), since we need to reconstruct the SDG after every program change.

Scalability, instead, would require a *demand-driven* approach that computes only the necessary information related to the actual program slice – at the time when needed. The only demand-driven summary edge computation technique was published by Orso et al. [2001], which is however not applicable to programs containing recursive procedures.

## 1.5 System Issues

Programs written in different programming languages pose different challenges to static source code analysis. Program slicing algorithms are based on some graph representation of the program (system dependence graphs, control flow graphs), therefore the proper construction of the given graph representation is crucial, which also influences the precision of the resulting slice. Variable types, control structures, and the complexity of the instruction set can significantly differ in different programming languages. For example, the use of pointers in the C programming language or polymorphism in C++ make hard to figure out statically the exact memory location(s) that the variables actually point to. Traditional control structures such as conditional branches, loops, and procedure calls are often not sufficient to represent control structures in programming languages with explicit concurrency (like Ada or Java). The set of supported operations can be simple in some programming language (e.g., in C), and they can be complex in others (e.g., in COBOL). In order to have an efficient slicing tool, we need to consider programming language specific characteristics, and choose the appropriate representation.

COBOL differs from “modern” programming languages in many aspects. One of the main differences was identified as the massive use of global variables. Our experiments showed that SDGs are not adequate for representing COBOL programs because of the large number of global variables. SDGs require introducing extra parameter vertices (in

and out) at each procedure entry and call site for each global variable. Thus, the memory requirements can be very high in the case of real-world programs. Moreover, because of COBOL's complex instructions, for each statement that assigns values to more (even several hundreds of variables at once, e.g., `MOVE` statement) we have to create multiple nodes in the SDG in such a way that each contains at most one definition (SDG-based slicing uses graph reachability). Control flow graphs (CFGs) are not sensitive to these characteristics, where basically each node represents one statement, and one node can contain assignments to arbitrary number of variables – the connection between in-node variable references and assignments can be specified as *influences* separately. For comparison, the control flow graph representation constructed for one of the investigated program systems (Chapter 3) contained a total of 210,965 nodes, whereas SDG would have required introducing more than 85 million extra parameter vertices to represent parameter passing via global variables.

Program slicing research has been mainly focused on developing general algorithms and most of the experiments concerned with analyzing source codes written in the C programming language. Other programming languages have been addressed only to some extent. This work was motivated by COBOL that has not yet been or only partly addressed by previous papers on program slicing.

Unique control structures such as indirect calls, `STOP RUN`, `GOBACK`, `PERFORM SUB1 THRU SUBNn` as well as data elements such as `REDEFINES`, `RENAMES`, and `MOVE` make the construction of the CFGs for COBOL non-trivial [Field and Ramalingam 1999; Deursen and Moonen 1999]. This thesis does not indent to describe how to construct CFGs for COBOL, which would require a much longer discussion. Instead, we focus on a novel slicing algorithm presented in the next chapter that operates over CFGs. We note that the application of the method is however not limited to COBOL, but allows a wider application to a larger class of programming languages.

## 1.6 Motivation

Rethinking of previous techniques and the development of a novel program slicing method were motivated by the difficulties raised at analyzing legacy COBOL systems. Some years ago, before the year of millennium, we proposed a theoretic solution to solve the "bomb of millennium" [Forgács and Hajnal 1998b]. At that time we faced the fact that what a large amount of COBOL codes are actively used – estimated over 240 billion lines of code, in

almost every major industry from banking to manufacturing. Program slicing could had been an efficient means in helping localizing potential bugs related to the two-digit year storage, however, when we tried to apply existing techniques, we found that none of them is suitable indeed to slice legacy COBOL programs in practice.

Though the year of 2000 has been passed without IT catastrophe, the difficulties of maintaining aging legacy systems have remained unsolved. Many of the legacy systems are more than 30–40 years old, whose maintenance is very labor-intensive and costly task. The lack of proper documentation, ad-hoc maintenance activities over such long lifetimes, and the poor logical structure of these programs make maintenance even more difficult. What is more, there is a huge risk involved in transforming and modernizing such applications, which companies are typically unwilling to undertake.

Program slicing could be a powerful tool of aiding such maintenance activities. COBOL has been fallen out of the focus of the program slicing research so far. Probably this is why previous program slicing techniques proved to be inappropriate for industrial-scale COBOL programs. However, COBOL is still the dominant language for business applications [Brown 2000]. Preliminary experimental results show that the algorithm proposed in this thesis is applicable for large-size programs. We hope that this work helps in making program slicing more widely used and practical.

## 1.7 Overview

This chapter introduced the basic concepts of program slicing, its main forms and applications. We reviewed previous program slicing techniques and identified their strengths and weaknesses with respect to their applicability to large-size programs. The remainder of the thesis is structured into five chapters.

Chapter 2 presents a novel program slicing approach based on control flow graphs. After introducing the basic concepts and definitions, an algorithm is introduced that uses token propagation to calculate precise data-flow and full program slices. The basic algorithm is then extended to local variables and parameter passing. Related work is also discussed.

Chapter 3 presents the solutions we used at implementing the slicer prototype and its evaluation on real-world COBOL systems. The results are based on a large number of test cases. Slice sizes and computation times are reported. Scalability of the approach is also discussed.

Chapter 4 presents the so called reason-why algorithm aimed at reasoning about the computed slice elements that can help users in comprehending program slices.

Chapter 5 investigates further improvement and application possibilities of the method. Different time-space tradeoffs of the algorithm design are discussed such as using postorder processing, preprocessing, reusing previous calculations, or computing flow edges on demand, respectively. Definition-use graph construction and the application of the method to other slicing variants called dicing and chopping are also described.

Chapter 6 summarizes the thesis.

## 1.8 Accomplishments

This thesis is aimed at being self-contained as much as possible for clarity reasons, therefore it contains presentations of other authors' work as indicated by citations. Besides the introduction and overview of program slicing concept, variants, and applications the main accomplishments of this thesis are:

- Analysis of existing static techniques with respect to their applicability to large-size, legacy COBOL codes and the use of program slicing in software testing and maintenance (this chapter).

- Proposal of a novel token propagation-based static program slicing approach (Chapter 2).

- Evaluation of the proposed method on industrial-scale COBOL programs (Chapter 3).

- Proposal of a novel "slice explainer" technique to aid slice comprehension (Chapter 4).

- Proposal and analysis of further possible improvements and applications (Chapter 5).

# Chapter 2

# Slicing via Token Propagation

A practical slicing method is precise, scalable, and adaptable to consider different programming language constructs and features. Approaches addressing precision by explicitly maintaining the call stack proved to be impractical in the case of large-size programs. Limiting the considered calling-context stack improves the performance but causes a reduced precision; reaching full precision (unbounded context depth) is unaffordable in the case of real-world programs (or even impossible, in the presence of recursion). The system dependence graph-based (SDG) approach calculates precise slices; however, with the increase of the investigated program size exhaustive analysis becomes overly expensive. It is especially crucial, when the program changes frequently (interactive contexts), which would require a demand-driven approach. System dependence graphs are more sensitive to program constructs often occur in legacy systems, such as the use of global variables and complex instructions, than control flow graphs.

This chapter proposes a novel static program slicing technique based on token propagation. The method calculates accurate program slices with respect to realizable program paths, and is based on control flow graphs, which have less space requirements compared to SDGs. The algorithm is conceptually simple, which allows of easy implementation, but general enough to adapt to a larger class of programming languages. Precision is obtained by propagating tokens along realizable program paths (using backtrack indices). The token propagation method is inherently demand-driven: it computes the necessary information only, when they are needed.

After having defined the basic concepts in Section 2.1, a forward data-flow slicing algorithm is introduced in Section 2.2, which presents the basic idea of the approach. Data-flow slicing considers data dependences only. In Section 2.3, the method is extended to compute full forward program slices by accommodating control dependences. Section 2.4 describes how the token propagation can be reversed to compute backward program slices. Section 2.5 describes how the method can be applied to programs using local variables and

parameter passing between procedures (or programs). Finally, the presented method is compared to other related techniques in Section 2.6.

## 2.1 Definitions

Computer programs can be represented by directed graphs called *control flow graphs* (CFGs), in which nodes correspond to the statements and edges represent the possible flow of control between them.

First, we define the *intraprocedural* control flow graph for one procedure; then we assemble the *interprocedural* graph representation of the program composed of multiple procedures.

DEFINITION 2.1 (INTRAPROCEDURAL CONTROL FLOW GRAPH). An *intraprocedural control flow graph* (iCFG) $G = (N, E)$ of procedure $P$ is a directed graph in which $N$ contains one node for each statement (or basic block[2]) in $P$, and $E$ contains edges that represent the possible flow of control between the statements in $P$. $N$ contains two distinguished nodes: $n_e$ and $n_x$, representing unique entry and exit points of $P$. A *predicate node* that represents the predicate of a conditional statement has exactly two successors. $n_x$ has no successors, $n_e$ has exactly one successor. Each node in $N$ is reachable from $n_e$, and $n_x$ is reachable from each node in $N$.

We note that *unreachable statements* are not represented in the iCFG, and if $P$ contains multiple exit points, $E$ contains an edge from these nodes to $n_x$[3].

DEFINITION 2.2 (INTERPROCEDURAL CONTROL FLOW GRAPH). An *interprocedural control flow graph* (ICFG) $G = (N, E)$ is a directed graph composed of one or more iCFGs associated with each procedure of the program that are linked interprocedurally by *call* and *return edges* as follows: $N$ is composed of the set of iCFG nodes but each node that represents a call statement is split into two nodes: a *call site* and a *return site*. The *callSiteOf*, *returnSiteOf* operators are used to refer to call site $c$ and return site $r$ belonging

[2] CFGs can also be built from basic blocks that represent single-entry, single-exit statement sequences.

[3] It is the classical method of treating multiple exit points that was adequate in the case of COBOL. We note that in some programming languages multiple exit points cannot be handled in this way, and so another sort of augmentation of the CFG might be necessary.

together such that *c=callSiteOf(r)* and *r=returnSiteOf(c)*. *E* is composed of the set of iCFG edges, and it is augmented with *interprocedural* edges such that a *call edge* is added from every call site *c* to the entry node of the iCFG associated to the called procedure, and a *return edge* from the exit node of the called procedure's iCFG to the return site *returnSiteOf(c)*.

This definition of interprocedural control flow graphs corresponds to the definition of *super graphs* of Myers [1981]. Note that call sites and return sites are not (directly) connected.

In the literature, both intra- and interprocedural control flow graphs are often referred to as *control flow graphs* (CFGs), for short. In the following, we shall also use the short notation, wherever the distinction is not relevant. We also note that edges of the CFG are referred to as *control flow edges* to distinguish from *control edges* representing *control dependences* (described later).

Calling relationships between procedures can be represented by a directed graph:

DEFINITION 2.3 (CALL GRAPH). The *call graph* is a directed graph $G = (N, E)$ in which $N$ contains one node for each procedure, and there is an edge in $E$ from node $n_i$ to node $n_j$ ($n_i$, $n_j \in N$) if procedure corresponding to $n_i$ contains call to procedure corresponding to $n_j$.

Call graphs are often defined as directed multi-graphs, where there can be more than one edge between nodes $n_i$ and $n_j$ if $n_i$ calls $n_j$ multiple times.

The call graph contains a distinguished node corresponding to the *main* procedure that gets the control first when the program is being executed. It is assumed that every procedure is reachable from the main procedure in the call graph.

In the absence of recursion, procedures can be sorted such that calling procedures precede the called ones:

DEFINITION 2.4 (RPOSTORDER). *rPostorder* is a linear ordering of nodes of the (acyclic) call graph *G* in which each node comes before all nodes to which it has call.

rPostorder (*reverse-postorder*, *r-postorder*, also known as topological sorting) can be determined by using a depth-first search in the call graph starting from the node of the main procedure.

A program may contain recursive procedures. The recursive regions of the call graph are called *strongly connected components*:

DEFINITION 2.5 (STRONGLY CONNECTED COMPONENT). A *strongly connected component* (SCC) of a call graph $G$ is a (maximal) subgraph of $G$ in which each node is reachable from every other node.

Strongly connected components can be contracted to a single node; the resulting graph is a directed acyclic graph (DAG) for which rPostorder sequence can be determined.

Potential program executions can be represented by paths in the CFG:

DEFINITION 2.6 (PATH). A *path* $p = n_1, n_2, \ldots, n_k$ ($k$>1) in the CFG $G = (N, E)$ is a sequence of nodes such that for every consecutive pair $(n_i, n_{i+1})$ there is an edge in $E$ for $i = 1, 2, \ldots, k$-1.

Note that paths – in contrast to real program executions – do not necessarily start from the entry node of the main procedure.

We define an abstract *call stack* for paths:

DEFINITION 2.7 (CALL STACK). The *call stack* of a path $p = n_1, n_2, ..., n_k$ in the CFG at node $n_i$ ($\in p$) is a stack of call sites (initially empty), onto which node $n_j$ is pushed if $n_j$ is a call site, or the topmost node is popped off (no operation when the stack is empty) if $n_j$ is an exit node, for the sequence of nodes $j = 1$ to $i$-1.

Paths that incorrectly nest call and return sites cannot occur during real program execution, therefore we distinguish *realizable* paths:

DEFINITION 2.8 (REALIZABLE PATH). A path $p = n_1, n_2, ..., n_k$ is *realizable* if at each exit node $n_x$ ($\in p$) either the call stack is empty or for return site $r$ following $n_x$ and call site $c$ popped off at $n_x$ the condition $r$=*returnSiteOf(c)* holds.

Realizable paths are also known as *valid* paths and they can also be defined using a context-free grammar [Reps 1993]. Note that when the call stack is not empty on return,

the return site must *match* the call site on the top of the call stack. At empty call stack, realizable paths are allowed to ascend to calling procedures, and/or descend to called procedures without a return, respectively (unbalanced paths). Any *subpath* (subsequence) of a realizable path is also realizable.

We distinguish *same-level* realizable paths. A same-level realizable path is a realizable path that starts and ends in the same procedure and every call has the corresponding return (and vice versa):

DEFINITION 2.9 (SAME-LEVEL REALIZABLE PATH). A path $p = n_1, n_2, ..., n_k$ is a *same-level* realizable path if $p$ is realizable, nodes $n_1$ and $n_k$ are contained by the iCFG assigned to the same procedure, and the call stack of $p$ is empty at $n_k$.

CFGs can be extended to model data elements:

DEFINITION 2.10 (VARIABLE DEFINITION). A node $n$ in the CFG *defines* a program variable $v$ if a value is assigned to $v$ at statement corresponding to $n$.

DEFINITION 2.11 (VARIABLE USE). A node $n$ in the CFG *uses* a program variable $v$ if the value of variable $v$ is referenced at statement corresponding to $n$.

DEFINITION 2.12 (INFLUENCE). The definition of variable $u$ is *influenced* by a use of variable $v$ in node $n$ in the CFG if $v$ is used in $n$, $u$ is defined in $n$, and the value assigned to $u$ is dependent on the value of the referenced variable $v$.

A variable definition in a node $n$ *affects* another node $m$ if $m$ can be reached by a path from $n$ in the CFG which contains no (re-)definition for the variable defined at $n$. Such a path is called a *definition-clear path*:

DEFINITION 2.13 (DEFINITION-CLEAR PATH). A path $p$ in the CFG is a *definition-clear path* with respect to variable $v$ if none of the nodes on $p$ (excluding start and end nodes) contain definition for $v$.

DEFINITION 2.14 (DEFINITION-USE PAIR). The definition of variable *v* in node *n* and the use of *v* in node *m* form a *definition-use pair* (*du pair*) if there is a definition-clear path with respect to *v* from *n* to *m*.

Du pairs represent *direct data dependences* between program statements.

Conditional statements (such as `if`, `for`, `while`, `switch`, etc.) introduce a different kind of dependence between program statements, called *control dependence*. As the outcome of a *predicate* (the logical expression representing the condition of a conditional statement) determines the program branch to be executed, it has direct impact on the execution (or not execution) of the statements contained by the conditionally executed branches. Prior to defining control dependence, we need to define (intraprocedural) *postdomination* relationship between control flow graph nodes:

DEFINITION 2.15 (POST DOMINATION). A node *m* (*strictly*) *postdominates* a node *n* in the CFG if every path from *n* to $n_x$ contains *m*, and $n \neq m$.

DEFINITION 2.16 (CONTROL DEPENDENCE). A node *m* is *control dependent* on node *n* in the CFG if (1) there is a path *p* from *n* to *m* such that every node *m'* on *p* (excluding *n* and *m*) is postdominated by *n*, and (2) *m* is not postdominated by *n*.

We note that there are different notions of control dependence in the literature. The definition above is the most widely used, and considered to be the "standard", representing *direct* control dependences. Programs that contain infinite loops (e.g., event listeners in reactive programs) or procedures with multiple or no exit nodes (where the unique end node property cannot be guaranteed [Venkatesh et al. 2007]) may require alternative definitions, such as *weak control dependence* [Podgurski and Clarke 1990]. The transitivity of control dependences – which is not captured by present definition – is however considered in another way when computing *closure slices* (described later). In many applications of slicing, such as debugging or program understanding, having slices that preserve termination behaviour is less important than having smaller slices. Since these applications are on focus in this thesis, the classical definition is appropriate for our purpose. We also note that conditionally executed explicit halt statements (`abort`, `exit`, `halt`, `STOP RUN`, etc.) may necessitate introducing additional interprocedural control

dependences (other than control dependences due to procedure calls), which would make the presentation of the idea much more complicated; hence, it is omitted in this work. Methods and algorithms for computing control dependences can be found in [Loyall and Mathisen 1993; Harrold et al. 1998; Ranganath et al. 2007].

We will assume that intraprocedural control dependences are computed in advance and are represented in the CFG by *control edges*: there is a control edge from *n* to *m* if *m* is control dependent on *n*. In Figure 3, there are control edges between nodes *a4* and *a5*, and *a4* and *a7*. Interprocedural control dependences due to control dependent procedure calls are represented by introducing control edges from call sites to the entry node of the called procedures, and from entry nodes to all the nodes in the procedure (except entry, exit, and return sites), respectively. In Figure 3, an interprocedural control edge has been added between nodes *a5* and *b1*, and one intraprocedural control edge between nodes *b1* and *b2*. (We omit interprocedural control edges where call sites are not control dependent, and control edges from entry nodes where the entry node is not control dependent.)

Direct and indirect effects between statements can be defined as a transitive flow of data and control dependences:

DEFINITION 2.17 (DEPENDENCE CHAIN). A *dependence chain* is a sequence of nodes $n_1$, $n_2$, ..., $n_k$, where each node $n_{i+1}$ is either directly data or control dependent on node $n_i$ for $i = 1, 2, \ldots, k\text{-}1$.

Nodes $n_2$, $n_3$, ..., $n_k$ are said to be *affected* by node $n_1$, which corresponds to the concept of *syntactic dependence* of Podgurski and Clarke [1990]. Nodes $n_1$, $n_2$, ..., $n_k$ of the dependence chain are referred to as *chain nodes*. A dependence chain containing data dependences only is called a *definition-use chain* (*du chain*).

For simplicity of the presentation and without loss of generality, we assume one definition per node such that it is influenced by all the (potential) uses in that node. Statements corresponding to complex instructions (which may contain more than one variable assignment) can be represented by multiple CFG nodes, each containing a single definition and zero or more uses corresponding to the influencing variable uses.

Similarly to paths, not all dependence chains are *realizable*. A dependence chain is realizable if it can be *covered* by a realizable path.

DEFINITION 2.18 (COVERAGE PATH). A path *p covers* a dependence chain $n_1, n_2, ..., n_k$ if it goes through chain nodes $n_1, n_2, ..., n_k$, and each subpath $p_i$ 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 $p_i$ are control dependent on $n_i$ (control dependence), respectively, for $i$ = 1, 2, …, $k$-1.

DEFINITION 2.19 (REALIZABLE DEPENDENCE CHAIN). A dependence chain is *realizable* if it can be covered by a realizable path.

A slicing criterion specifies a program point and a set of program variables:

DEFINITION 2.20 (SLICING CRITERION). The *slicing criterion* is a pair $C = <I, V>$, where $I$ is a program statement and $V$ is a subset of program variables.

Though Weiser's original definition allows selecting arbitrary set of program variables at program point $I$, $V$ is typically a subset of program variables used at $I$, or a single variable used at $I$, respectively. As program statements correspond to nodes in the control flow graph, by "program point" $I$ we will also refer to a node in the CFG.

The backward static program slice $S$ with respect to slicing criterion $C=<I, V>$ consists of all the statements of the program that have direct or indirect effect on the values computed for variables $V$ at $I$. The forward static program slice with respect to slicing criterion $C=<I, V>$ consists of all the statements that depend on the definitions made to program variable(s) $V$ at $I$. As program instructions can directly be related to CFG nodes, program slices can be defined as the set of chain nodes of the possible dependence chains in the CFG that end (backward slicing), or start (forward slicing) at the node of the slicing criterion, respectively:

DEFINITION 2.21 (BACKWARD STATIC SLICE). The *backward static slice S* of a program with respect to slicing criterion $C=<I, V>$ is a set of nodes in the CFG such that for each node $n$ in $S$ there exist a dependence chain from $n$ to the node corresponding to $I$.

DEFINITION 2.22 (FORWARD STATIC SLICE). The *forward static slice S* of a program with respect to slicing criterion $C=<I, V>$ is a set of nodes in the CFG such that for each node $n$ in $S$ there exist a dependence chain from the node corresponding to $I$ to $n$.

The slicing method is called *precise* – up to realizable program paths – if the slice is computed upon realizable dependence chains. According to Weiser's original definition a slice is also executable that can be compiled and run. Similarly to other recent approaches, we compute *closure slices* [Binkley 1993] that contain all components that might affect a given computation but are not necessarily executable.

## 2.2 Forward Data-flow Slicing

Data-flow slicing is a reduction of full slicing which considers data dependences only. A forward data-flow slice consists of the set of chain nodes of all the possible definition-use chains that start from the node of the slicing criterion.

We assume that we are given the control flow graph of the program and a slicing criterion that consists of a node and a single variable defined at that node. We first also assume that the program contains global and scalar variables only, and there is no parameter passing between procedures (other than via global variables).

We note that the assumption that the variable of the slicing criterion is defined at the node of the slicing criterion is not a restriction, but makes the presentation simpler. We also note that the algorithm provides safe results[4] in the case of arrays and records as well by treating them as a whole (conservative approach).

The basic idea of the method is to explore definition-use chains by propagating *tokens* over the control flow graph. A token is sort of reaching definition information (in forward slicing) associated with a definition, or definitions of the same variable, respectively. The token propagation starts from the node of the slicing criterion with a token created for the initial definition, which is propagated to successor nodes iteratively along definition-clear paths with respect to the defined variable. Those nodes that are reached by the token and contain use of the defined variable are marked as “in the slice” (definition-use pairs). Definitions influenced by these uses induce new token propagations from the affected nodes to explore indirect dependences (definition-use chains).

In order to propagate tokens along definition-clear paths tokens carry the identifier of the defined variable denoted as the index of the token, referred to as *token index*. For slicing criterion $C=<n, \{x\}>$ (definition of variable $x$ at node $n$) a token $RD_x$ (Reaching Definition) is created and propagated to the successor node of $n$. Nodes that contain

[4] Results are referred to as “safe” if all possible dependences are taken into account, though some of the dependences might not occur during real program execution.

definition (redefinition[5]) of the variable corresponding the token index block the propagation, otherwise the token is propagated through. Nodes that contain use of the token index are marked as in the slice, and if this use influences a definition of a variable *z*, a new propagation starts at that node with token $RD_z$. This is similar to the conventional reaching definition computation [Hecht 1977] apart from that instead of using bit-vectors, we treat reaching definitions belonging to different variables separately.

Without context information the propagation would traverse all possible program paths, including non-realizable ones. To avoid it we introduce *backtrack index* into tokens to control propagations from procedure exit nodes. The backtrack index is also a variable identifier, denoted as upper index in tokens. The token created for the slicing criterion is initialized with a special Ø backtrack index (no context). Backtrack indices of tokens remain unchanged during the intraprocedural propagation. New tokens created for influenced definitions get the backtrack index of the token affecting the use.

Tokens entering the entry node of a called procedure store their token index as backtrack index, whereas tokens leaving procedure exit nodes are forced to return to those callers only that contain a *matching* token: a token having token index identical with the backtrack index of the token to be returned. The backtrack index of the token on return is "restored" to the backtrack index of the original token stored in the call site; if the call site contains several matching tokens (identical token indices), the token is returned as multiple tokens with backtrack indices corresponding to the different backtrack indices. Tokens having Ø backtrack index are propagated to all return sites unchanged.

[5] Any definition made to a *scalar* variable *redefines* its former value, and so it breaks the effect of any previous definitions made to that variable. In the case of array variables, where a definition made to one array element does not invalidate the effect of definitions made to other array elements, definitions are not considered as redefinitions. Therefore, in the presence of arrays, the distinction between terms *definition* and *redefinition* is important.

The token propagation method can be summarized by the following rules below:

Rule 0. A token $RD_x^o$ 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, a 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)*[6]. Token $RD_z^o$ is propagated from an exit node to all return sites unchanged.

A given $RD_x^y$ token can be propagated to a given node once, therefore a token is propagated over intra- or interprocedural loops at most once. The token propagation stops when no more propagation is possible. Reaching this fixed point, nodes of the slice are marked as in the slice as a result. Note the difference from maintaining the call stack explicitly: the presented method propagates one token for a given variable to a called procedure (e.g., $RD_x^x$), thus it avoids the re-analysis of procedures multiple times for different call stacks.

EXAMPLE 2.1. In the example shown in Figure 3 the token propagation starts with token $RD_x^o$ from slicing criterion node *a2*, which is marked as in the slice (Rule 0). $RD_x^o$ is propagated to nodes *a3*, *a4* (*a4* marked as in the slice), *a5*, *a7*, and *a8* (Rules 1, 2). $RD_x^x$ is propagated to entry node *b1* from call site *a5* (Rule 3). Tokens $RD_x^x$ and $RD_z^x$ (created at *b2*, marked as in the slice) are propagated to exit node *b3*; tokens $RD_x^o$, $RD_z^o$ are propagated to return site *a6* (Rule 4).

[6] Note that it is just the same whether an $RD_z^x$ reaches the exit or an $RD_x^y$ reaches the call site node prior to the other.

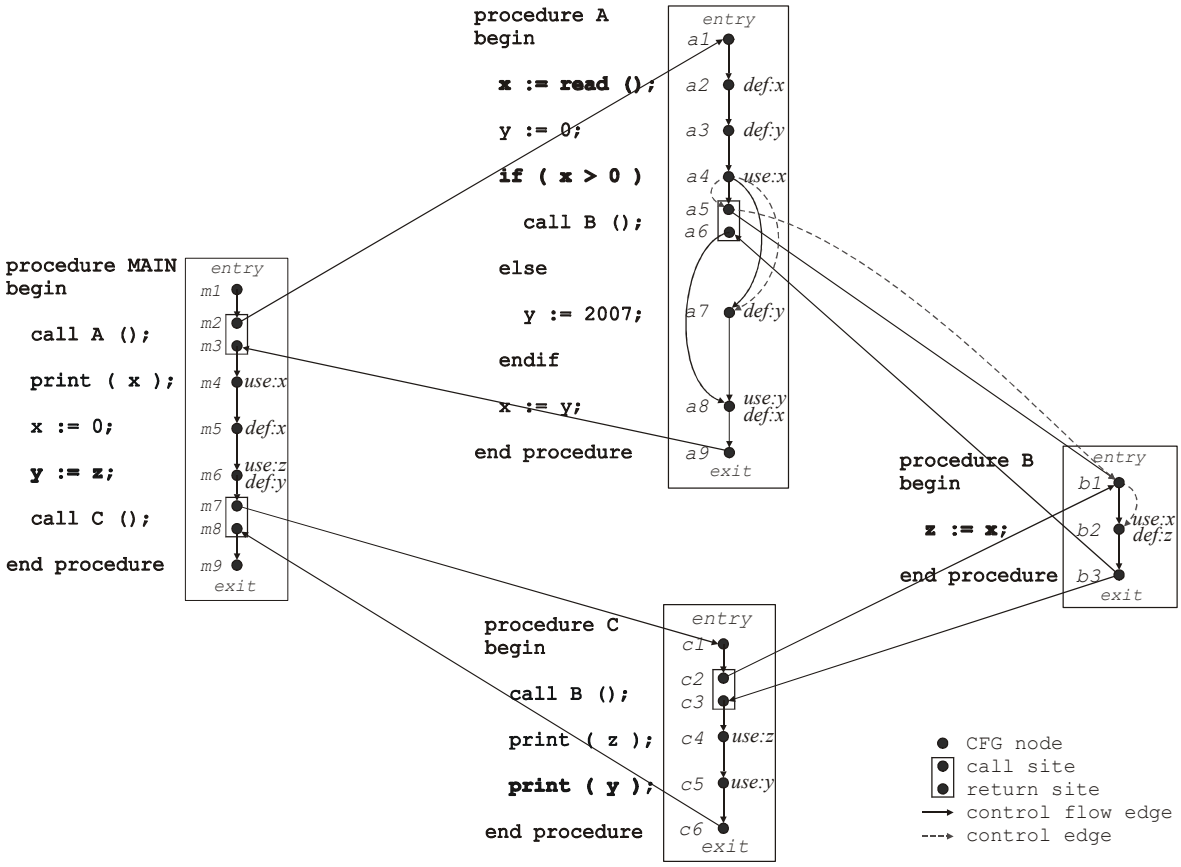



**Fig. 3. Example program graph**

Variable *x* is defined at node *a8*, therefore $RD_x^o$ is not propagated through *a8*. $RD_z^o$ is propagated to exit node *a9* and nodes *m4*, *m5*, *m6* (marked as in the slice), and *m7*. From call site *m7*, $RD_z^z$ and $RD_y^y$ (started by Rule 2 from node *m6*, marked as in the slice) are propagated to entry node *c1*, and from *c2* to *b1*. Variable *z* is defined at *b2*, therefore only $RD_y^y$ is propagated back to return site *c3* and node *c5* (marked as in the slice). $RD_y^y$ is returned to the `MAIN` procedure, and the token propagation stops. As a result, nodes *a2*, *a4*, *b2*, *m6*, and *c5* are marked as in the slice (highlighted in boldface characters in Figure 3).

Note that the algorithm ensures context-sensitive propagation. Thus $RD_z^x$ from exit node *b3* is not propagated to *c3* (because it does not contain a matching token) and *c4* is (correctly) not included in the slice that a token propagation along non-realizable path would have caused.

Intuitively, the idea of the method can be interpreted as the combination of two conventional techniques: the *intra*procedural token propagation is similar to the classical reaching definition computation over CFGs [Hecht 1977], however, we treat definitions of

different variables by different tokens instead of using bit-vectors. *Inter*procedural token propagation can be related to the use of summary edges in the SDGs: a token $RD_z^x$ propagated to a procedure exit node represents (and is directly equivalent with) procedure summary edge $x \rightarrow z$. The backward propagation of a token $RD_z^x$ to call site $c$ containing $RD_x^y$ and the continuation of the propagation with token $RD_z^y$ from return site *returnSiteOf(c)*, can be interpreted as the propagation of $RD_x^y$ from $c$ to *returnSiteOf(c)* considering the transitive dependence $x \rightarrow z$ due to call (reflected in the token index). This is similar to how *path edges* are extended at call sites in [Reps et al. 1994], but we compute summary edges on-demand and perform token propagation over CFGs.

The pseudo-code of the forward data-flow slicing algorithm is shown in Figure 4 in lines 5–41 (excluding lines 36–38). Algorithm `ComputeSlice` has two inputs: the interprocedural control flow graph $G$ and the slicing criterion $C=(s, v)$, where $s$ is a node in $G$ and $v$ is a program variable. The result of the algorithm is the set of nodes in $G$ that are marked as in the slice. The token propagation is implemented using a *worklist* algorithm, where worklist elements are pairs containing a token and a node from where the token is to be propagated. The auxiliary procedure `Propagate` is used to store a token at a given node, and to add this token (and the node) to the worklist if this token has not been yet contained by that node. The initialization of the worklist is performed in lines 5 and 6 (Rule 0); the intraprocedural token propagation is described in lines 26–35 (Rules 1 and 2); the forward interprocedural token propagation is described in lines 11–15 (Rule 3, 4); and the backward interprocedural token propagation is described in lines 16–25 (Rule 4). Note that Rule 4 is to be applied in two cases: when a new token is propagated to an exit node (lines 21–23) and when a new token is propagated to a call site (lines 13–15). It ensures that tokens in return sites are always "synchronized" with tokens in call sites and exit nodes. Lines 13–15 achieve reuse of the previous tokens propagated to the exit node of the called procedure, thus the algorithm avoids reanalysis of a procedure called from different call sites.

```
algorithm ComputeSlice
input   C=(s, v): slicing criterion for node s and variable v
        G:        program graph
output  nodes of the slice are marked in G
global  Worklist: set of pairs (token, node), initially empty

procedure Propagate (RD_x^y : token , n : node)
begin
[1]  if RD_x^y ∉ n then
[2]    Insert RD_x^y into n
[3]    Insert (RD_x^y , n) into Worklist
[4]  endif
end

begin ComputeSlice
[5]  Insert (RD_v^Ø, s) into Worklist                    // Rule 0
[6]  Mark s
[7]  while Worklist ≠ Ø do
[8]    Select and remove a pair (RD_x^y , n) from Worklist
[9]    if x ≠ C then                                     // data tokens
[10]     switch n

[11]       case call site for called procedure P:        // Rule 3
[12]         Propagate (RD_x^x , entryNodeOf(P))
[13]         foreach z | RD_z^x ∈ exitNodeOf(P) do
[14]           Propagate (RD_z^y , returnSiteOf(n))      // Rule 4
[15]         endfor

[16]       case exit node:                               // Rule 4
[17]         foreach return site r connected with n do
[18]           if y = Ø then
[19]             Propagate (RD_x^Ø, r)
[20]           else
[21]             foreach z | RD_y^z ∈ callSiteOf(r) do
[22]               Propagate (RD_x^z , r)
[23]             endfor
[24]           endif
[25]         endfor

[26]       default:
[27]         foreach successor m of n do
[28]           if m does not redefine x then             // Rule 1
[29]             Propagate (RD_x^y , m)
[30]           endif
[31]           if m uses x then                          // Rule 2
[32]             Mark m
[33]             foreach defined variable z influenced by use of x
                 do
[34]               Propagate (RD_z^y , m)
[35]             endfor
[36]             if m is a predicate then                // Rule 5
[37]               Propagate (RD_C^y , m)
[38]             endif
[39]           endif
[40]         endfor

[41]       endswitch

[42]   else                                              // control tokens
[43]     switch n

[44]       case call site for called procedure P:
[45]         Propagate (RD_C^C , entryNodeOf(P))         // Rule 3
[46]         foreach z | RD_z^C ∈ exitNodeOf(P) do
[47]           Propagate (RD_z^y , returnSiteOf(n))      // Rule 4
[48]         endfor

[49]       case entry node:                              // Rule 8
[50]         foreach node m within the procedure do
[51]           Mark m
[52]           Propagate (RD_C^C , m)
[53]         endfor

[54]       case predicate:                               // Rule 5, 6
[55]         foreach control dependent node m of n do
[56]           Mark m
[57]           Propagate (RD_C^y , m)
[58]         endfor

[59]       default:                                      // Rule 7
[60]         foreach defined variable z do
[61]           Propagate (RD_z^y , m)
[62]         endfor

[63]     endswitch
[64]   endif
[65] endwhile

end ComputeSlice
```

**Fig. 4. Pseudo-code of the forward slicing algorithm**

By applying the token propagation method we obtain correct data-flow slices with respect to realizable program paths. Before the proof, let us make some remarks about the token propagation rules. Except for the very first token propagation rule (Rule 0) every rule is triggered as a consequence of a previously applied rule. Each rule considers a *source token* propagated to some *source node*, and propagates a *target token* to a *target node*. The target node is the successor node of the source node in the CFG. In whatever order we apply the propagation rules, for each token we could reconstruct the actual chain of rules (*rule sequence*) that resulted in that token in that node. What is more, as it will be shown later, for each propagated token there exists a *realizable rule sequence* that caused (or could have been caused) the token to be propagated to that node. This realizable rule sequence corresponds to a series of consecutive edges in the CFG, i.e. a path, which starts from the node of the slicing criterion, and which is realizable. This path serves as a coverage path for a definition-use chain, hence, whenever a token is propagated to a node $m$ that uses the token index and $m$ is marked as in the slice, a *realizable* definition-use chain from the slicing criterion to the marked node $m$ can be presented that proves the correctness of the computed slice.

First we define the necessary concepts and lemmas used during proof of correctness and completeness.

DEFINITION 2.23 (RULE SEQUENCE). Given the CFG $G$ of a program. A *rule sequence* is a chain of consecutive token propagation rules $S=(n_1, RD_1)\rightarrow(n_2, RD_2)\rightarrow\ldots\rightarrow(n_k, RD_k)$ ($k>1$), where $R_i=(n_i, RD_i)\rightarrow(n_{i+1}, RD_{i+1})$ is one of the token propagation rules applied to *source token* $RD_i$ in *source node* $n_i$ ($\in G$) that propagates *target token* $RD_{i+1}$ to *target node* $n_{i+1}$ ($\in G$) for $i = 1, 2, \ldots, k\text{-}1$.

Note that in a rule sequence, the target node and the target token of one rule application correspond to the source node and source token of the next one. Each rule application can be associated with an edge in the CFG: application of Rules 0, 1, or 2 can be associated with an intraprocedural edge, application of Rules 3 or 4 can be associated with call or return edges, respectively. In the case of application of Rule 4, the source token is considered to be the token in the procedure exit node. The series of consecutive edges in the CFG forms a path, called *corresponding path*.

Depending on whether this path is realizable we distinguish *realizable* rule sequences:

DEFINITION 2.24 (REALIZABLE RULE SEQUENCE). Given the CFG $G$ of a program. A rule sequence $S=(n_1, RD_1)\rightarrow(n_2, RD_2) \rightarrow\ldots\rightarrow(n_k, RD_k)$ is *realizable* if path $p=(n_1, n_2, \ldots, n_k)$ is a realizable path in $G$.

In the case of Rules 0, 1, 2, and 3, the target token is determined by considering merely the source token, as well as in the case of Rule 4, when the backtrack index of the source token is Ø. However, when the backtrack index of the source token is not Ø, Rule 4 also takes a *matching* token in the selected call site into consideration – to determine the backtrack index of the target token to be returned. These tokens are not explicit (or even included) in a rule sequence defined above. To make them self-contained we define the notion of *coherent rule sequence*. A coherent rule sequence is a realizable rule sequence in which at each application of Rule 4 where the backtrack index of the source token is not Ø, the considered token corresponds to the one propagated in the rule sequence from the call site matching the return site corresponding to the target node:

DEFINITION 2.25 (COHERENT RULE SEQUENCE). A rule sequence $S$ is *coherent* if path $p$ corresponding to $S$ is realizable and for each application of Rule 4 of the form $(n_x, RD_z^x)\rightarrow(r, RD_z^y)$, where $x\neq\emptyset$, there is a call site $c$ matching return site $r$ on $p$ such that Rule 3 applied at $c$ is of the form $(c, RD_x^y)\rightarrow(n_e, RD_x^x)$. ($n_e$ is the entry node and $n_x$ is the exit node of the same procedure called by $c$.)

Note that a subsequence of a coherent rule sequence is also coherent if its corresponding path is a same-level realizable path, since on this path every call has the corresponding return.

As intraprocedural token propagation rules do not change the backtrack index, and the backtrack index value is restored on return, tokens propagated in coherent rule sequences have identical backtrack indices within the “same calling context”, which is formalized by the following lemma:

LEMMA 2.1. Given the CFG *G* of a program, a coherent rule sequence *S,* and path *p* in *G* corresponding to *S*. Considering any two nodes *n*, *m* on *p* such that the subpath of *p* from *n* to *m* is a same-level realizable path, the backtrack index of the token propagated from *n* and the backtrack index of the token propagated to *m* in *S* are identical.

PROOF . Let us consider subpath *p'* of *p* between nodes *n* and *m*, and subsequence *S'* of *S* corresponding to *p'*. Since *p'* is a same-level realizable path, *S'* is coherent.

If *p'* does not contain any procedure calls, *S'* consists of a series of intraprocedural token propagation rules. Since in applications of Rules 0, 1, and 2 the backtrack index of the target token is identical with the backtrack index of the source token, the backtrack index of the token propagated to *m* must be identical with the backtrack index of the token propagated from *n*.

If *p'* contains exactly one procedure call, i.e., there is one call site *c* and one return site *r* matching *c*, the backtrack index of the token propagated from *c* must be identical with the backtrack index of the token propagated to *r* (Definition 2.25). Since *S'* contains intraprocedural token propagation rules between *n* and *c*, and *r* and *m*, the backtrack index of the token propagated from *n* and to *m* must be identical as well.

If *p'* contains a sequence of procedure calls and returns, the identity of backtrack indices holds for all (outermost) call and return site pairs. As backtrack indices are identical in the intervening intraprocedural sections between calls, the identity of backtrack indices of tokens propagated from *n* and to *m* holds in this case as well.

□

Regarding the presented token propagation method, we distinguish relevant rule sequences that start with Rule 0, i.e. $R_1=(s, RD_v^{\varnothing})\rightarrow(m, RD_v^{\varnothing})$, where *s* is the node of the slicing criterion, *v* is the variable of the slicing criterion, and *m* is the successor node of *s*. Such rule sequences are referred to as *full coherent rule sequences*. Note that Rule 0 is the only applicable rule at the beginning of the slice computation (without having the source token actually being propagated to the source node), and is to be applied once.

In a coherent rule sequence *S*, if the backtrack index of the source token is not Ø at any procedure exit node $n_x$, the call stack of path *p* corresponding to *S* cannot be empty (a necessary and satisfactory condition to have a call site matching the target return site, Definition 2.25). The following lemma shows that it also holds for any node in a full

coherent rule sequence other than procedure exit nodes; and conversely, Ø backtrack index indicates empty call stack at any node of $p$:

LEMMA 2.2. Given the CFG $G$ of a program, a full coherent rule sequence $S$, and path $p$ in $G$ corresponding to $S$. Considering a source (or target) token $RD_x^y$ propagated from (or to) a node $n$ in $S$, $y$ is Ø if and only if the call stack of $p$ is empty at $n$.

PROOF. The very first rule application of the full coherent rule sequence $S$ is the application of Rule 0 with source node $s$ corresponding to the slicing criterion and source token $RD_v^o$ where $v$ is the variable of the slicing criterion. The call stack is empty at $s$, and the backtrack index of the source token is Ø, thus the lemma holds for the very first node of $p$. In the following we show that the lemma holds for all target nodes in $S$.

The *necessary* condition requires showing that if the call stack of $p$ is empty at some target node, the backtrack index of the target token propagated to that node is Ø. It is proved by contradiction. Assume that there is a rule application in $S$ such that the call stack of $p$ is empty at the target node but the backtrack index of the target token is not Ø. If there is one or more such rule applications in $S$, there must be a first one. Let us denote it by $R$.

$R$ cannot be the application of Rule 3, because the call stack of $p$ is not empty at a procedure entry node following a call site, and $R$ cannot be not the very first rule application of $S$, since the backtrack index of the target token is Ø in the application of Rule 0.

If $R$ is the application of an intraprocedural rule (Rule 1 or 2), the backtrack indices of source and target tokens must be identical: non-Ø, as assumed; as well as the call stack of $p$ at source and target nodes: empty. Considering the rule application $R'$ in $S$ directly preceding $R$, we can see that the call stack of $p$ is empty at the target node of $R'$ (source node of $R$), and the backtrack index of the target token of $R'$ is non-Ø (source token of $R$). It contradicts the initial assumption that $R$ is the first such a rule application.

$R$ cannot be the application of Rule 4 to a source token having Ø backtrack index, because, according to the rule definition, the backtrack index of the target token is also Ø (contradicts the assumption).

If $R$ is the application of Rule 4 to a source token having non-Ø backtrack index applied at some procedure exit node $n_x$, which propagates its target token to target return site $r$, since $S$ is coherent, according to Definition 2.25, there must a call site $c$ matching $r$ on $p$ such that the backtrack index of the source token of Rule 3 applied at $c$ and the backtrack

index of the target token propagated by Rule 4 to *r* are identical. As assumed, the backtrack index of the target token of *R* (Rule 4) is not Ø, therefore the backtrack index of the source token of Rule 3 applied at *c* must be non-Ø either. As the call stack of *p* is identical at *c* and *r*, and it is empty at *r* (assumed), it is empty at *c* too. Considering the rule application *R'* in *S* directly preceding the application of Rule 3 at *c*, we can see that the call stack of *p* is empty at the target node of *R'* (*c*), moreover, the backtrack index of the target token of *R'* (source token of Rule 3) is non-Ø. As *R'* precedes *R* in *S*, *R* is not the first such rule application as assumed, which is a contradiction again.

The *sufficient* condition requires showing that if the call stack of *p* is non-empty at some target node, the backtrack index of the token propagated to that node cannot be Ø. The proof is by contradiction again. Assume that there is a rule application in *S* such that the call stack of *p* is not empty at the target node but the backtrack index of the target token is Ø. If there is such rule application, there is a first one, let us denote it by *R*.

*R* cannot be the application of Rule 3, since the backtrack index of the target token propagated to a procedure entry node is identical with the token index of the source token in the call site, which cannot be Ø. (Token indices always correspond to variable identifiers, set by Rule 0 or 2, and are unchanged by other propagation rules.)

If *R* is the application of an intraprocedural rule (Rule 1 or 2), backtrack indices of target and source tokens are identical (Ø, as assumed), as well as the call stack of *p* at its source and target nodes (not empty, as assumed). Considering the rule application *R'* in *S* directly preceding *R*, we can see that the call stack of *p* is not empty at the target node of *R'* (source node of *R*), and the backtrack index of the target token of *R'* is Ø (source token of *R*). It contradicts the initial assumption that *R* is the first such rule application.

It is assumed that the call stack of *p* is not empty at the target node. If *R* is the application of Rule 4 (return edge) and the call stack of *p* is not empty at the return site (target node), the call stack of *p* cannot be empty at the preceding procedure exit node (source node) either. (If the call stack is non-empty after return, the call stack contained at least two elements before.)

If *R* is the application of Rule 4 to a source token having Ø backtrack index, hence, in the rule application *R'* in *S* directly preceding *R*, the call stack of *p* is not empty at the target node of *R'* (source node of *R*), and the backtrack index of the target token of *R'* is Ø (source token of *R*). So, the assumption that *R* is the first such rule application does not hold.

If $R$ is the application of Rule 4 to a source token having non-Ø backtrack index applied at some procedure exit node $n_x$, according to Definition 2.25, there must a call site $c$ matching $r$ on $p$ such that the backtrack index of the source token of Rule 3 applied at $c$ and the backtrack index of the target token propagated by Rule 4 to $r$ are identical, i.e., Ø, as assumed. The call stack of $p$ is not empty at $c$ either, which is the same as at $r$ (assumed to be non-empty). Hence, in the rule application $R'$ in $S$ directly preceding the application of Rule 3 at $c$, the call stack is not empty at the target node of $R'$ ($c$) and the backtrack index of the target token of $R'$ is Ø, which contradicts the initial assumption that $R$ is the first such rule application.

□

The lemma implies that if the call stack of the path $p$ corresponding to a full coherent rule sequence $S$ is not empty at some node $n$ on $p$, the backtrack index of the token propagated to (or from) $n$ cannot be Ø.

Full coherent rule sequences can be related to realizable definition-use chains starting from the slicing criterion. It is shown by the following lemma below:

LEMMA 2.3. Given the CFG $G$ of a program, a slicing criterion $C=<s, \{v\}>$, a full coherent rule sequence $S$, and path $p$ in $G$ corresponding to $S$. If the variable corresponding to the token index of the target token of the last rule application in $S$ is used at target node $n$, there exists a realizable definition-use chain in $G$ from $s$ to $n$.

PROOF . The very first rule application in $S$ is the application of Rule 0 with source node $s$ corresponding to the node of the slicing criterion, and source and target token $RD^{\varnothing}_{v}$ where $v$ is the variable of the slicing criterion.

First assume that $S$ does not contain any application of Rule 2. In this case, $S$ consists of the application of Rule 0 followed by zero or more applications of Rules 1, 3, or 4. (If $S$ is composed solely of the application of Rule 0, i.e., the variable defined at $s$ is used in its successor node $n$, ($s$, $n$) is a realizable definition-use chain; thus the lemma holds.) Since token indices of source and target tokens are identical in applications of Rules 1, 3, 4, token indices of all the tokens propagated in $S$ are identical and equal to $v$. Neither call sites nor procedure exit nodes contain variable definition (Rule 3 and 4), and none of the source nodes of the applications of Rule 1 can contain a definition for the token index of the source token. Therefore $p$ is a definition-clear path wrt. $v$. Since variable $v$ is used in

target node $n$ of the last rule application of $S$, ($s$, $n$) is a definition-use chain. $p$ is realizable, since $S$ is coherent, hence ($s$, $n$) is a realizable definition-use chain, so the lemma holds.

If $S$ contains one or more applications of Rule 2, let us consider the node sequence ($n_1$, $n_2$, …, $n_k$), where $n_1 = s$, $n_k = n$, and nodes $n_i$ ($1<i<k$) are the source nodes of the applications of Rule 2 in $S$ (in the order they occur). Let us denote the subsequence of $S$ between nodes $n_i$ and $n_{i+1}$ by $S_i$ ($1 \le i < k$). The first rule application in subsequence $S_1$ is the application of Rule 0; while in any other subsequence $S_i$ ($i \neq 1$) the first rule application is the application of Rule 2. In either case, the token index of the target token of the first rule application corresponds to the variable defined at its source node ($n_i$), and this rule application is followed by zero or more applications of Rules 1, 3, or 4. For the same reasons described above, token indices of target tokens in $S_i$ are equal and identical to the variable defined at $n_i$, and the path corresponding to $S_i$ is definition-clear. The variable defined at node $n_i$ is used at node $n_{i+1}$, because either at node $n_{i+1}$ $S$ contains an application of Rule 2 (which implies a use in node $n_{i+1}$ for the token index corresponding to the variable defined in node $n_i$), or $n_{i+1}$ is $n$, which contains a use for token index of the target token as assumed by the lemma. Hence, node sequence ($n_1$, $n_2$, …, $n_k$) is a definition-use chain from $s$ to $n$ covered by path $p$. Since $p$ is realizable, it is a realizable definition-use chain, so the lemma holds.

□

The following lemma shows the reverse direction: realizable definition-use chains can be associated with full coherent rule sequences.

LEMMA 2.4. Given the CFG $G$ of a program and a slicing criterion $C=<s, \{v\}>$. If there is a realizable definition-use chain in $G$ from slicing criterion node $s$ to a node $n$, there exists a full coherent rule sequence $S$ such that the variable corresponding to the token index of the target token of the last rule application of $S$ is used in its target node $n$.

PROOF. To prove the lemma we show that a full coherent rule sequence can be constructed for any realizable definition-use chain ($n_1$, $n_2$, …, $n_k$) ($k>1$), where $n_1 = s$ and $n_k = n$. Let us consider the realizable coverage path $p$ of the definition-use chain ($n_1$, $n_2$, …, $n_k$) (such a path must exist according to Definition 2.19), which is composed of subpaths $p_1$, $p_2$, …, $p_{k-1}$, where subpath $p_i$ ($1 \le i < k$) is a path in $G$ from $n_i$ to $n_{i+1}$, and it is definition-clear wrt. the variable defined at $n_i$ (Definition 2.18). First, we show that a rule sequence $S$ can be

constructed for $p$ by assigning one token propagation rule to each edge of $p$; then, we prove that $S$ is a full coherent rule sequence.

Let us consider the first subpath $p_1$ of $p$, and assign the application of Rule 0: $(n_{1_1}, RD_v^{\varnothing}) \rightarrow (n_{1_2}, RD_v^{\varnothing})$ to the first edge of $p_1$, where $n_{1_1}$ is the node of the slicing criterion $(s)$, $v$ is the variable of the slicing criterion, and $n_{1_2}$ is the second node on subpath $p_1$. The source token and the source node of the following rule application to be assigned are given by the target token and the target node of the preceding rule application, and depending on the edge type, we assign one of the rule applications to the next edge $e = (n, m)$ on $p_1$ below:

(1) Rule 1: $(n, RD_x^y) \rightarrow (m, RD_x^y)$ if $e$ is an *intraprocedural* edge,

(2) Rule 3: $(n, RD_x^y) \rightarrow (m, RD_x^x)$ if $e$ is a *call edge*,

(3) Rule 4: $(n, RD_x^{\varnothing}) \rightarrow (m, RD_x^{\varnothing})$ if $e$ is a *return edge* and the backtrack index of the source token is Ø,

(4) Rule 4: $(n, RD_x^y) \rightarrow (m, RD_x^z)$ if $e$ is a *return edge* and the backtrack index of the source token is not Ø, where $z$ is the backtrack index of the source token propagated in $S$ from the call site matching return site $m$.

We apply the above assignments for subsequent edges of $p_1$, iteratively, until to every edge of $p_1$ a rule application has been assigned. Note that case (4) assumes a rule application assigned previously to a call edge matching the current return edge; as it will be shown later, such a rule application must exist. The target node of the last rule application assigned to the last edge of $p_1$ is $n_2$, which is the first node of the next subpath of $p_2$.

Let us consider the following subpath $p_i$ ($i$ = 2, 3, …, $k$-1), and assign the application of Rule 2: $(n_{i_1}, RD_x^y) \rightarrow (n_{i_2}, RD_z^y)$ to the first edge of $p_i$, where $z$ is the variable defined at $n_{i_1}$ $(n_i)$, and source token $RD_x^y$ is given by the target token of the last rule application assigned to the last edge of the preceding subpath $(p_{i\text{-}1})$. For subsequent edges of $p_i$ we apply the same assignments (1)–(4) as on $p_1$.

The above procedure is continued as long as to every edge of $p$ the appropriate rule application has been assigned, i.e. the complete rule sequence $S$ has been constructed.

The applications of rules 0 and 2 (assigned to the first edges of the subpaths) as well as the applications of rules assigned in cases (2) and (3) are *valid*, they fulfill to the related rule definitions. The application of Rule 1 assigned in (1) is also valid, because subpath $p_i$ ($1 \leq i < k$) is definition-clear wrt. the variable defined at $n_i$ that corresponds to the token index of the propagated source token. Rule assignment in case (4) can be performed only if (*a*) there is a call site *c* on *p* matching target return site *m*, and is valid if (*b*) the token index of the token propagated from *c* corresponds the backtrack index of the source token at *n*. Note that if both conditions hold at each assignment (4), rule sequence *S* is coherent, as applications of Rule 4 in *S* fulfill Definition 2.25. Also note that *S* is a full coherent rule sequence, as it starts with Rule 0.

To prove that conditions (*a*) and (*b*) hold at each assignment (4) assume that, by contradiction, there is a return edge $e = (n, m)$ on *p* such that either ($\neg a$) there is no call site *c* on *p* matching return site *m*, or ($\neg b$) the token index of the source token in the rule application assigned to the call edge from *c* is not identical with the backtrack index of the token to be propagated from *n*. Let *e* be the first such return edge on *p*, where the construction of the rule sequence fails. Let us denote by *S'* the rule sequence successfully constructed for subpath *p'* of *p* from $n_1$ up to *n*. Note that *p'* is realizable (as *p* is realizable), and *S'* is a full coherent rule sequence. Since the backtrack index of the token propagated to *n* is not Ø (because case (4) is applicable at *e*), according to Lemma 2.2, the call stack of *p'* cannot be empty at *n*, so there must exist a call site *c* on *p'* matching return site *m*, which contradicts ($\neg a$). According to Lemma 2.1, since *S'* is coherent and the subpath of *p'* between procedure entry node $n_e$ following *c* on *p'* and procedure exit node *n* is a same-level realizable path, the backtrack index of the token propagated from $n_e$ must have the same backtrack index as the backtrack index of the token propagated to *n*. According to Rule 3, the backtrack index of the token propagated to $n_e$ is identical with the token index of the token propagated from *c*, hence, the token index of the token propagated from *c* must be identical with the backtrack index of the token propagated to *n*, which contradicts ($\neg b$). Since there cannot be such return edge along *p*, where conditions (a) or (b) fail, the constructed rule sequence *S* is a full coherent rule sequence.

The token index of the target token propagated in the last rule application of *S* to target node $n_k$ corresponds to the variable defined at node $n_{k-1}$, which is used in target node $n_k = n$ (definition-use chain), thus the lemma holds.

□

We say that a token $RD_x^y$ propagated to a node $m$ is *reachable* by a full coherent rule sequence if there exists a full coherent rule sequence $S$ in which the target node and target token of the last rule application are $m$ and $RD_x^y$, respectively. In the following we show that in whatever order we apply the token propagation rules, each propagated token is reachable by a full coherent rule sequence.

LEMMA 2.5. Given the CFG $G$ of a program and a slicing criterion $C=<s, \{v\}>$. If we apply the token propagation rules for $C$ over $G$, any token $RD_x^y$ propagated to a node $m$ is reachable by a full coherent rule sequence.

PROOF. We use induction to prove the lemma. First we show that the lemma holds for the token propagated by the very first propagation rule; then we show that if the lemma holds for all the previously propagated tokens, it will also hold for any token propagated by a subsequent token propagation rule.

*Base Case*: Token $RD_v^o$ propagated by Rule 0 to a node $m$, where $v$ is the variable of the slicing criterion and $m$ is the successor node of the slicing criterion node $s$, is reachable by a full coherent rule sequence.

Rule sequence $(s, RD_v^o) \rightarrow (m, RD_v^o)$ is a full coherent rule sequence, because its first rule application is the application of Rule 0, path $(s, m)$ is a realizable path in $G$, and it contains no applications of Rule 4. The target node and the target token of the last rule application is $m$ and $RD_v^o$, thus token $RD_v^o$ propagated by Rule 0 to $m$ is reachable by a full coherent rule sequence, hence, the base case holds. Note that Rule 0 is the first and only applicable rule at the beginning of any token propagation.

*Inductive Step:* If all the previously propagated tokens are reachable by a full coherent rule sequence and we apply any of the relevant propagation rules $R$ to a token $RD_z^x$ propagated to a node $n$, the token propagated by $R$ to a node $m$ is also reachable by a full coherent rule sequence.

If $R$ is one of the applications of Rules 1, 2, 3, or $R$ is the application of Rule 4 to a source token having Ø backtrack index, respectively, let us consider the full coherent rule

sequence $S$ that reaches $RD_z^x$ propagated to $n$ (induction hypothesis). Let us denote by $p$ the realizable path corresponding to $S$, and by $p'$ the path corresponding to the extended rule sequence $S'$ obtained by appending $R$ to $S$. In the case of Rules 1, 2 or 3, $p'$ is given by the concatenation of a realizable path (that is $p$) and an intraprocedural edge corresponding to $R$, which also results in a realizable path. In the case of Rule 4, $p'$ is given by concatenating a return edge corresponding to $R$ to $p$. From Lemma 2.2 it follows that, since the backtrack index of the token propagated in $S$ from $n$ is Ø, the call stack of $p$ is empty at $n$. As at empty call stack, appending return edge to a realizable path results in a realizable path, $p'$ is realizable, and so is $S'$. As $S$ is a full coherent sequence, and no new application of Rule 4 has been added by $R$ to $S$ to a source token having non-Ø backtrack index, $S'$ is also a full coherent rule sequence. The target node and the target token of the last rule application of $S'$ are $m$ and $RD_z^x$, therefore the token propagated by $R$ to $m$ is reachable by a full coherent rule sequence, so the inductive step holds.

If $R$ is the application of Rule 4 to a source token having non-Ø backtrack index, i.e. $R$ is of the form $(n, RD_z^x) \rightarrow (m, RD_z^y)$, where $x \neq Ø$, the definition of Rule 4 implies that call site $c = callSiteOf(m)$ already contains a token $RD_x^y$, and $n$ contains a token $RD_z^x$. Let us denote the full coherent rule sequence that reaches $RD_z^x$ propagated to $n$ by $S_n$, and the full coherent rule sequence that reaches $RD_x^y$ propagated to $c$ (induction hypothesis) by $S_c$. From Lemma 2.2 it follows that, as the backtrack index of the token propagated in $S_n$ to $n$ is not Ø, the call stack of $p_n$ is not empty at $n$. Therefore, there must be a call site on $p_n$ such that the subpath $p_n'$ of $p_n$ between the procedure entry node $n_e$ following call site $c$ and procedure exit node $n$ is a same-level realizable path, so subsequence $S_n'$ of $S_n$ corresponding to $p_n'$ is coherent; furthermore, from Lemma 2.1 it follows that, as a token having backtrack index $x$ ($RD_z^x$) is propagated to $n$, the source token in the first rule application of $S_n'$ is $RD_x^x$. Let us consider the rule sequence $S'$ obtained by concatenating rule sequences: $S_c$, $(c, RD_x^y) \rightarrow (n_e, RD_x^x)$, $S_n'$, $(n, RD_z^x) \rightarrow (m, RD_z^y)$. Path $p'$ corresponding to $S'$ is realizable, because the subpath of $p'$ between nodes $n_e$ and $n$ ($p_n'$) is a same-level realizable path, as well as the subpath of $p'$ between nodes $c$ and $m$, which is appended to the realizable path corresponding to $S_c$. $S'$ is coherent, because $S_c$ and $S_n'$ are coherent, and for the application of Rule 4, $(n, RD_z^x) \rightarrow (m, RD_z^y)$, the matching application of Rule 3, $(c, RD_x^y) \rightarrow (n_e, RD_x^x)$, can be found in $S'$ at the call site $c$ matching $m$ (Definition 2.25). $S'$ starts with the application of Rule 0 ($S_c$), and the target node and the target token of the last

rule application of *S'* are *m* and $RD_z^y$. Hence, the token propagated by *R* to *m* is reachable by a full coherent rule sequence, and the inductive step holds.

□

After defining the necessary concepts and the related lemmas we can turn to the proof of correctness and completeness. To prove correctness of the slice computed by the token propagation method we show that each node marked during the token propagation is affected by the slicing criterion; completeness requires showing that every affected node will be marked during the token propagation.

THEOREM 2.6 (CORRECTNESS OF THE DATA-FLOW SLICE). Given the CFG *G* of a program and a slicing criterion $C=<s, \{v\}>$. A node *n* is marked as in the slice by the token propagation rules applied for *C* over *G* only if $n = s$ or there exists a realizable definition-use chain in *G* from *s* to *n*.

PROOF. The two propagation rules that mark a node as in the slice are Rules 0 and 2. By definition, the slicing criterion is included in the resulting slice. The node marked by Rule 0 corresponds to the node of the slicing criterion *s*, which fulfills the first condition of the theorem. A node *n* is marked by applying Rule 2 only if a token, say $RD_{x_0}^y$ has been propagated to *n* such that *n* contains a use of variable *x*. If such a token is propagated to *n*, from Lemma 2.5 it follows that *n* is reachable by a full coherent rule sequence *S*. As *S* reaches *n*, the target token of the last rule application of *S* is $RD_x^y$, whose token index used in target node *n*. Hence, considering *S*, from Lemma 2.3 it follows that there exist a realizable definition-use chain in *G* from *s* to *n*, which proves the theorem.

□

The token propagation stops if no more token propagation is possible, because either none of the rules applicable to any of the previously propagated tokens or the token that would be propagated by any of the relevant rules had already been propagated to the target node by some other rule before. The following theorem shows that when the token propagation stops, all affected nodes are marked by the token propagation rules.

THEOREM 2.7 (COMPLETENESS OF THE DATA-FLOW SLICE). Given the CFG $G$ of a program and a slicing criterion $C=<s, \{v\}>$. If there exists a realizable definition-use chain from slicing criterion node $s$ to a node $n$ in $G$, or $n = s$, respectively, $n$ is marked as in the slice by the propagation rules applied to $C$ over $G$.

PROOF. The node of the slicing criterion $s$ is marked by the very first rule application of the token propagation, by Rule 0, which fulfills the second implication ($n = s$) of the theorem.

The first implication requires showing that if there exists a realizable definition-use chain from $s$ to $n$ in $G$, $n$ is marked before no more token propagation is possible. From Lemma 2.4 it follows that there is a full coherent rule sequence $S$ such that the variable corresponding to the token index of the target token of the last rule application in $S$, say $RD_x^y$, is used in its target node $n$. If $RD_x^y$ is propagated to $n$, $n$ is marked by Rule 2 fulfilling the theorem. Now we show that $RD_x^y$ must be propagated to $n$ before the token propagation stops, what is more, every target token in $S$ must be propagated to the target node by the end of the token propagation.

Assume that by contradiction there is a rule application in $S$ whose target token has not been propagated to its target node but the token propagation stops. If there is such a rule application in $S$, there is a first one; let us denote it by $R$. $R$ cannot be the very first rule application of $S$, as Rule 0 is applicable at the beginning of the token propagation, and so the propagation cannot stop yet. Since $R$ is the first such rule application, the source token of $R$ has been propagated to its source node. $R$ is a valid and so an applicable propagation rule that would result in a new token in its target node, which contradicts the assumption that the token propagation can already stop.

□

We note that we assume finite-size programs – of practical relevance – having finite-size CFGs and a finite number of program variables. In this case, as there can only be a finite number of possible tokens propagated to a finite number of nodes, the token propagation method terminates in finite steps. Note that it holds even if there are an infinite number of program paths, or definition-use chains, respectively, due to potential loops.

## 2.3 Forward Slicing

The token propagation can be extended to accommodate control dependences by introducing *control tokens*. Control tokens are created at predicate nodes and propagated

along control edges. Nodes reached by control tokens are marked as in the slice; definitions in control dependent nodes start new token propagations to reveal indirect dependences.

In *control tokens*, a special token index: *C* is used to distinguish from *data tokens* where token indices are variable identifiers[7]. When a data token $RD_x^y$ is propagated to a predicate node that uses variable *x* a new control token $RD_C^y$ is created and propagated to the nodes that are control dependent on the predicate. Each node to which $RD_C^y$ is propagated is marked as in the slice; and if this node contains a definition of a variable *z*, a new propagation starts with data token $RD_z^y$.

Data tokens created due to interprocedural control dependences are to be returned to the call site from where the control dependence originates. For this reason, backtrack index *C* is stored in control tokens entering called procedures, and data tokens having backtrack index *C* are returned to the call site(s) containing control token. From procedure entry nodes control tokens are propagated to all the nodes within the procedure (except entry, exit, and return sites). Note that the same rules, Rule 3 and 4 (described in the previous section), can be applied to propagate control tokens to called procedures, or to return data tokens having backtrack index *C*, respectively.

The rules related to the control token propagation are summarized 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*.[8]

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, exit, and return sites).

[7] Note that though control tokens are not related to *reaching definitions*, because of the similarity of the propagation, using the same notation *RD* simplifies the presentation and the pseudo-code.

[8] This rule serves as propagating control tokens inside nested predicates. Depending on the definition and computation of control dependences (whether it considers transitivity or not) this rule may or may not be required.

Intraprocedurally, control tokens thus simply transfer the backtrack index of the token affecting the predicate to data tokens created at control dependent definition nodes that ensures context-sensitive propagation of the newly created tokens. Interprocedurally, backtrack index *C* assigned to data tokens created in the called procedures due to control dependences ensures that these tokens are returned to the call site(s) from where the control dependence originates (containing control token), which is analogous to the backward propagation of data tokens having variable identifiers as backtrack index.

EXAMPLE 2.2. In the example shown in Figure 3, a new control token $RD_C^o$ is created when data token $RD_x^o$ is being inserted into predicate node *a4*, and $RD_C^o$ is propagated to nodes *a5* and *a7* along control edges (Rule 5). Nodes *a5* and *a7* are marked as in the slice, and a new data token $RD_y^o$ is created at node *a7* (Rule 7). From call site *a5*, control token $RD_C^c$ is propagated to entry node *b1* (Rule 3) and node *b2* (Rule 8). Node *b2* is marked as in the slice, and a new data token $RD_z^c$ is created and propagated to exit node *b3*. $RD_z^o$ is propagated to return site *a6* (Rule 4). $RD_y^o$ , which is created at *a7*, and $RD_z^o$ , which is returned to *a6*, are propagated in the following steps according to the data token propagation rules (Rules 1–4). Node *a8* is marked as in the slice by $RD_y^o$, where a new data token $RD_x^o$ is created. Tokens $RD_x^o$, $RD_y^o$, and $RD_z^o$ are propagated to return site *m3*. $RD_x^o$ is propagated to nodes *m4* (marked as in the slice) and *m5*, where variable *x* is defined (hence, the propagation of this token stops). $RD_y^o$ is propagated to nodes *m4*, *m5*, and *m6*, where variable *y* is defined. The propagation steps of token $RD_z^o$ from return site *m3* have been described in the previous section. In the case of full slicing, nodes *a5*, *a7*, *a8*, and *m4* are marked as in the slice, in addition to the nodes added by data-flow slicing.

Considering control dependences, slicing is extended to explore dependence chains. The pseudo-code related to the propagation of control tokens is shown in Figure 4 in lines 36–38 and 42–65. Control tokens are propagated from predicates to control dependent nodes in lines 36–38, 54–58 (Rule 5, 6); data tokens are created at control dependent nodes in lines 59–62 (Rule 7); the forward interprocedural control token propagation is described in lines 44–48 (Rule 3, 4); and the propagation of control tokens from entry nodes is described in lines 49–53 (Rule 8). Data tokens having backtrack index *C* are propagated backwards in lines 21–23 (Rule 4). Note that no control tokens can be propagated to exit nodes because there are no control edges to them.

Theorems of correctness and completeness presented for data-flow slicing are valid in the case of full slicing as well: a node is marked by the token propagation rules if and only if there is a dependence chain from the slicing criterion to that node. Since the proof is quite analogous to one presented in the previous section, we discuss the key differences only.

Correctness requires showing that each propagated token can be associated with a rule sequence (Lemma 2.5) composed of rule applications chosen from the extended rule set, which can be associated with a dependence chain (Lemma 2.3). With regard to Lemma 2.5, as propagating control tokens along control edges extends previous rule sequences either intraprocedurally (control edges from predicates or procedure entry nodes) or interprocedurally forwards (control edges from call sites), it does not violate realizability nor coherence. (Definition 2.24 still applies to rule sequences built upon the extended rule set, considering that Rule 4 is also applicable to return tokens with control backtrack index.) The amendment of Lemma 2.3 involves that nodes of the dependence chain covered by the corresponding path are pointed by source nodes of potential applications of: Rule 5 (data-control), Rule 6 (control-control), and Rule 7 (control-data) – in addition to the application of Rule 2 (data-data). As control tokens propagate along control edges to directly or indirectly control dependent nodes, subpaths between chain nodes fulfill Definition 2.18, that is, the corresponding path is a coverage path for the dependence chain.

Completeness requires showing that for any dependence chain an appropriate rule sequence can be constructed. The coverage path of the dependence chain potentially contains control edges, hence, the amendment of Lemma 2.4 involves extending rule application assignments to control edges (Rules 5–8) that can be constructed analogously.

In both cases, Lemmas 2.1 and 2.2 apply to rule sequences based on the extended rule set unchanged, and the token index of the target token of the last rule application of the rule sequence (Lemmas 2.3, 2.4) can be either used in its target node (data dependence), or it is *C* (control dependence).

## 2.4 Backward Slicing

The algorithm of backward slicing can be obtained by reversing the token propagation rules of forward slicing. We refer to tokens as *LV* (Live Variable) tokens in the case of backward slicing. *LV* tokens are propagated to predecessor nodes along definition-clear

paths with respect to the used variable. Nodes reached by *LV* tokens that define the used variable are marked as in the slice; new token propagations start from all uses influencing the definition. Interprocedurally, *LV* tokens are propagated from return sites to exit nodes and from entry nodes to call sites, respectively. Backtrack indices of *LV* tokens are used analogously to forward slicing.

Control tokens are created at each node wherever a new data *LV* token is created (including the slicing criterion node), and propagated along control edges backwards. Control tokens reaching predicate nodes start new *LV* token propagations from all the uses in the predicate. Control tokens reaching procedure entry nodes are propagated to call sites, and from call sites to other predicates, or entry nodes, respectively, on which this node is control dependent.

The pseudo-code of the backward slicing algorithm is omitted, since it is quite similar to the forward one. The propagation rules of *LV* tokens are shown below:

Rule 0. A token $LV_x^o$ is created for slicing criterion $C=<n, \{x\}>$, which is propagated to the predecessor node(s) of *n*. Node *n* is marked as in the slice.

Rule 1. If a token $LV_x^y$ is propagated to a node *n* that does not define variable *x*, the token is propagated to the predecessor node(s) of *n* unchanged.

Rule 2. If a token $LV_x^y$ is propagated to a node *n* that defines variable *x*, *n* is marked as in the slice. A new token $LV_z^y$ is created for use of variable *z* influencing definition *x*, which is propagated to the predecessor node(s) of *n*.

Rule 3. If a token $LV_x^y$ is propagated to a return site *r*, token $LV_x^x$ is propagated to the exit node of the called procedure.

Rule 4. Any return site *r* that contains a token $LV_x^y$ and entry node *e* (of procedure called by *callSiteOf(r)*) that contains a token $LV_z^x$ induce the propagation of token $LV_z^y$ from call site *callSiteOf(r)*. Token $LV_z^o$ is propagated from an entry node to all call sites unchanged.

Rule 5. If a token $LV_C^y$ is propagated to predicate node *p*, *p* is marked as in the slice. A new token $LV_z^y$ is created for use of variable *z* in *p*, which is propagated to the predecessor node(s) of *p*.

Rule 6. If a token $LV_x^y$ is propagated to a node $n$ that defines variable $x$, a new token $LV_C^y$ is created and propagated to nodes on which $n$ is control dependent. For slicing criterion $C=<n, \{x\}>$ a token $LV_C^o$ is created and propagated to nodes on which $n$ is control dependent.

Rule 7. If a token $LV_C^y$ is propagated to a call site $c$, token $LV_C^y$ is propagated to nodes on which $c$ is control dependent.

Note the asymmetry in forward and backward slicing (Rules 6 and 7) which is due to that control edges are not symmetric in the different directions: entry nodes typically have more outgoing control edges, whereas exit nodes have no incoming control edge; predicates have more outgoing control edges, whereas nodes have typically one incoming control edge. Rule 6 and 7 also ensure the propagation of control tokens to entry nodes (which is performed by Rule 8 in the forward direction), or to the predicates of the enclosing conditional statement, respectively.

## 2.5 Local Variables, Parameter Passing

This section discusses how the presented token propagation method can be extended to local variables and parameter passing. Keywords and constructs mentioned below basically derive from COBOL (which was the programming language motivated this work), but similar concepts can be found in other programming languages. We restrict to describing the forward case (the backward case is analogous).

We consider COBOL program systems where programs contain "local" variables and use parameter passing at external program calls. In the following we describe how the propagation rules can be extended to this case.

*Local variables.* COBOL applications typically consist of several programs that call each other. Each program can be represented by a CFG, called a *program graph*, where *program call sites* and *program return sites* (`CALL` statements) are linked to the entry and exit nodes of the called program's main procedure. We refer to such set of program graphs as a *program system graph*. Variables declared in one program are not accessible in other programs (unless they are explicitly passed by reference), therefore from program call sites tokens are propagated directly to the related program return sites when the index variable of the token is not passed (or passed by value, respectively), as these variables cannot be redefined (*killed*) during the call. From program exit nodes, only tokens related to formal

parameters passed by reference are propagated back to calling programs; definitions made to other variables have no effect after return.

*Parameter passing.* There are two standard ways of passing parameters between programs: *call-by-value* and *call-by-reference*. In the first case, the value of the actual parameter is passed to the formal parameter; in the latter case, its memory reference[9] is passed, thus any modification of the formal parameter is reflected back the passed actual parameter. Parameter passing requires a conversion of token indices – from actual to formal parameter, and vice versa – during the inter-program token propagation. A token $RD_a^y$ from a program call site is propagated to the entry node of the called program as token $RD_f^f$ if actual parameter *a* is passed as formal parameter *f* (either by value or by reference). A token $RD_g^f$ from a program exit node is propagated to program call site *c* only if *c* contains a token having token index *a* (say $RD_a^y$), where *a* is the actual parameter passed as formal parameter *f*. On return, the token index (*g*, which is a formal parameter passed by reference) is converted back to the matching actual parameter (say *b*), and the backtrack index is restored correspondingly to the backtrack index of the token stored in the call site (i.e., $RD_b^y$ is propagated to the program return site).

The rules below complete the rule set of forward slicing (described in Sections 2.1 and 2.2) for the case of program-local variables and parameter passing:

Rule 3.b If a token $RD_x^y$ is propagated to a program call site *c*, where actual parameter *x* is passed as formal parameter *f*, token $RD_f^f$ is propagated to the entry node of the called program's main procedure. If *x* is not passed by reference, token $RD_x^y$ is propagated to program return site *returnSiteOf(c)*.

Rule 4.b Any program call site *c* that contains a token $RD_x^y$ and exit node *e* of the called program's main procedure that contains a token $RD_g^f$ induce the propagation of token $RD_z^y$ to program return site *returnSiteOf(c)* if actual parameter *x* is passed as formal parameter *f*, and actual parameter *z* is passed as formal parameter *g* by reference. Token $RD_g^o$ is propagated from *e* to *returnSiteOf(c)* as token $RD_z^o$.

[9] COBOL (before COBOL-97) supports no pointer type variables. The variable reference cannot be accessed explicitly or modified. The same reference is passed on a potential subsequent pass-by-reference.

These rules can be adapted analogously for programs containing procedure-local variables. Token indices and backtrack indices related to global variables (as well as value *C*, respectively, representing control dependence) require no token index conversion during the interprocedural token propagation.

## 2.6 Related Work

Various algorithms for calculating interprocedural slices exist. The first method published by Weiser [1984] is not context-sensitive. There are studies [Agrawal and Guo 2001; Krinke 2002; Binkley and Harman 2003; Krinke 2006] 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. Therefore, we concentrate on precise slicing methods.

Most of the methods are based on system dependence graphs published first by Horwitz et al. [1990]. System dependence graphs can be considered as the whole program extension of the program dependence graph [Ottenstein and Ottenstein 1984; Ferrante et al. 1987]. Using SDGs, slicing is reduced to a graph reachability problem. The key element of the approach is the computation of transitive dependences due to procedure calls (summary edges).

Reps et al. [1994] introduced an improved summary edge computation algorithm to speed up slicing using graph reachability between formal parameter vertices. Considering COBOL programs, where practically all the variables are global, the cost of Reps' summary edge computation technique is bounded by $O(E_{SDG} \times V + TotalSites \times V^3)$, where $E_{SDG}$ is the number of edges in the SDG, *TotalSites* is the total number of call sites, and $V$ is the number of (global) variables in the program. This is followed by a two-pass traversal of the SDG to calculate the slice that requires linear time in the size of the SDG.

The cost of the presented full slicing algorithm is bounded by $O(E_{CFG} \times V^2 + TotalSites \times V^3)$, where $E_{CFG}$ is the number of edges in the program graph. In the worst case, every possible token is propagated along every CFG edge, and from exit nodes every token is propagated as $V$ different tokens to return sites. Although at our COBOL systems $E_{CFG}$ was much less than $E_{SDG}$ (by at least two orders of magnitude), the $V$ multiplier in the first term is due to the token propagation used to reveal du pairs (for

each backtrack index[10]), whereas Reps' method exploits a priori flow edges between nodes that however necessitates the construction of the SDG. We note that Reps' summary edges are also derived by our method, on demand: a token $RD_x^y$ in the exit node corresponds to procedure summary $y \rightarrow x$.

As mentioned earlier, the key difference in our algorithm (in addition to exhaustive versus on-demand nature) is that SDGs are huge monolithic graphs that usually exceed the internal memory in the case of real-world applications. It is very difficult to predict which part of the graph should be kept in the main memory. The presented algorithm has the potential to process one program (or one procedure) at a time, resulting in a limited number of time-consuming read and write operations.

Agrawal and Guo [2001] 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 [2002] showed that this algorithm has flaws, and presented a corrected explicitly context-sensitive algorithm. The approach however proved to be impractical to calculate precise program slices due to combinatorial explosion of the set of the potential call stacks.

Livadas and Croll [1992] introduced parse-tree-based SDGs, and considered aliasing, global and static variables. Sinha et al. [1999] extended the SDG method for programs with arbitrary interprocedural control flow, which allows a more precise analysis of program codes containing `stop run`, `exit`, `try-throw-catch`, and similar instructions.

Atkinson and Griswold [1996] also reported that the application of the SDG for larger systems may require prohibitive space and time. They used CFGs and the invocation graph approach [Emami et al. 1994] for context-sensitive slicing. However, that method is exponential in cost at unbounded context-depth. Mock et al. [2002] limited the considered context-depth to two, as they were not able to compute fully context-sensitive slices in a reasonable time. Liang and Harrold [1999] proposed a precise slice computation method that is also based on data-flow information propagation over the CFG. Their algorithm runs in polynomial time, but its actual complexity is not clear, as no analysis is given.

Slicing is a demand problem, and though some of the previously discussed methods are demand-driven to some extent ([Atkinson and Griswold 1996; Liang and Harrold 1999;

[10] The very first token propagation (for any backtrack index) reveals all the intraprocedural du pairs of a given node. The addition of explicit flow edges (that could be reused by subsequent propagations) would however increase the size of the program graph.

Agrawal and Guo 2001]), the most wide spread algorithm based on the SDG is exhaustive. Reps [1993] presented a general demand version of context-sensitive interprocedural analysis problems. Application of his magic-sets method to interprocedural slicing called valid path algorithm was presented. Though the complexity of the algorithm has not been explicitly written, considering the experimental results it seems that the computation time for one slice element significantly increases with the size of the slice. On the contrary, in the presented method, the computation time of one slice element is independent from the resulting slice size. Horwitz et al. [1995] and Reps et al. [1995] have converted a large class of data-flow analysis problems to a special kind of graph reachability problem using *exploded supergraphs*. The construction of the exploded supergraph, in which flow functions are represented explicitly at nodes, however, requires substantial time and space. Duesterwald et al. [1997] proposed a general framework for demand-driven data-flow analysis using fixed-point computation over the CFG. They also yield polynomial-time algorithms, but the efficiency of the approach to solve data-flow analysis problems (other than slicing) was shown only on moderate size programs. Orso et al. [2001] published an incremental slicing method based on data-dependence types using SDGs. They compute summary edges on demand, but that algorithm is not applicable to recursive programs (the recursively called `ComputeSummaryEdges` function potentially gets into infinite loop). The presented slicing technique is demand-driven and applicable to recursive procedures (programs) as well due its fixed-point computation (the token propagation is continued as long as new token can be propagated).

Hajnal and Forgács [2002] proposed a token propagation-based method to compute realizable definition-use chains. That method does not consider control dependences, parameter passing, and is not fully demand-driven (relies on *kill* sets computed in advance). Furthermore, definition identifiers were used as token indices which can cause procedures to be reanalyzed several times for different definitions of the same variable. Hajnal and Forgács [2012a] presented an improved algorithm to compute precise program slices on demand (presented in this thesis) which avoids these limitations and reanalysis of procedures. By using variable identifiers as token indices, it has become possible to reduce the slice computation time from hours to minutes or seconds.

## 2.7 Conclusions

As described in Chapter 1, existing CFG-based dataflow techniques can have prohibitive time requirements to calculate precise program slices in the presence of recursion, while SDG-based approaches rely on exhaustive analysis that poses space requirements and scalability problems. The technique proposed in this chapter attains the accuracy of the SDG approach at avoiding its space requirements by computing summary edges on-demand.

The presented method is conceptually simple which allows of easy implementation and computes precise slices up to realizable program paths. Scalability is addressed by its demand-driven nature, i.e., the method computes the necessary information with regard to the slice currently being computed. As the technique is based on control flow graphs, it is more easily adaptable to accommodate specific system issues (such as complex instructions, massive use of globals) at moderate space requirements, and less sensitive to program modifications.

# Chapter 3

# Evaluation

COBOL is often thought as an old-fashioned programming language which is of little importance by now. The fact is that several hundred billion lines of COBOL codes are actively used today in almost every major industry; what is more, COBOL's dominance is expected to last over the next ten years. Many of the legacy systems are more than 30–40 years old, whose maintenance is very labor-intensive and costly task. Program slicing is a potentially useful analysis for aiding different maintenance activities, including program comprehension, re- and reverse engineering.

To evaluate the presented slicing approach a prototype of the slicing algorithm has been implemented and evaluated on a large COBOL system, which is in use at a company from the financial domain. During the implementation of the algorithm presented in the previous chapter, practical problems had to be solved such as how to interface with tools capable of providing the necessary input to the slicer, how to represent programming language specific constructs, and how the token propagation method can be implemented efficiently.

To evaluate the performance of the slicing method we selected a considerably large set of test cases randomly, on which both slice computation times (data-flow and full slicing in both directions) and slice sizes were measured. Our objective was to measure how fast slices can be calculated for a given slicing criterion "from scratch" without reusing any results of previous token propagations to assess the usability of the method in interactive contexts.

Section 3.1 provides details about the most important design and implementation decisions. Section 3.2 presents the subject system of the experiments. Empirical results are given in Section 3.3. Section 3.4 discusses related tools; finally, section 3.5 concludes the chapter.

## 3.1 Prototype Implementation

A prototype of the presented slicing algorithm has been implemented in the Java programming language. Though C or C++ potentially outperforms Java, features like automatic garbage collection, exception handling, and strict type safety along with object-oriented design made Java an attractive choice for developing robust codes quickly.

In the following subsections, we describe the most important design and implementation solutions applied during the prototype development.

### Interfaces

The implemented slicer prototype operates over control flow graphs (CFGs), i.e., on an abstract representation of a program. On its own the slicer prototype is not able to parse specific source codes, visualize the resulting slice, or associate control flow graph nodes with the related source code lines or variables, respectively, instead, the slicer was designed to be an individual, programming language independent component which can be connected to any Integrated Development Environment (IDE) in the future – inasmuch as it is capable of constructing the control flow graph of the subject program written in a specific source code language.

To be able to pass input to the slicer, namely, the CFG and the slicing criterion, and return the output, the slice, it was necessary to define proper interfaces. The format of the interface was chosen to be XML. XML is a general-purpose description language, which is widely accepted and standard format. To formalize how CFGs can be described in XML we constructed an XML Schema Definition (XSD). This schema includes all the concepts of the traditional control flow graphs in the proper structure: programs, procedures, nodes, edges, variables, variable definitions and uses. By parsing the XML description of a CFG, the slicer constructs an internal representation of the CFG in the system memory over which the token propagation will be performed.

The other input of the slicer is the slicing criterion. As a slicing criterion defines a program point and a program variable (i.e., a part of the CFG), it was convenient to specify its format in XML too using (basically) the same XSD. Similarly, we applied the same format to the resulting slice.

At analyzing COBOL source codes, it early turned out that traditional control flow graph concepts are not sufficient to capture all COBOL constructs, which necessitated the introduction of new concepts.

COBOL systems typically consist of several programs calling each other; hence, we introduced the concept of *program system* that represents a set of interconnected *program graphs* (CFGs). To represent program calls (`CALL` statements) we introduced new node types, called *program call sites* and *program return sites*, such that program calls accommodate parameter passing in contrast to procedure call and return sites (`PERFORM` statements) where no parameter passing occurs (other than via global variables). In COBOL, the number of actual parameters may differ at different call sites calling the same program, e.g., it may occur that at a program call site, more than one actual parameters are passed to a single formal parameter (of some compound type), while at another place, the number of actual- and formal parameters are equal. For this reason, we introduced the concept of *virtual parameter positions* such that actual parameters influence virtual parameter positions at call sites, and virtual parameter positions influence the corresponding formal parameters of the called program at program entry points, respectively. Using this indirection, it became possible to bind variable number of actual parameters to a fixed number of formal parameters, and vice versa.[11]

The use of compound data (arrays, structures, records) necessitated the distinction of *scalar* and *array* variables. In contrast to definitions made to scalar variables (of primitive data types) where definitions count as *re*definitions, definitions made to array variables are treated differently during the token propagation: the propagation of a token having token index corresponding to a variable of array type is not blocked at a node containing definition for that variable (Rule 1).

For illustration, in Figure 5, the XML fragment of a CFG (a), a slicing criterion (b), and the slice (c) are shown. In Figure 5.a, the XML description of a program named `Program1` is shown, whose main procedure's id is 1 (`mainProcId` attribute). It contains a global variable with id 1 which is of scalar type, and a global variable of array type having id 2. The first formal parameter corresponding to the first virtual parameter position (`position` attribute) is represented by variable 47, whereas the second formal parameter in the second virtual parameter position corresponds to variable 49. Procedure 1 has unique entry and exit nodes (`<entry>` and `<exit>` elements). The successor of the entry node is node 1 (`succ` attribute). Node 1 is a definition node defining variable 1

[11] Actual parameters are associated with virtual parameter positions by the parser, which information is available during parse-time; whereas the slicer handles influences to virtual parameter positions during the token propagation.

(defs attribute). Node 3 is a use-definition node (it contains both uses and defs attributes), where the use of variable 1 influences the definition of variable 2 (influence attribute). Node 2 is a predicate node: it contains use and has control dependent nodes (controls attribute). The entry node controls all the nodes within the procedure (controls attribute) except the exit node. Node 4 is a procedure call site calling procedure 15 (proc attribute); its return site is node 5 (retId attribute), whose successor is node 6 (succ attribute). Node 6 is a program call site calling another program Program2 (prg attribute) and passing variable 17 in the first virtual parameter position (params attribute); its return site is node 7 (retId attribute), whose successor is node 8. The slicing criterion shown in Figure 5.b specifies slicing criterion node: node 1 in procedure 1 in program Program1, and variable 1 defined at that node. Slicing is to be performed in the forward direction (type attribute). The slice shown in Figure 5.c contains two programs: Program1 and Program2. Program1 contains two procedures: 1 and 15 in program Program1. In procedure 1, node 1 is the slicing criterion (by definition included in the slice), nodes 2 and 3 are data-dependent (uses attribute), whereas nodes 3, 4, 5, 6, and 7 are control-dependent (controlDependent attribute). Procedure 15 in Program1, as well as Program2 is *entirely* control-dependent (controlDependent attribute), which implies that all the nodes of these components are in the slice (control-dependent).

```
<prg name="Program1" mainProcId="1">
    <vars>
           <var id="1" type="scalar" />
           <var id="2" type="array" />
           ...
    </vars>
    <params>
           <var id="47" position="1" />
           <var id="49" position="2" />
           ...
    </params>
    <proc id="1">
           <entry id="0" succ="1" controls="1 2 ... 64" />
           <node id="1" succ="2" defs="1" />
           <node id="2" succ="3 8" uses="1" controls="3 4 5 6 7" />
           <node id="3" succ="4" uses="1" defs="2" influence="1 2" />
           <proccall id="4" proc="15" retId="5" succ="6" />
           <prgcall id="6" prg="Program2" retId="7" succ="8" params="17 1" />
           ...
           <exit id="65" />
    </proc>
    ...
    <proc id="15">
           ...
    </proc>
</prg>
```

(a) XML fragment of a program's CFG

```
<slicingcriterion name="Program1" type="forward">
    <prg name="Program1">
           <proc id="1">
                  <node id="1" defs="1" />
           </proc>
    </prg>
</slicingcriterion>
```

(b) XML fragment of a slicing criterion

```
<slice>
    <prg name="Program1">
           <proc id="1">
                  <node id="1" defs="1" />
                  <node id="2" uses="1" />
                  <node id="3" uses="1" controlDependent="true" />
                  <proccall id="4" retId="5" controlDependent="true" />
                  <prgcall id="6" retId="7" controlDependent="true" />
           </proc>
           <proc id="15" controlDependent="true">
                  ...
           </proc>
    </prg>
    <prg name="Program2" controlDependent="true">
    </prg>
</slice>
```

(c) XML fragment of the slice

**Fig. 5. XML fragment of a CFG, a slicing criterion, and the slice**

### Efficient Token Propagation

The storage how control flow graphs are built in memory as well as the proper algorithmic solutions used to implement the token propagation was highly important regarding the performance. These issues are discussed in the following paragraphs.

Using a single worklist for all the tokens to be propagated as proposed in the pseudo-code in Chapter 2 is inefficient, as this list can grow extremely large, moreover, tokens belonging to different contexts are mixed (frequent context switching might be necessary when only a part of the program system fits into main memory). In the implemented slicer, therefore we used individual worklists, called *procedure token worklists*, in each procedure that contain tokens to be propagated intraprocedurally (propagations through call sites insert elements into the called procedures' token worklists).

After selecting and removing an element from a token worklist (a token and a source node), graph reachability is used within the procedure to determine the nodes reachable from the source node along definition-clear paths. Using graph reachability, we avoided the repeated applications of Rule 1 on the same token. During classical graph reachability successor nodes are *marked* iteratively until each marked node has marked successors only, where marking is usually implemented by setting a boolean *visited* state (attribute) of nodes. To ensure mark propagation along definition-clear paths, we used an additional boolean attribute called *redef* mark to indicate nodes containing (re)definition of the token index of the token currently being propagated. Nodes marked as *redef* block the propagation of *visited* marks. *redef* marking is performed prior to *visited* mark propagation, and all *redef* marks are cleared when the propagation of a given token finishes.

Among the *visited* nodes tokens are inserted into *relevant* nodes only from where new propagation may start. These nodes are the procedure entry and exit nodes, call and return sites, and nodes containing use/definition of the token index. It reduces the memory requirements of the method and avoids re-propagation of tokens from the same node multiple times.

Classical graph reachability requires resetting *visited* attribute in all nodes after each search. To avoid it we used an integer-valued *visited* field in nodes and an integer counter named *currentMarkValue* such that the value of *currentMarkValue* is incremented before each graph search. This value is propagated from the source node, and once the propagation of this value finishes, reached nodes have *visited* attribute equal to

*currentMarkValue*; all other nodes hold a different (some previous) value. *visited* attributes in nodes are reset only when *currentMarkValue* reaches the upper bound of its range.

To perform *redef* marking efficiently we stored nodes associated to uses/definitions in procedures at building the CFG. Using a hash table indexed by variable definitions/uses (as keys) we can rapidly fetch the list of nodes containing definitions or uses for that variable, and mark or clear *redef* marks, respectively.

#### Efficient Token Storage

During slicing several tens or hundreds of millions of token propagations may occur. Selecting a proper storage for tokens is crucial from both time (token insertion and fetching time) and space (memory requirements) points of view. We found no a uniform solution efficient in all nodes but depending on the node type different storage method had been applied. In the case of nodes containing uses, definitions, and return sites, we perform token insertions only (forward slicing), and hence, binary trees are used to achieve fast insertion time and low space requirements.

Token storage at call sites and exit nodes are interrelated: inserting a token into a call site triggers getting all the tokens in the exit node of the called procedure having backtrack index corresponding to the inserted token (forward slicing), while inserting a token into a procedure exit node triggers getting tokens from call sites having token index identical with the backtrack index of the inserted token, respectively (Rule 4). These two operations implied that tokens at call sites are hashed by token indices (backtrack indices are stored in binary trees), while tokens at exit nodes are hashed by backtrack indices (token indices are stored in binary trees, respectively).

Using the storage above, we achieved the performance of several hundred thousand token propagations per second on an average personal computer in 2012.

## 3.2 Subject Programs

COBOL is often thought as an old-fashioned programming language which is of little importance by now. The fact is that COBOL is still the dominant language for business applications, and almost every major industry relies on it. In 1997, it was estimated that as much as 80 percent of the world's computer code ran on COBOL, and there were 240 billion lines of COBOL code in use [Brown 2000]. Although the role of COBOL has been

slightly reduced during the past decades, COBOL's dominance is still expected to last over the next ten years as well [Binkley 2007].

COBOL applications are often very large, many of them consist of more than 1,000,000 lines of code, and even applications over 10,000,000 lines are not considered unusually large. COBOL programs typically deal with enormous volumes of data, and rely on a huge number (possibly tens of thousands) of global variables (`DATA DIVISION`), as the principal program structuring mechanism is the `PERFORM` statement.

Many of the legacy systems are more than 30–40 years old, whose maintenance is very labor-intensive and costly task. The lack of proper documentation, ad-hoc maintenance activities over such long lifetimes, and the poor logical structure of programs can make maintenance very difficult. What is more, there is a huge risk involved in transforming and modernizing such applications, which companies are typically unwilling to undertake. Program slicing is a potentially useful analysis for aiding different maintenance activities, including program comprehension, reverse engineering, debugging, and testing. Hence, the empirical results presented in the following section have an important practical relevance.

The subject of our experiments was a large COBOL system from the financial domain, which consisted of over 8 million lines of code (LOC, including variable declaration part, comments, and empty lines). This large system could be decomposed into independent subsystems of which we investigated five of different sizes (with total of 166 programs and 1.2 million LOC). Table 1 presents details about the subsystems: the total number of programs and procedures, global variables, and program graph nodes. The least subsystem contained 67,000 LOC (*System1* in Table 1), whereas the largest one contained 532,000 LOC (*System5* in Table 1). We were given the *program system* representation of the subsystems, which was constructed using Panorama Analyser[12].

[12] Panorama Analyser is a commercial COBOL analyzer tool. Currently this product is not available.

**Table 1. Details of the investigated systems**

| | #Programs | #Procedures | #Global variables | #Program graph nodes |
|---|---|---|---|---|
| *System1* | 15 | 233 | 14 386 | 11 126 |
| *System2* | 18 | 369 | 41 685 | 18 958 |
| *System3* | 25 | 525 | 52 367 | 25 275 |
| *System4* | 32 | 1955 | 64 884 | 132 085 |
| *System5* | 76 | 3183 | 189 405 | 210 965 |

## 3.3 Empirical Results

We carried out our experiments on a P4 3GHz PC with 2GB RAM under JDK 1.6, using a maximum heap size of 1.5GB (JVM option `Xmx`). For each subsystem we selected 1000 random definitions as slicing criteria for forward slicing, and 1000 random uses for backward slicing. We performed data-flow and full slicing in both directions – thus we computed a total of 4000 slices per subsystem. Our objective was to measure how fast slices can be calculated for a given slicing criterion "from scratch" – without reusing any results of previous token propagations.

We note that the total of 20,000 test cases is only a portion of the possible slicing criteria. Yet, considering that the size of some slices was close to the size of the whole subsystem, we expect no worse results for the rest of the slicing criteria and believe that the preliminary results provide a fair basis for demonstrating the practical applicability of the approach.

The results of the slice computations are summarized in Table 2: execution times and slice sizes. It shows that data-flow slicing gives prompt result for any of the slicing criteria: it takes less than three seconds on average and only 24 seconds in the worst case (with over 2,000 slice elements). The performance of backward slicing was also very good in all the investigated systems: it took around two minutes in the worst case (with over 36,000 slice elements). We also obtained slices quickly at forward slicing in three of the five subsystems: within nine seconds on average and around one minute in the worst case. In spite of the fair average results in the last two systems (less than four minutes), forward slicing can be time consuming in some cases: it took almost 30 minutes in the worst case (although, in 78% of the slicing criteria in *System4* and in 83% of the slicing criteria in *System5* slices were computed in less than one minute). However, when the computation

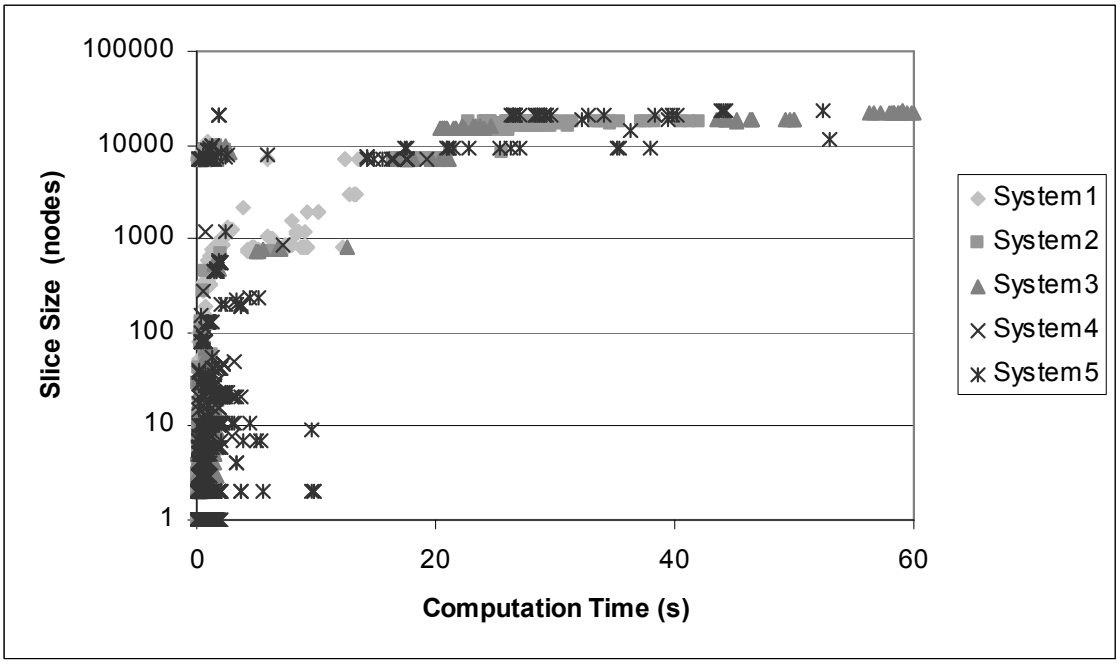
time exceeded 60 seconds, the resulting slices were very large: they contained more than 10,000 nodes – which roughly correspond to the number of affected source code lines. These large slices are less useful for human users, as they are very difficult or impossible to understand, respectively (in practice, slicing can be aborted after one minute). The increase in the slice size was caused by control dependences escalated over the whole subsystem due to recursion (via program calls – in some COBOL versions, recursion between procedures is also possible), which was present in all the five investigated subsystems. The same criteria were used at data-flow slicing, which resulted in much smaller slices.

**Table 2. Execution results of the slicing algorithm on 1000 random slicing criteria**

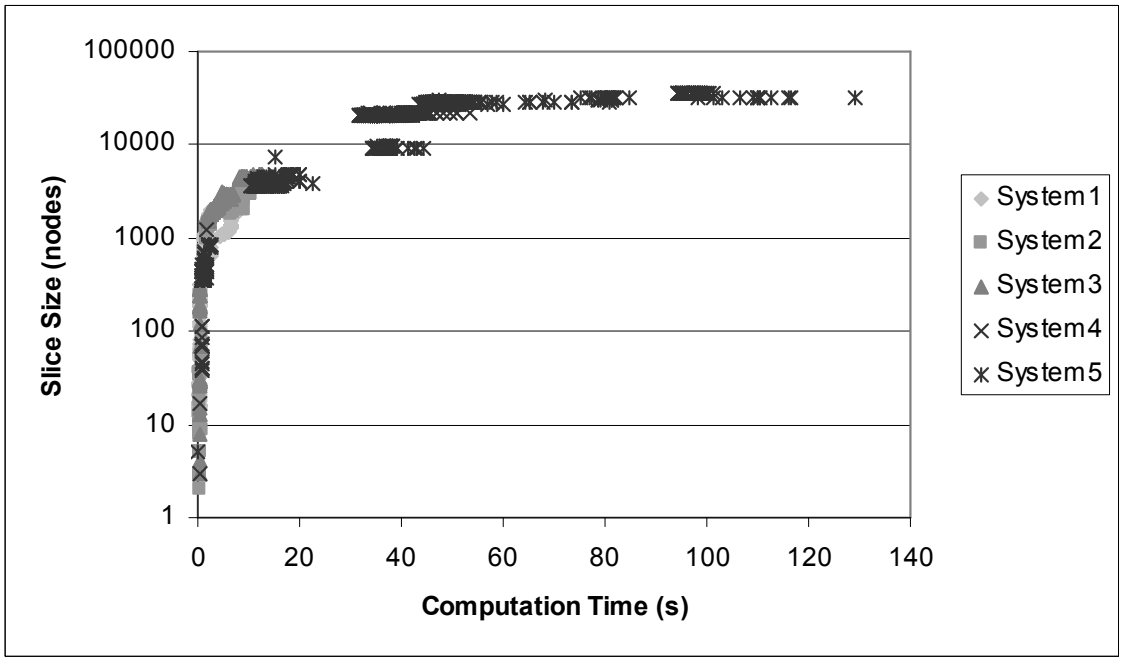
| | **Forward data-flow slicing** | | **Backward data-flow slicing** | | **Forward slicing** | | **Backward slicing** | |
|---|---|---|---|---|---|---|---|---|
| | **exec. time** | **slice size** | **exec. time** | **slice size** | **exec. time** | **slice size** | **exec. time** | **slice size** |
| *System1* | 0.1<br>0.4<br>2.7 | 1<br>34<br>967 | 0.1<br>0.4<br>1.4 | 1<br>42<br>667 | 0.1<br>1.7<br>21 | 1<br>745<br>11 029 | 0.2<br>1.6<br>7.5 | 13<br>646<br>2143 |
| *System2* | 0.1<br>0.7<br>4.4 | 1<br>43<br>1102 | 0.1<br>0.7<br>2.2 | 1<br>89<br>813 | 0.1<br>4.0<br>42 | 1<br>2897<br>17 930 | 0.4<br>8.8<br>12.8 | 2<br>2953<br>4306 |
| *System3* | 0.1<br>0.6<br>3.0 | 1<br>18<br>773 | 0.1<br>0.7<br>2.3 | 1<br>31<br>698 | 0.1<br>5.5<br>69 | 1<br>2353<br>24 193 | 0.4<br>5.0<br>12 | 3<br>2801<br>4751 |
| *System4* | 0.1<br>2.5<br>18.4 | 1<br>150<br>1907 | 0.1<br>1.2<br>9.6 | 1<br>127<br>4143 | 0.1<br>185<br>994 | 1<br>28 613<br>131 424 | 0.6<br>36<br>53 | 3<br>20 470<br>22 228 |
| *System5* | 0.1<br>1.8<br>24 | 1<br>149<br>2835 | 0.1<br>1.4<br>17 | 1<br>162<br>4052 | 0.1<br>230<br>1793 | 1<br>26 001<br>187 879 | 0.1<br>35<br>129 | 5<br>16 046<br>36 024 |

Execution times are given in seconds, slice sizes are given in number of nodes. Minimum/average/maximum values are shown.

Figure 6 presents slice computation times and slice sizes (on a logarithmic scale) in the case of full slicing. Figure 6.b shows the slicing results for all the backward slicing criteria. Figure 6.a shows the results for those forward slicing criteria for which the computation time was less than 60 seconds (beyond 60 seconds, slice sizes always exceeded 10,000 nodes).




(a) Forward slicing




(b) Backward slicing

**Fig. 6. Slice sizes and computation times (full slicing)**

We can see that when the computation time took more than 20 seconds, slices contained more than 1,000 slice elements; above one minute, slices were over 10,000 nodes. This characteristic makes the approach suitable for applications in interactive contexts, because if the computation would take more than one minute, the size of the slice will likely be too big to be evaluated by a human user. Nevertheless, in most cases the resulting slices were small and could be computed quickly.

The average number of the required token propagations was 6,579 at data-flow slicing in our least subsystem, and 9,872 in the biggest one. The average value varied between 148,788 and 13 million at full slicing. In the worst case, we had to perform 98 million token propagations. These values indicate the total number of tokens that need to be stored in CFG nodes, that is, the memory requirements of the slicing algorithm. In order to get a quantitative picture about the number of summary edges determined (required at computing a single slice), we counted the number of tokens propagated to entry and exit nodes. We measured 1,672–7,837 tokens on average propagated to entry/exit nodes at data-flow slicing, and 36,699–969,367 tokens at full slicing. In the worst case, 12.7 million tokens were propagated to the exit nodes.

## 3.4 Comparison with Other Works

The author is aware of only a few papers concerning with slicing COBOL. Ning et al. [1994] presented a toolset called Cobol/SRE (Cobol System Renovation Environment) that supports different reengineering tasks of legacy COBOL systems. Among others, the toolset allows the user to compute forward and backward (and condition-based) slices to help in extracting meaningful business functions. The paper provides no details about the applied method, so it is not clear how precise the computed slices are with regard to realizable program paths. Lanubile and Visaggio [1993] presented a transform slicing method to aid the extraction of reusable functions from ill-structured programs. The slice is obtained by iteratively solving data flow equations based on the program's control flow graph, similarly to Weiser's original method. The approach was demonstrated on an example COBOL program. In [Lanubile and Visaggio 1997], the method was extended to the interprocedural case by maintaining interprocedural walks explicitly. However, as with recording the call stack explicitly, this solution suffers combinatorial explosion in the case of recursion.

There are other types of analysis techniques that can efficiently support maintenance tasks for COBOL [Canfora et al. 1996; Komondoor et al. 2005; Ramalingam et al. 2006]. These techniques concern with recovering and inferring data models and types, but are not directly related to program slicing.

There are a number of empirical studies performed on evaluating context-sensitive slicing [Atkinson and Griswold 1996; Liang and Harrold 1999; Agrawal and Guo 2001; Krinke 2002; Binkley and Harman 2003; Krinke 2006]; however, we found no empirical results on slicing COBOL, neither data on the actual cost of SDG construction for real-world programs.

## 3.5 Conclusions

After having described some of the most important design and implementation solutions of the prototype slicer, the applicability of the novel slicing approach has been evaluated on large-size COBOL systems on a considerably large set of test cases. The empirical results show that the method is capable of computing accurate program slices quickly, whereas longer computation times always result in large slices.

The computation times of data-flow as well as full backward slices were short in all the investigated systems; also, we obtained fair average computation times for full forward slices. In some cases, forward slicing was time consuming, however, in all these cases, the resulting slices were too big to be evaluated by human users: slice sizes exceeded 1,000 slice elements after 20 seconds of computation. This characteristic makes the presented approach suitable for application in interactive context.

# Chapter 4

# Understanding Program Slices

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, larger program slices, but even slices containing only some tens of program instructions can be very difficult to understand. As William Griswold pointed out in his talk: *Making Slicing Practical: The Final Mile* [Griswold 2001], 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. Program slicing, however, beyond claiming that there is dependence between the slicing criterion and the computed slice element gives no explanation of the result that could help in understanding the effects between different parts of the program code by a human user.

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 actual chain of dependences from the slicing criterion to the selected slice element the user could be convinced that the influence indeed exists, and there is an unforeseen, undesired side effect of the modification that 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. Such imprecision, for example, can be due to infeasible program paths (no such program input that results in the execution of the traversed conditional branches) or programming language constructs that make impossible to recover statically the precise flow of data (use of pointers, dynamic constructs).

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 cannot be realized along any feasible path), 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.

Section 4.1 presents a method to provide explanation for the computed slice elements called the “reason-why algorithm”. Section 4.2 discusses the related work; finally, Section 4.3 concludes the chapter.

## 4.1 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. For clarity of the presentation we consider programs containing global and scalar variables. Local variables and parameter passing can be treated as described in Chapter 2.

### 4.1.1 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 presented in Chapter 2. 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 $(n, RD_x^o)$ will be referred to as the *source*; the pair $(m, RD_y^z)$ is referred to as the *target*. We note that we provide a 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 use-definition nodes and 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 token propagated to the predecessor node become the new target. Then, we continue finding such predecessors as long 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 will be correctly nested, i.e., the resulting reason-why path will be realizable.

**Pass 1**

Pass 1 (as well as Pass 2) consists of a series 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 token $RD_y^y$ in the called procedure's exit node). Note that as the backtrack index is not Ø, 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) or (c), we continue searching for the next predecessor of the current target as long 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.

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

We continue the above procedure as long as any of the call sites contains a token $RD_y^o$, 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 7. 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*, whose call site contains a token $RD_y^y$ and the called procedure contains summary edge $y \rightarrow y$ (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) y→y    -- return from B
3. (c2, RD_y^y)        -- call B
4. (c1, RD_y^y)        -- entry C
5. (m7, RD_y^∅)        -- 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 (if $m$ 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 $v \rightarrow y$ (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 Ø 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 of (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 long as we reach $n$.

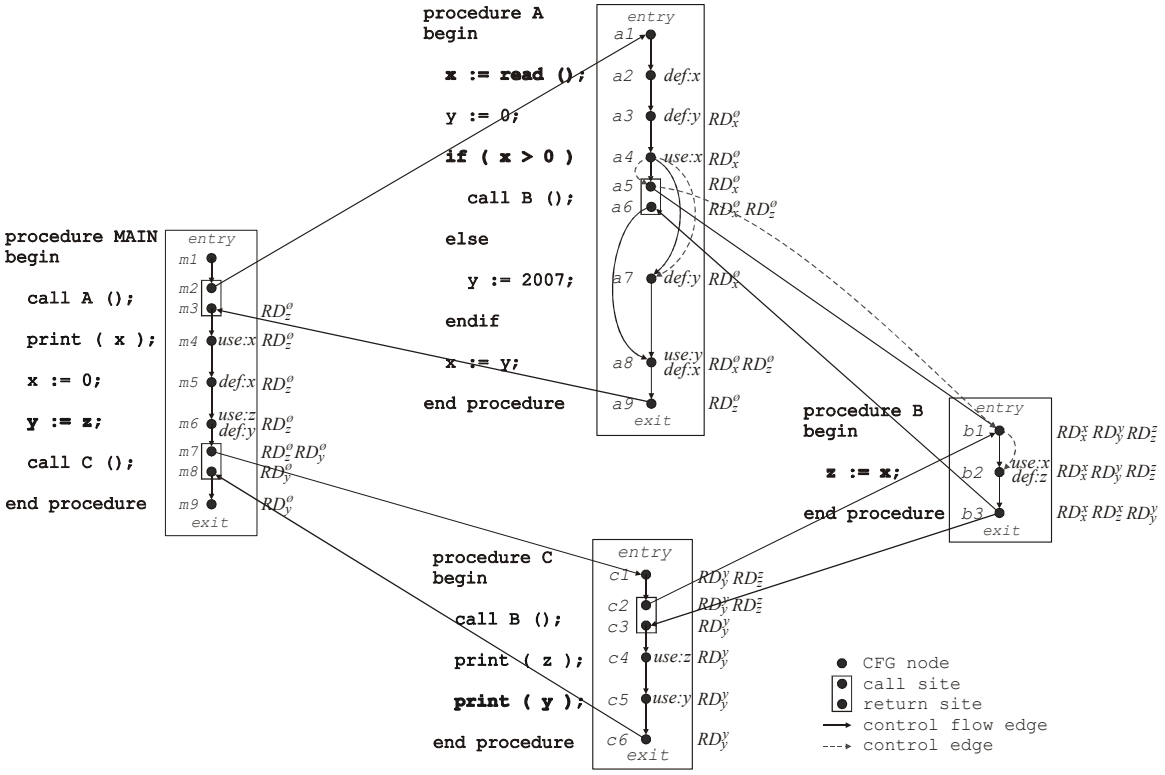



**Fig. 7. Tokens propagated during data flow slicing**

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*→*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.

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^∅) x→z      -- return from B
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 $v \rightarrow y$ is done 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 $RD_y^y$ (target) to the entry node of the called procedure and token $RD_v^y$ (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 exist 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 expected 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 go through node *b2*, 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 indented; 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)
```

### 4.1.2 Reasoning Full Slices

In full slicing, data dependent predicates induce propagation 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 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 both 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 too. 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 procedure exit nodes, the interprocedural traversal is unchanged.

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

## 4.2 Related Work

Various algorithms for calculating interprocedural slices exist, however, we are aware of no reasoning technique have been proposed to justify slice elements computed by these methods.

Chopping [Jackson and Rollins 1994b] 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. A chop thus gives the set of nodes composed of (all) the dependence chains between the source and the target, but does not provide information about a particular dependence chain from source to target, neither an appropriate calling sequence that covers is.

The solution proposed in this chapter answers both questions. We are aware of no other similar techniques for this problem.

## 4.3 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 comprehension of the resulting program slices requires considerable expertise and time. This chapter proposes a solution to "explain" slice elements by computing an actual dependence chain from the slicing criterion to the chosen slice element.

We implemented the presented reason-why algorithm in the Java programming language and integrated with the slicing tool presented in Chapter 3. We carried out several experiments on the same COBOL system and slices computed in programs of different sizes. The results showed that in all the cases 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 (in contrast to slicing). Note that slice computation has to be performed once, then several reasoning tasks can be initiated on the resulting slice elements. To our experiences these dependence chains are easily overviewed or analyzed by a human user, which is also due to control information included.

# Chapter 5

# Further Enhancements, Applications

In the case of large-size programs, the number of tokens to be propagated and stored can be very high. This chapter investigates different time-space tradeoffs and alternatives for the algorithm design.

The number of tokens to be stored during the slice computation can be reduced by calculating the topological sorting of procedures, then processing them in postorder. This processing order allows discarding tokens stored in procedures calling already processed procedures. The number of tokens to be propagated can be reduced by pre-computing the so called GREF-GMOD-KILL information. These sets can be used to filter unnecessary token propagations in advance. Between program modifications, subsequent slicing tasks can be sped up by reusing the results of previous calculations, namely, the summary edges. Furthermore, the token propagation method can be adapted to calculate and exploit flow edges, which can also reduce the number of required token propagations at the cost of increased space requirements.

A variant of the algorithm is capable of constructing definition-use (du) graphs, which can aid program comprehension and data-flow based testing. Finally, it is shown that how the token propagation method can be applied to calculate slicing variants called dicing and chopping.

For clarity of the presentation, we restrict to discussing forward slicing in this chapter, unless explicitly noted.

Section 5.1 describes how the number of stored tokens can be reduced via postorder processing of procedures. Section 5.2 discusses how the slice computation can be sped up by preprocessing the program. Section 5.3 describes how previous calculations can be reused. Section 5.4 discusses how the token propagation method can explore and exploit flow edges on demand. Section 5.5 presents how the token propagation method can be

applied to construct definition-use graphs. Section 5.6 describes how the method can be applied to other variants of slicing; finally, section 5.7 concludes the chapter.

## 5.1 Reducing Token Storage via Postorder Processing

Because of the possibly large number of tokens to be propagated in the case of real-world, large-size codes – which reached about one hundred million in our experiments – it might be necessary to reduce the memory requirements of the method. At implementing the slicer prototype (Chapter 3), we already used a kind of token storage reduction: we stored tokens in *intra*procedurally relevant nodes only. This section describes an *inter*procedural-level solution, during which, we determine the topological sorting of procedures and process them in postorder. Such processing order allows discarding tokens stored previously in internal nodes of the already processed procedures, at keeping entry and exit node tokens (summary edges) only.

The topological order of procedures can efficiently be calculated by a depth-first search in the call graph (first, assuming directed acyclic graphs), after which each procedure can be assigned a unique rPostorder index such that the index of each procedure is less than the index of any of the called procedures. Postorder processing of procedures means we always select a procedure (among the procedures to be processed) with the highest rPostorder index; processing a procedure means we apply the propagation rules to source tokens in the procedure as long as there is applicable one. Using procedure token worklists, as described in Chapter 3, a procedure is to be processed, when its token worklist is not empty, and its processing is continued until its worklist becomes empty. (Initially all the token worklists are empty except the procedure containing the node of the slicing criterion.) On selecting a procedure having index $i$ to process, token worklists of all procedures having index greater than $i$ are empty; after having completed processing this procedure, the index of the procedure to be selected next may be greater than $i$ when a token is propagated out of a call site to a called procedure, or less than $i$, respectively, when no more intraprocedural propagation is possible and tokens are propagated interprocedurally from the exit node to caller procedures only (if any).

Assume that we have just processed a procedure with rPostorder index $i$ and all token worklists of procedures having index greater than $i$ are still empty. If we discard all tokens – except the ones stored in entry and exit nodes – in all procedures with index greater than

or equal to *i*, and continue the token propagation, the resulting slice will not differ from the one that would have been computed without token deletion, as it is shown below.

The intraprocedural propagation in untouched procedures (having index less than *i*) is unaffected by the deletion, the only change could be due to interprocedural propagation Rules 3 and 4. As tokens in exit nodes are kept, even at calling a "cleaned-up" procedure the application of Rule 4 still results in the same tokens in the return sites. At applying Rule 3, two cases may occur: the token is already contained by the entry node of the called procedure (no propagation is performed), which case still corresponds to the propagation without deletion, or a new token, not yet contained by the entry node is propagated to it,. The latter case is relevant only when propagation is performed to a cleaned-up procedure. When a new token $RD_y^y$ is propagated to an entry node, all the tokens propagated intraprocedurally as a consequence of this token (including new propagations induced at affected nodes) have common backtrack index, identical to *y*. Or conversely, as $RD_y^y$ is a new token to the entry node, no other tokens may have existed in this procedure before with backtrack index *y*. Therefore, new propagations from entry nodes are surely propagated in the same way as no deletion would have happened (new and deleted tokens are *orthogonal*). In conclusion, since subsequent token propagations are unchanged from the point of token deletion on, we get the same slice as a result.

In the case of cyclic call graphs, we can still determine a generalized invocation order of procedures [Forgács 1994] – giving the same index to all the members of the SCC. Token deletion, in the presence SCCs, can be deduced form the acyclic case.

As the cost of determining the topological sorting of procedures as well as the cost of discarding unnecessary tokens is relatively low, the proposed time-space tradeoff may effectively reduce the memory requirements of the method.

## 5.2 Reducing Propagations via GREF-GMOD-KILL

The token propagation method presented in Chapter 2 represents a fully demand-driven approach to slicing that does not require any preprocessing information about the program – other than the control flow graph. Preprocessing means a preliminary analysis of the program (code or its graph representation) during which the whole program is analyzed gathering certain information in advance. This information can assist and speed up latter tasks. When one or only a few slices need to be computed (or the resulting slice is small, respectively), only a portion of the globally obtained information is exploited, hence, a

demand-driven solution likely to calculate slices the fastest – also considering the overhead of preprocessing. When several slicing tasks need to be performed, however, this cost can return.

Such a preprocessing possibility is the computation of GREF, GMOD, and KILL information about procedures. The GREF set is the set of program variables to which a procedure (or the procedures called by the procedure, respectively) contains variable reference, the GMOD set is the set of variables that a procedure may modify, whereas the KILL set is the set of program variables that the procedure surely defines during its call (all paths from entry to exit contains definition). This preprocess information needs to be computed once and can be reused in any subsequent slicing task later as long as the program code has not been changed. (Algorithms for computing these sets can be found in [Banning 1979; Forgács 1994].)

Regarding the token propagation method, the GREF set can be applied to block the propagation of a token $RD_x^y$ through a call site when $x$ is not in the GREF set of the called procedure – and so no use or further dependences can arise; thus, token $RD_x^y$ can directly be propagated to the return site, unless $x$ is in the KILL set.

In the method presented in Chapter 2, if a control token $RD_c^y$ is propagated to a call site $c$, it induces in the propagation of control tokens over all nodes of every (directly or indirectly) called procedure, and results in each variable $z$ defined in these procedures to return as data token $RD_z^y$ to return site $r$. Using the GMOD set, instead of propagating the control token to the called procedure from call site $c$, we can directly propagate data tokens corresponding to variables in the GMOD set to return site $r$, and simply mark called procedure(s) as "control dependent". (A control dependent procedure implies all its nodes to be included into the resulting slice.) Tokens propagated from the return site get the backtrack index of the control token propagated to the call site (i.e., $y$).

We implemented the above solution and measured both preprocessing and slicing times. We used the same subject programs and slicing criteria described in Chapter 3. The computation times of the GREF, GMOD, KILL sets (altogether) varied between 1 (*System 1*) and 43 seconds (*System 5*). In data-flow slicing, we observed no significant speed up, neither in full backward slicing. Full forward slicing also resulted in similar computation times in the case of smaller programs (*System 1*, *2*, and *3*). In larger programs (*System 4* and *5*), however, the average computation time has reduced to 19–40% of the slicing time without preprocessing. Hence, we can conclude that, in the case of large programs, the

performance of slicing may improve using the GREF, GMOD, KILL sets, and the cost of preprocessing can return even at performing a single slicing task.

## 5.3 Reuse of Summary Edges

As long as the program code has not been changed, summary edges remain unchanged as well. During subsequent slicing tasks, the propagation of the same token from the same procedure entry node results in the same tokens in the procedure exit node; therefore, by saving and loading entry and exit node tokens, the re-propagation of tokens from entry nodes can be avoided. However, simply saving and loading these tokens may result in incomplete slices, as "blocked" token propagations from procedure entry nodes due to loaded tokens omit potential slice nodes in the called procedures in latter slice computations (no marking will be performed by Rule 2).

Hence, in addition to entry and exit node tokens, "partial slices"[13] need to be maintained. A partial slice is composed of nodes marked as a consequence of a token propagated from a procedure entry node. These partial slices are related to tokens stored in entry nodes; or to be more specific, as tokens in entry nodes have identical token and backtrack indices, partial slices can directly be associated with program variables at procedures.

The backtrack index of the token currently being propagated determines the *source token* propagated from the entry node: if a token $RD_y^x$ with backtrack index $x$ is inserted into a node in a procedure, this token necessarily derives from a token $RD_x^x$ propagated from the entry node of this procedure. In this way, on token insertations, marked nodes, i.e., elements of the partial slices, can be associated with (added to) the partial slice of the corresponding backtrack index variable. (Nodes marked due to token insertions having Ø backtrack index are not maintained in any partial slices.)

Once a slice computation has been finished, we can save and later load these partial slices in addition to summary edges. In a subsequent slicing task, when the propagation of a token $RD_x^y$ is propagated from a call site to a procedure entry node already containing $RD_x^x$ (loaded) and so its propagation is blocked, we simply include the partial slice of the called procedure associated with program variable $x$ into the resulting slice, and continue the

[13] A similar concept but with different computation was proposed in [Harrold and Ci 1998].

token propagation from the return site considering summary edges of the called procedure, i.e., exit node tokens (loaded).

Because of the interprocedurally induced token propagations, partial slices obtained due to the propagation of a token $RD_x^x$ from a procedure entry node does not limit to a single procedure; a token $RD_y^x$ propagated from a call site to the called procedure potentially induces new propagation with token $RD_y^y$ from its entry node. The partial slice belonging to variable $y$ obtained in the called procedure therefore has to be *linked* to the partial slice associated with variable $x$ of the caller procedure. On blocked token propagations, all linked partial slices are to be included in order to obtain correct slices.

With new propagations potentially new summary edges and new partial slices are computed, which can be saved incrementally.

## 5.4 On-demand Computation of Flow Edges

The presented token propagation method operates over control flow graphs and does not require revealing intraprocedural dependences between statements in advance, which is a prerequisite of the SDG-based approach. On the other hand, using the current technique, it may occur that different tokens with the same token- but different backtrack indices are propagated from the same node redundantly – exploring the same paths intraprocedurally.

To avoid it, once the token propagation from a node for a given token index has been completed, the revealed flow edges can be stored between the source and the reached node (from entry nodes, definition nodes, and return sites to use nodes, call sites, and exit nodes, respectively). Later, when a token with the same token- but a different backtrack index is to be propagated from the same node again, tokens can directly be propagated through the previously explored flow edges omitting propagations over intermediate control flow graph nodes.

This solution corresponds to a demand-driven construction of the program-dependence graph; however, the space requirements of maintaining flow edges in addition to the control flow graph can be very high in the case of large programs.

## 5.5 On-demand Construction of the Du-graph

The token propagation method presented in Chapter 2 uses variable identifiers as token- and backtrack indices by which a significant speed up can be gained compared to the

method proposed by Hajnal and Forgács [2002]. That solution used definition identifiers as token indices. By contracting tokens belonging to different definitions of the same variable, however, we lose the information which are the actual definitions affecting the use being marked. In some applications, this information is more important than computing slices quickly.

A definition-use (du) graph is a directed graph in which nodes represent program statements and edges represent potential data-dependences between them. Using definition identifiers as token and backtrack indices, we can construct the du graph during the token propagation (in the context of the current slice). The constructed du graph can then be visualized that can aid program analysis, program comprehension, regression testing, or support data-flow based testing criteria, respectively. Note that when creating test cases to satisfy such a criterion it is highly important to consider realizable definition-use pairs, definition-use chains, respectively. If we apply the method for all definitions contained by the program, the set of definition-use pairs can be computed simultaneously. These definition-use pairs need to be covered by test cases to satisfy the all-uses (*all du pairs*) criterion [Rapps and Weyuker 1985]. In addition, by controlling the length of the investigated definition-use chains, this algorithm can support other testing criteria such as Ntafos' required k-tuples [Ntafos 1984] or all program-functions [Forgács and Bertolino 2002]. Note that flow edges of PDGs differ from du graph edges: flow edges restrict to intraprocedural dependences, whereas du graph edges represent dependences crossing procedure boundaries as well.

## 5.6 Dicing, Chopping

There are other variants of slicing based on set operations on one or more slices, called dicing [Lyle 1984] and chopping [Jakson and Rollins 1994b]. They provide a more focused approach at localizing the set of statements likely to contain the bug during debugging.

Dicing uses the information that the results of some variables fail on some test cases while other variables pass all tests. It reduces the number of statements to be examined. A program dice is obtained by subtracting the successful execution slices (slices of variables showing correct values) from the failed execution slice (slices of variables showing incorrect results).

The token propagation method is capable of computing the union slice of several slicing criteria simultaneously by starting multiple initial tokens corresponding to a "compound

slicing criterion"[14] (all tokens having Ø backtrack index). Note that the computation of the union slice is likely faster than computing individual slices, as token propagations common in different computations are to be performed once.

It can be exploited in dicing at computing the union slice of all statements resulting in correct values. During the first phase, we use mark label *passed* to mark nodes (instead of "in the slice" in Chapter 2). In the next phase, at keeping *passed* marks, we compute the slice of the failed variable, using mark label *failed*, with the modification that we do not mark or start new token propagations from nodes marked as *passed*. The dice (subtracted slice) is given by the set of nodes marked as *failed*. Note that the computation cost of the latter slice is likely less than computing the full backward slice, as it skips previous token propagations from *passed* nodes.

Chopping reveals statements involved in a transitive dependence from one specific statement (source criterion) to another one (target criterion). It shows how one variable affects the other. A chop for a chopping criterion (*s*, *t*) is the set of nodes that are part of a dependence chain from source node *s* to target node *t*. A program chop can be defined as the intersection of the backward and the forward slice, from *t*, and from *s*, respectively. As it has been shown in Chapter 3, backward slice computation times are typically shorter than forward slice computation times, for a chopping criterion (*s*, *t*) we compute the backward slice of *t*, and use mark label *backward* in the first phase. In the next phase, the forward slice of *s* is computed with the modification that new token propagations are started from *backward* marked nodes only, and we use mark label *forward*, respectively. The chop (intersection slice) is given by the set of nodes marked by labels both *forward* and *backward*. Note that the computation time of the forward slice is likely shorter than the computation time of the full forward slice from *s*, as in the second phase, token propagations are started from *backward* marked nodes only.

## 5.7 Conclusions

The algorithm presented in Chapter 2 is a demand-driven approach requiring no preliminary, exhaustive analysis of the program's code, which results in low computation times when only a few slices need to be computed, or in interactive use, respectively. In

[14] A union slice of a "compound slicing criterion" composed of more program points and possibly more program variables is the union of slices computed for its "atomic slicing criteria" composed of a single program point and a single variable.

other cases, however, performing some preprocessing of the program, the time and space requirements of the technique can further be reduced.

By computing the topological sorting of procedures and processing them in postorder numerous tokens stored previously can be discarded that reduces the space requirements of the method. By computing the GREF-GMOD-KILL sets several token propagation can be filtered in advance that potentially reduces both time and space requirements of the method at the cost of some preprocessing overhead. Another time-space tradeoff is the reuse of the results of previous calculation by maintaining summary edges and partial slices. In this way, repeated computation of summary edges and re-propagation of tokens can be avoided. It is also discussed how flow edges can be identified and exploited on the fly to reduce the number redundant token propagations.

A variant of the token propagation method is presented that can be applied to construct definition-use graphs to aid program comprehension and testing data-flow based testing. It is also discussed how the method can be adapted to calculate program dices and chops.

# Chapter 6

# Summary

Data-flow analysis is an important technique of program analysis, which is already used in optimizing compilers. The key concepts of data-flow analysis were developed in the late 60s. Over the past decades, the majority of new applications have focused on software quality.

The concept of program slicing extends data-flow analysis to accommodate control dependences. Using program slicing, parts of the code can be extracted automatically, called a *slice*, which focuses on selected aspects of semantics. As program slices are typically much smaller than the whole program code they can be more easily understood or maintained.

Program slicing was originally motivated to aid debugging activities. Various notions of program slices have been proposed as well as a number of methods to compute them. By now numerous applications of program slicing exist in software engineering, including software testing, software maintenance, program comprehension, re- and reverse engineering, and program integration.

The motivation of the dissertation was to be able to analyze legacy COBOL systems. We found that previous techniques are not adequate to slice large COBOL systems. By applying existing methods either precision or scalability is violated. System issues are often omitted in previous approaches; moreover, interactive contexts require a demand-driven solution.

The proposed novel static program slicing technique is based on token propagation over the control flow graph. The algorithm is conceptually simple, which allows of easy implementation, but general enough to adapt to a larger class of programming languages. Tokens are propagated along realizable program paths by which we obtain accurate results. The method is inherently demand-driven, that is, it computes the necessary information when they are needed. The technique is compared to other related solutions.

An efficient implementation of the proposed algorithm has been presented, and its performance was evaluated on real-world COBOL codes. Experiments were performed on a large number of test cases to provide details about its real efficiency, applicability.

To make slicing more user-friendly we proposed a method to reason about slice elements that aids slice comprehension.

We also investigated different time-space tradeoffs and alternatives for the algorithm design. We described how to reduce the number of tokens to be stored and how to speed up slicing by preprocessing, computing flow edges, or reusing the results of previous calculations, respectively. Construction of the definition-use graphs as well as the adaptation of the method to dicing and chopping is also discussed.

## 6.1 New Scientific Results

In this section the main contributions of the dissertation are summarized.

THESIS 1. *I have analyzed existing static program slicing techniques with respect to their applicability to large-size, legacy COBOL codes, and concluded that previous methods are impractical due to their prohibitive time or space requirements; practical use of program slicing on industrial-scale codes requires a demand-driven approach.*

These results are shown in Sections 1.3–1.5. Author's publications related to the thesis are: [Forgács and Hajnal 1998b; Forgács et al. 1998; Hajnal and Forgács 2002; Hajnal and Forgács 2012a]

THESIS 2. *I have developed a novel demand-driven static program slicing technique based on token propagation over control flow graphs, which computes accurate program slices with respect to realizable program paths. I proved correctness and completeness of the method.*

These results are shown in Sections 2.2–2.5. Author's publications related to the thesis are: [Hajnal and Forgács 2002; Hajnal and Forgács 2012a]

THESIS 3. *I implemented the proposed demand-driven technique and evaluated its performance on industrial-scale COBOL codes. I concluded that the method is capable of efficiently calculating precise program slices of reasonable size; longer computation times result in overly large slices uninterpretable for human users.*

These results are shown in Sections 3.1–3.3. Author's publication related to the thesis is: [Hajnal and Forgács 2012a]

THESIS 4. *I have developed a novel "slice explainer" technique to aid slice comprehension. The algorithm is applicable to calculate reasoning about computed slice elements in the form of actual dependence chains at low cost, which is confirmed by empirical results.*

These results are shown in Sections 4.1, 4.3. Author's publication related to the thesis is: [Hajnal and Forgács 2012b]

THESIS 5. *I proposed possible improvements for the algorithm design, and showed that by applying these techniques, time or space efficiency of the token propagation-based slicing can be further increased. I have developed modified algorithms to determine definition-use graphs, program dices and chops.*

These results are shown in Sections 5.1-5.6. Author's publication related to the thesis is: [Hajnal and Forgács 2002]

## 6.2 Further Research Directions

In Chapter 5, we proposed several enhancement possibilities that are not yet fully evaluated experimentally. The presented demand-driven token propagation method assumes interactive contexts in which only a few slices need to be computed between program modifications. There can be however other usage scenarios that require more program slicing tasks to be performed (e.g. program comprehension). We are planning to study such usage scenarios and design *typical* slicing criterion sequences. Based on that, the proposed time-space tradeoffs can be more precisely evaluated.

We considered no pointer variables (references, function pointers, etc.), which are present in modern programming languages. *Points-to* analysis during which we determine the set of possible variables to which a pointer may point to and its integration with the proposed token propagation method are also a great challenge. Object-oriented programming constructs rise other problems haven't been investigated yet, but are also worth researching.

# Abbreviations

| | |
|---|---|
| *APF* | All Program-Functions |
| *CFG* | Control Flow Graph |
| *DAG* | Directed Acyclic Graph |
| *iCFG* | Intraprocedural Control Flow Graph |
| *ICFG* | Interprocedural Control Flow Graph |
| *IDE* | Integrated Development Environment |
| *PDG* | Program Dependence Graph |
| *SCC* | Strongly Connected Component |
| *SDG* | System Dependence Graph |

# Appendix: Author's Publications

## Publications on the topic of the thesis

[J] JOURNALS

[J1] Ákos Hajnal, István Forgács. 2012. A demand-driven approach to slicing legacy COBOL systems. JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS, 24(1), pp. 67–82, John Wiley & Sons. (IF: 0.844)

[J2] Ákos Hajnal, István Forgács. 2012. Understanding program slices. ACTA CYBERNETICA (to appear).

[C] CONFERENCES

[C1] Ákos Hajnal, István Forgács. 2002. A precise demand-driven def-use chaining algorithm. In: 6th European Conference on Software Maintenance and Reengineering (CSMR'2002), IEEE Computer Society, pp. 77–86.

[C2] Ákos Hajnal, István Forgács. 1998. An applicable test data generation algorithm for domain errors. In: 1998 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98), ACM New York, pp. 63–72. SOFTWARE ENGINEERING NOTES, 23(2), ACM New York, pp. 63–72.

[C3] István Forgács, Ákos Hajnal. 1998. Automated test data generation to solve the Y2k problem. In: 2nd International Software Quality Week Europe 1998 (QWE'98), p. 10 (paper 2S).

[C4] István Forgács, Ákos Hajnal, Éva Takács. 1998. Regression slicing and its use in regression testing. In: 22nd Annual International Computer Software and Applications Conference (COMPSAC'98), IEEE Computer Society, pp. 464–469.

[O] OTHER

[O1] István Forgács, Ákos Hajnal. 2011. Forráskód-analízis: Olvassunk a sorok között! COMPUTERWORLD SZÁMÍTÁSTECHNIKA, XLII:(41), pp. 10–12 (in Hungarian).

[O2] István Forgács, Ákos Hajnal. 1997. Szoftver tesztelés. Jegyzet Szoftver minőség és tesztelés tárgyhoz, BME, ELTE Doktori Iskola, 1997/1998-as tanév, p. 41 (in Hungarian).

# Further publications

[J] JOURNALS

[J3] David Isern, Antonio Moreno, David Sánchez, Ákos Hajnal, Gianfranco Pedone, László Zsolt Varga. 2011. Agent-based execution of personalised home care treatments. APPLIED INTELLIGENCE, 34(2), Springer-Verlag, pp. 155–180. (IF: 0.849)

[B] BOOK CHAPTERS

[B1] Ákos Hajnal, Tamás Kifor, Gergely Lukácsy, László Zsolt Varga. 2011. Web services as XML data sources in enterprise information integration. In: Enterprise Information Systems: Concepts, Methodologies, Tools and Applications, 2011, Hershey PA: Information Science Reference, pp. 972–985. Services and Business Computing Solutions with XML: Applications for Quality Management and Best Processes (P. Hung, Ed.), 2009, IGI Global, pp. 82–97.

[B2] Ákos Hajnal, Antonio Moreno, Gianfranco Pedone, David Riano, László Zsolt Varga. 2009. Formalizing and leveraging domain knowledge in the K4CARE home care platform. In: Semantic knowledge management: An ontology-based framework (A. Zilli, E. Damiani, P. Ceravolo, A. Corallo, G. Elia, Eds.), Information Science Reference, pp. 279–302.

[B3] László Zsolt Varga, Ákos Hajnal, Zsolt Werner. 2005. The WSDL2Agent tool. In: Software Agent-Based Applications, Platforms and Development Kits (R. Unland, M. Klusch, M. Calisti, Eds.), Whitestein Series in Software Agent Technologies, Birkhauser Basel, pp. 197–223.

[C] CONFERENCES

[C5] Tamás Kifor, Tibor Gottdank, Ákos Hajnal, Péter Baranyi, Brúnó Korondi, Péter Korondi. 2011. Smartphone emotions based on human-dog interaction. In: 2nd International Conference on Cognitive Infocommunications: CogInfoCom 2011, IEEE Computer Society, pp. 1–6.

[C6] Tamás Kifor, Tibor Gottdank, Ákos Hajnal, Csanád Szabó, András Róka, Brúnó Korondi, Péter Korondi. 2011. EthoPhone, human-dog interaction inspired affective computing for smartphone. In: Proceedings IEEE/ASME International Conference on Advanced Intelligent Mechatronics, IEEE Computer Society, pp. 542–547.

[C7] Ákos Hajnal, David Isern, Antonio Moreno, Gianfranco Pedone, László Zsolt Varga. 2007. The role of knowledge in designing an agent platform for home care. In: 2nd International Conference on Knowledge Management in Organizations (KMO), pp. 16–26.

[C8] Ákos Hajnal, David Isern, Antonio Moreno, Gianfranco Pedone, László Zsolt Varga. 2007. Knowledge driven architecture for home care. In: Multi-agent systems and applications V: 5th International Central and Eastern European Conference on Multi-agent Systems (CEEMAS 2007), pp. 173–182. LECTURE NOTES IN ARTIFICIAL INTELLIGENCE, Vol. 4696, Springer-Verlag, pp. 173–182.

[C9] Ákos Hajnal, Gianfranco Pedone, László Zsolt Varga. 2007. Ontology-driven agent code generation for home care in Protégé. In: 10th International Protégé Conference: Budapest, Hungary, pp. 91–93.

[C10] Ákos Hajnal, Tamás Kifor, Gianfranco Pedone, László Zsolt Varga. 2007. Benefits of provenance in home care. In: Healthgrid 2007. STUDIES IN HEALTH TECHNOLOGY AND INFORMATICS, Vol. 126: From genes to personalized healthcare: grid solutions for the life sciences (N. Jacq, Y. Legré, H. Muller, I. Blanquer, V. Breton, D. Hausser, V. Hernández, T. Solomonides, M. Hofman-Apitius, Eds.), IOS Press, pp. 330–337.

[C11] László Zsolt Varga, Ákos Hajnal, Zsolt Werner. 2004. An agent based approach for migrating web services to semantic web services. In: 11th International Conference on Artificial Intelligence: Methodology, Systems (AIMSA 2004). LECTURE NOTES IN COMPUTER SCIENCE, Vol. 3192, Springer-Verlag, pp. 371–380.

[C12] László Zsolt Varga, Ákos Hajnal. 2003. Engineering web service invocations from agent systems. In: 3rd International/Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2003). LECTURE NOTES IN COMPUTER SCIENCE, Vol. 2691, Springer-Verlag, pp. 626–636.

[O] OTHER

[O3] Jonathan Dale, Ákos Hajnal, Martin Kernland, László Zsolt Varga. 2003. Integrating web services into Agentcities. Agentcities Technical Recommendation Document (actf-rec-00006).

# Bibliography


AGRAWAL, G., AND GUO, L. 2001. Evaluating explicitly context-sensitive program slicing. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, 6–12.

AGRAWAL, H., DEMILLO, R.A, AND SPAFFORD, E.H. 1993. Debugging with dynamic slicing and backtracking. *Software, Practice and Experience, 23(6)*, 589–616.

AGRAWAL, H., AND HORGAN, J. 1990. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, New York, USA, 246–256.

ANTONIOL, G., FIUTEM, R, LUTTERI, G, TONELLA, P., ZANFEI, S., AND MERLO, E. 1997. Program understanding and maintenance with the CANTO environment. In *International Conference on Software Maintenance,* 72–81.

ATKINSON, D.C., AND GRISWOLD, W.G. 1996. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, 16–27.

BANNING, J.P., 1979. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 29–41.

BINKLEY, D. 1993. Precise executable interprocedural slices. In *ACM Letters on Programming Languages and Systems*, 2(1–4), 31–45.

BINKLEY, D. 1997. Semantics guided regression test cost reduction. In *IEEE Transactions on Software Engineering*, 23(8), 498–516.

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

BINKLEY, D. 2007. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, 104–119.

BINKLEY, D., AND HARMAN, M. 2003. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the International Conference on Software Maintenance*, 44–53.

BINKLEY, D., HORWITZ, S., AND REPS, T. 1995. Program integration for languages with procedure calls. In *Transactions on Programming Languages and Systems*, 4(1), 3–35.

BROWN, G.D. 2000. COBOL: The failure that wasn't. *COBOLReport.Com*. Available: http://web.archive.org/web/20000926032455/ www.cobolreport.com/columnists/gary/05152000.htm [28 October 2009].

CANFORA, G., CIMITILE, A., AND DE LUCIA, A. 1998. Conditioned program slicing. In *Information and Software Technology Special Issue on Program Slicing*, 40(11–12), 595–607.

CANFORA, G., CIMITILE, A., DE LUCIA, A., AND DI LUCCA, G.A. 1994a. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance*, 424–433.

CANFORA, G., CIMITILE, A., AND MUNRO, M. 1994b. $RE^2$: Reverse engineering and reuse reengineering. In *Journal of Software Maintenance: Research and Practice*, 6(2), 53–72.

CIMITILE, A., DE LUCIA, A., AND MUNRO, M. 1996. A specification driven slicing process for identifying reusable functions. In *Journal of Software Maintenance: Research and Practice*, 8(3), 145–178.

CODESURFER 2009. GrammaTech, Inc., http://www.grammatech.com/products/codesurfer [28 October 2009].

DE LUCIA, A., FASOLINO, A.R., AND MUNRO, M. 1996. Understanding function behaviours through program slicing. In *Proceedings of the 4th IEEE Workshop on Program Comprehension*, 9–18.

VAN DEURSEN, A., AND MOONEN, L., 1999. Understanding COBOL systems using inferred types. In *Proceedings of the 7th International Workshop on Program Comprehension*, 74–81.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M.L. 1997. A practical framework for demand-driven interprocedural data flow analysis. In *ACM Transactions on Programming Languages and Systems*, 19(6), 992–1030.

EMAMI, M., GHIYA, R., AND HENDREN, L.J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 20–24.

FERRANTE, J., OTTENSTEIN, K.J., AND WARREN, J.D. 1987. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, 9(3), 319–349.

FIELD, J., AND RAMALINGAM, G. 1999. Identifying procedural structure in Cobol programs. In *ACM SIGSOFT Software Engineering Notes*, 24(4), 1–10.

FORGÁCS, I. 1994. Double iterative framework for flow-sensitive interprocedural data flow analysis. In *ACM Transactions on Software Engineering and Methodology*, 3(1), 29–55.

FORGÁCS, I., AND BERTOLINO, A. 2002. Preventing untestedness in data flow-based testing. In *Software Testing, Verification & Reliability*, 12(1), 29–58.

FORGÁCS, I., AND HAJNAL, Á. 1998a. An applicable test data generation algorithm for domain errors. In *Proceedings of the 1998 ACM/SIGSOFT International Symposium on Software Testing and Analysis*, *Software Engineering Notes* 23(2), 63–72.

FORGÁCS, I., AND HAJNAL, Á. 1998b. Automated test data generation to solve the Y2k problem. In *Proceedings of the 2nd International Software Quality Week Europe*, p. 2S.

FORGÁCS, I., TAKÁCS, É., AND HAJNAL, Á. 1998. Regression slicing and its use in regression testing. In *Proceedings of IEEE International Computer Software and Applications Conference*, 464–469.

GALLAGHER, K.B. 1990. Surgeon's assistant limits side effects. In *IEEE Software, 7(64)*, p. 95.

GALLAGHER, K.B. 1992. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the Conference on Software Maintenance*, 236–244.

GALLAGHER, K.B., AND LYLE, J.R. 1991. Using program slicing in software maintenance. In *IEEE Transactions on Software Engineering*, 17(8), 751–761.

GRISWOLD, W.G. 2001. Making slicing practical: the final mile. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, p.1.

GUPTA, R., HARROLD, M.J., AND SOFFA, M.L. 1992. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, 299–308.

GUPTA, R., SOFFA, M.L., AND HOWARD, J. 1997. Hybrid slicing: Integrating dynamic information with static analysis. In *ACM Transactions on Software Engineering and Methodology*, 6(4), 370–397.

GYIMÓTHY, T., BESZÉDES, Á., AND FORGÁCS, I. 1999. An efficient relevant slicing method for debugging. In *Lecture Notes in Computer Science*, 303–321.

HAJNAL, Á., AND FORGÁCS, I. 2002. A precise demand-driven def-use chaining algorithm. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, 77–86.

HAJNAL, Á., AND FORGÁCS, I. 2012a. A demand-driven approach to slicing legacy COBOL systems. In *Journal of Software: Evolution and Process*, 24(1), 67–82.

HAJNAL, Á., AND FORGÁCS, I. 2012b. Understanding program slices. In *Acta Cybernetica* (to appear).

HARMAN, M., AND DANICIC, S. 1995. Using program slicing to simplify testing. In *Software Testing, Verification and Reliability*, 5(3), 143–162.

HARMAN, M., AND DANICIC, S. 1997. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension*, 70–79.

HARMAN, M., HIERONS, R.M., FOX, C., DANICIC, S., AND HOWROYD, J. 2001. Pre/Post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*, 138–147.

HARROLD, M. J., AND ROTHERMEL, G. 1997. Aristotle: A system for research on and development of program-analysis-based tools. *Technical Report OSU-CISRC-3/97-TR17*, Department of Computer and Information Science, The Ohio State University.

HARROLD, M.J., ROTHERMEL, G., AND SINHA, S. 1998. Computation of interprocedural control dependence. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 11–20.

HECHT, M.S. 1977. Flow analysis of Computer Programs. *Elsevier North-Holland Inc*.

HIERONS, R., HARMAN, M., AND DANICIC, S. 1999. Using program slicing to assist in the detection of equivalent mutants. In *Software Testing, Verification and Reliability*, 9(4), 233–262.

HIERONS, R., HARMAN, M., FOX, C., OUARBYA, L., AND DAOUDI, M. 2002. Conditioned slicing supports partition testing. In *Software Testing, Verification and Reliability*, 12(1), 23–28.

HOFFNER, T., KAMKAR, M., AND FRITZSON, P., 1995. Evaluation of program slicing tools. In *2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, 51–69.

HORWITZ, S., PRINS, J., AND REPS, T. 1989. Integrating non-interfering versions of programs. In *Transactions on Programming Languages and Systems,* 11(3), 345–387.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. In *ACM Transactions on Programming Languages and Systems*, 12(1), 26–60.

HORWITZ, S., REPS, T., AND SAGIV, M. 1995. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 104–115.

INDUS. SAnToS Laboratory: Indus, a toolkit to customize and adapt Java programs. http://indus.projects.cis.ksu.edu [8 November 2011]

JACKSON, D., AND ROLLINS, E.J. 1994a. Abstraction mechanisms for pictorial slicing. In *Proceedings of the IEEE Workshop on Program Comprehension*, 82–88.

JACKSON, D., AND ROLLINS, E.J. 1994b. A new model of program dependences for reverse engineering. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2–10.

KOREL, B., AND LASKI, J. 1988. Dynamic program slicing. In *Information Processing Letters 29,3*,155–163.

KRINKE, J. 2002. Evaluating context-sensitive slicing and chopping. In *Proceedings of the International Conference on Software Maintenance*, 22–31.

KRINKE, J. 2003. Advanced Slicing of Sequential and Concurrent Programs. Ph.D. Thesis, Universität Passau.

KRINKE, J. 2006. Effects of context on program slicing. In *Journal of Systems and Software*, 79(9), 1249–1260.

KRINKE, J., AND SNELTING, G. 1998. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, Special issue on Program Slicing, 661–675.

LIANG, D., AND HARROLD, M. J. 1999. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the IEEE International Conference on Software Maintenance*, 421–430.

LIVADAS, P.E., AND ALDEN, S.D. 1993. A toolset for program understanding. In *Proceedings of the IEEE Second Workshop on Program Comprehension*, 110–118.

LIVADAS, P.E., AND CROLL, S. 1992. Program slicing. *Technical Report SERC-TR61 -F*, Computer Science and Information Services Department, University of Florida, Gainesville, FL.

LOYALL, J.P, AND MATHISEN, S.A. 1993. Using dependence analysis to support the software maintenance process. In *Proceedings of the Conference on Software Maintenance*, 282–291.

LYLE, J.R. 1984. Evaluating variations of program slicing for debugging. *Ph.D. Thesis*, University of Maryland.

LYLE, J., AND WALLACE, D. 1997. Using the unravel program slicing tool to evaluate high integrity software. In *Proceedings of Software Quality Week (May 1997)*.

MOCK, M., ATKINSON, D.C., CHAMBERS, C., AND EGGERS, S.J. 2002. Improving program slicing with dynamic points-to data. In *ACM SIGSOFT Software Engineering Notes*, 27(6), 71–80.

MYERS, E. M. 1981. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 219–230.

NISHIMATSU, A., JIHIRA , M., KUSUMOTO, S., AND INOUE, K. 1999. Call-mark slicing: An efficient and economical way of reducing slice. In *Proceedings of the International Conference of Software Engineering*, 422–431.

NTAFOS, S.C. 1984. On required element testing. In *IEEE Transactions on Software Engineering*, 10(6), 795–803.

ORSO, A., ORSO, R., SINHA, S., AND HARROLD, M.J. 2001. Incremental slicing based on data-dependences types. In *Proceedings of the IEEE International Conference on Software Maintenance*, 158–167.

OTTENSTEIN, K.J., AND OTTENSTEIN, L.M. 1984. The program dependence graph in a software development environment. In A*CM SIGPLAN Notices*, 19(5), 177–184.

PODGURSKI, A., AND CLARKE, L.A. 1990. A formal model of program dependences and its implications for software testing, debugging, and maintenance. In *IEEE Transactions on Software Engineering*, 16(9), 965–979.

RANGANATH, V.P., AMTOFT, T., BANERJEE, A., HATCLIFF, J., AND DWYER, M.B. 2007. A new foundation for control dependence and slicing for modern program structures. In *ACM Transactions on Programming Languages and Systems*, 29(5), Article 27.

RAPPS, S., AND WEYUKER, E.J. 1985. Selecting software test data using data flow information. In *IEEE Transactions on Software Engineering*, 11(4), 367–375.

REPS, T.W. 1993. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases*, 163–196.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 49–61.

REPS, T., HORWITZ, S., SAGIV, M., AND ROSAY, G. 1994. Speeding up slicing. In *ACM SIGSOFT Software Engineering Notes*, 19(5), 11–20.

SINHA, S., HARROLD, M. J., AND ROTHERMEL, G. 1999. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, 432–441.

STEINDL, C. 1998. Intermodular slicing of object-oriented programs. *Lecture Notes In Computer Science (1383)*, 264–278.

VENKATESH, G.A. 1991. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 107–119.

WEISER, M. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)*, 439–449.

WEISER, M. 1984. Program slicing. In *IEEE Transactions on Software Engineering*, 10(4), 352–357.

XU, B., QIAN, J., ZHANG, X., WU, Z., AND CHEN, L. 2005. A brief survey of program slicing. In *ACM SIGSOFT Software Engineering Notes*, 30 (2), 1–36.

# Abstract

COBOL is often thought as an old-fashioned programming language which is of little importance by now. However, the fact is that several billion lines of COBOL codes are actively used today and COBOL is still the dominant language for business applications. Many of the legacy systems are more than 30–40 years old, whose maintenance is very labor-intensive and costly task.

Program slicing is a potentially useful analysis for aiding such maintenance activities. The concept of program slicing was proposed by Mark Weiser that extends data-flow analysis by accommodating control dependences (effects of data-flow on control). Slicing has found its applications in different areas of software engineering including software testing, software maintenance, program comprehension, re- and reverse engineering, and program integration.

COBOL has been fallen out of the focus of the program slicing research so far, and as it is shown that, existing methods are inefficient in performing these tasks due to their prohibitive time or space requirements. The work followed aimed at developing a new static program slicing approach that addresses the challenges raise at slicing industrial-scale COBOL codes.

The dissertation presents a novel demand-driven static program slicing technique using token propagation, which is based on control flow graphs that are more easily adaptable to accommodate different programming language constructs, and attains the accuracy of the system dependence graph-based approach. Experimental results show that the presented method is indeed capable of computing precise program slices quickly, whereas longer computation times always result in overly large slices uninterpretable for human users.

A novel technique called the “reason-why algorithm” is proposed to reason about slice elements by determining an actual dependence chain from the slicing criterion to the chosen slice element. Without such a tool, verification or comprehension of the resulting program slice requires considerable expertise and time.

Different time-space tradeoffs and alternatives for the algorithm design are proposed to reduce the number of tokens to be propagated and stored. Modified algorithms are presented to determine definition-use graphs, program dices and chops.

# Kivonat

A COBOL-ra gyakran, mint egy elavult programozási nyelvre gondolnak, pedig napjainkban is több milliárd sornyi COBOL kód fut világszerte, sőt, még mindig ez a leggyakrabban használt programnyelv az üzleti alkalmazásokban. A COBOL rendszerek nemritkán 30–40 évesek, karbantartásuk rendkívül munkaigényes, költséges feladat.

A program szeletelés alkalmazása nagy segítséget nyújthat ezen feladatok elvégzésében. A program szeletelés ötletét Mark Weiser publikálta először, amely az adatfolyam-analízist terjeszti ki a vezérlési függőségekre. A program szeletelés alkalmazhatóságát a szoftver technológia számos területén igazolták, köztük a szoftver tesztelésben és karbantartásban, a programmegértés és visszafejtésben és a programintegráció területén.

A program szeletelés területén folytatott kutatás eddig kevés figyelmet fordított a COBOL programok szeletelési problémáira, és ahogy ezt megmutatjuk, a feladatra a létező módszerek nem alkalmazhatóak hatékonyan a gyakorlatban. A munka célja egy új statikus programszeletelési megközelítés kidolgozása volt, amely megoldásokat keres azokra a problémákra, amelyek ipari méretű COBOL programok szeletelésekor merülnek fel.

A disszertáció egy új igényvezérelt statikus programszeletelési technikát mutat be, amely a vezérlési folyamgráfokon történő token terjesztés révén pontos program szeleteket képes meghatározni. A folyamgráfok könnyebben adaptálhatóak a különböző programozási nyelvekben használt konstrukciók reprezentálására, és az algoritmus megtartja a rendszer függőségi gráf-alapú módszerek által elérhető szeletpontosságot. A kísérleti eredmények azt mutatják, hogy a módszer rövid idő alatt képes pontos program szeletek meghatározására, hosszabb számítási idők esetén az eredmény szeletek mérete túlságosan nagy lesz, amelyek már amúgy sem értelmezhetők a felhasználók számára.

A disszertáció bemutat egy új, program szelet megértést támogató technikát is, amely alkalmas konkrét függőségi láncok meghatározására a kiválasztott, indokolandó szeletelembe.

Különböző továbbfejlesztési lehetőségeket, algoritmus-tervezési alternatívákat ismerhetünk meg, amelyek segítségével csökkenthető a módszer futási idő- vagy tárhely igénye. Módosított algoritmusok alkalmasak definíció-felhasználás gráfok, program vágások, darabolások meghatározására.