Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology 48 (2006) 549–565
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Theory and algorithms for slicing unstructured programs

Mark Harman[a,*], Arun Lakhotia[b], David Binkley[c]

[a]Department of Computer Science, King's College, Strand, London WC2R 2LS, UK
[b]University of Louisiana at Lafayette, Lafayette, LA 70504, USA
[c]Loyola College, 4501 North Charles Street, Baltimore, MD 21210, USA

Received 21 January 2005; revised 31 May 2005; accepted 10 June 2005
Available online 30 August 2005

Abstract

Program slicing identifies parts of a program that potentially affect a chosen computation. It has many applications in software engineering, including maintenance, evolution and re-engineering of legacy systems. However, these systems typically contain programs with unstructured control-flow, produced using `goto` statements; thus, effective slicing of unstructured programs remains an important topic of study.

This paper shows that slicing unstructured programs inherently requires making trade-offs between three slice attributes: termination behaviour, size, and syntactic structure. It is shown how different applications of slicing require different tradeoffs. The three attributes are used as the basis of a three-dimensional theoretical framework, which classifies slicing algorithms for unstructured programs. The paper proves that for two combinations of these dimensions, no algorithm exists and presents algorithms for the remaining six combinations.

Keywords: Program slicing; Amorphous slicing; Unstructured control-flow

1. Introduction

Mark Weiser first defined a program slice in the context of debugging [35]. Since then program slicing has found many applications besides debugging [17,24,27,31] including program integration [23], comprehension [14,18] and reuse [3,12]. There also has been an active body of work in computing various types of slices resulting in a rich nomenclature for classifying slicing algorithms: intraprocedural vs. interprocedural; static vs. dynamic; backward vs. forward; executable vs. non-executable; and syntax-preserving vs. amorphous [6,7,13,21,33].

In short, intraprocedural slices consider a single procedure in isolation, while interprocedural slices consider multiple procedures with procedure calls. Static slices are computed from a program using static analysis while dynamic slices are computed from a program and an input and thus take into account a single execution of the program. A backward slice identifies program components that might affect a given computation. Its dual, a forward slice, identifies program components affected by a given component. An executable slice is an executable program that captures a subset of the original program's computation, while a non-executable (or closure) slice simply identifies the elements that affect (or are affected by) a given computation. These are often the same, but not always [4]. Finally, a syntax-preserving slice contains only portions of the original program's text, while an amorphous slice allows semantics-preserving transformations [18].

Slicing has found many applications because it allows the programmer to focus on a sub-computation; extracting it in the form of an executable subprogram—the slice. The sub-computation of interest may be one that the original author of the program had not considered and so the computation which denotes it may be arbitrarily scattered throughout the source code of the original program. The task of constructing the slice is thus the task of locating these scattered components and the supporting computations upon which they depend. It is a demanding problem because

* Corresponding author. Fax: +44 1895 251 686.
E-mail address: mark@dcs.kcl.ac.uk (M. Harman).

0950-5849/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.06.001

it requires a deep semantic analysis in order to ensure that the slice extracted preserves the behaviour of the original program with respect to the computation of interest.

The problem of slicing *unstructured* programs is important because many of the applications of slicing involve maintenance, evolution, and re-engineering of legacy systems, often written in programming styles which make heavy use of unstructured control flow [3,9,10,26]. Even recent systems contain a significant proportion of `goto` statements. For example, an inspection of Linux Kernel version 2.6.8.1 revealed that approximately 0.86% of the statements are `goto` statements. Finally, some programming languages (e.g. `C`), require the use of `break` statements in common constructions (such as the `switch` statement). The `break` statement denotes a limited form of unstructured control flow.

Ottenstein's Program Dependence Graph (PDG) based algorithm is currently the best known algorithm for intraprocedural slicing of structured programs [15,28]. This algorithm was not designed to compute slices of unstructured programs. Consequently, it fails to include any `goto` statements in a slice because a `goto` statement is neither the source of data nor control dependence. The literature contains several algorithms that extend Ottenstein's algorithm to compute slices of unstructured programs [1,2,11,20,25]. These algorithms are discussed in Section 6.

This paper focuses on the computation of static, backward, intraprocedural, executable slices of unstructured programs, henceforth simply referred to as slices. It makes the following contributions:

(1) Framework: The paper introduces the framework shown in Fig. 1 for classifying slicers of unstructured programs along three independent dimensions: termination behaviour, syntactic structure, and size. It is shown that slicing algorithms for two of the eight combinations within the framework, though desirable, simply do not exist. This non-existence result is not due to the familiar non-computability results relating to slice minimality [35]. Rather, it is a direct result of the particular properties of unstructured programs and their slices.
(2) Slicing algorithms: The paper presents slicing algorithms[1] for the remaining six combinations. These algorithms are built using common data structures, thereby facilitating examination of the tradeoffs present between the different possibilities. Finally, existing algorithms for slicing unstructured programs are placed into the framework. Interestingly, this reveals that, of the six possibilities, only three have been considered in previous slicing literature.

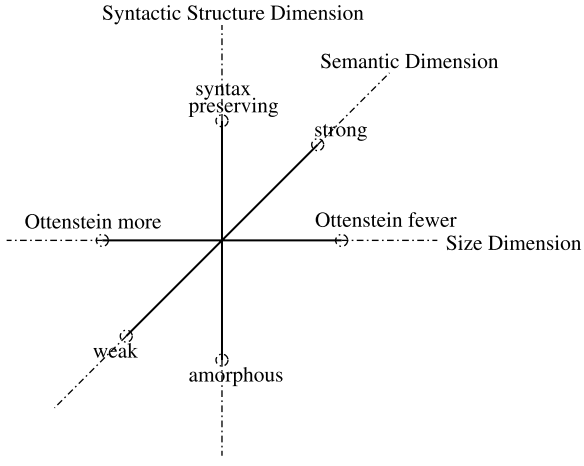

Fig. 1. The three dimensional framework.

The rest of the paper is organized as follows. Section 2 contains some necessary definitions. Section 3 presents the three-dimensions of the framework. Section 4 proves that two classes within the framework do not exist. Section 5 presents the new slicing algorithms. Section 6 discusses related work, places it into the framework, and compares it to the algorithms from Section 5. Finally, conclusions are presented in Section 7.

2. Definitions

This section defines the properties of the abstract syntax trees, control-flow graphs, and program dependence graphs required in subsequent sections. The language considered is essentially C, however the focus of the paper is on intraprocedural control issues; thus, the definitions and examples consider primarily assignment, `if-then-else`, `while`, `do-while`, sequence, `goto`, and 'special' statements. Interprocedural control issues (e.g. those introduced by `exit()`, `longjmp()`, or exceptions), and pointers, arrays, and other data dependence related features are mostly ignored.

2.1. Abstract Syntax Tree

The Abstract Syntax Tree (AST) is used to treat issues related to the order of statements in a program. This section defines the AST and two operators used to relate the ASTs of programs and their slices.

Definition 1. (Abstract Syntax Tree [16]). Each procedure P is represented by a standard Abstract Syntax Tree, denoted by AST(P). The relevant core of which is described as follows wherein bold text indicates the node kind (lower case) and contents (upper case), subordinate nodes appear

[1] The algorithms focus on the issue of unstructuredness and so issues to do with other language features (for instance pointer aliasing) are not considered.