Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 45–51
Information Processing Letters
www.elsevier.com/locate/ipl

Slicing for modern program structures: a theory for eliminating irrelevant loops

Torben Amtoft [1,2]

Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA

Received 1 July 2007; received in revised form 17 September 2007; accepted 2 October 2007

Available online 7 October 2007

Communicated by G. Morrisett

Abstract

Slicing is a program transformation technique with numerous applications, as it allows the user to focus on the parts of a program that are relevant for a given purpose. Ideally, the slice program should have the same termination properties as the original program, but to keep the slices manageable, it might be preferable to slice away loops that do not affect the values of relevant variables. This paper provides the first theoretical foundation to reason about non-termination *in*sensitive slicing *without* assuming the presence of a unique end node. A slice is required to be closed under data dependence and under a recently proposed variant of control dependence, called *weak order dependence*. This allows a simulation-based correctness proof for a correctness criterion stating that the observational behavior of the original program must be a *prefix* of the behavior of the slice program.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Program slicing; Control dependence; Observable behavior; Simulation techniques; Programming languages; Compilers

1. Introduction

Program slicing [12,11] has been applied for many purposes: compiler optimizations, debugging, model checking, protocol understanding, etc. Given the control flow graph (CFG) of a program and given a *slicing criterion*, the sets of nodes of interest, the following steps are involved in slicing:

(i) compute the *slice*, a set of nodes which includes the slicing criterion as well as those nodes that the slicing criterion depends on (directly or indirectly, with respect to data or with respect to control);
(ii) create the *slice program*, essentially by removing the nodes that are not in the slice.

One way to express the correctness of slicing is to demand that the observable behavior of the slice program is "similar" to the observable behavior of the original program. If "similar" implies that one is infinite exactly when the other is, as is often required in applications such as model checking, the slice must include all nodes that may influence the guards of potential loops. This can be achieved by weakening the "control dependence" relation, but the resulting slices may be so large that they

E-mail address: tamtoft@cis.ksu.edu.
URL: http://people.cis.ksu.edu/~tamtoft.
[1] Supported in part by AFOSR and by Rockwell Collins.
[2] The author would like to thank John Hatcliff and Venkatesh Prasad Ranganath for many interesting discussions and for comments on a draft of this paper, and also several anonymous referees for useful suggestions.

0020-0190/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2007.10.002

are less useful in applications such as program comprehension.

Most previous work on the theoretical foundation of slicing [2,5] assumes that the underlying CFG has a unique end node. This restriction prevents a smooth handling of control structures where methods have multiple exit points, or—more importantly—zero exit points (as in case of indefinitely running reactive systems). As reported in [9,10], the author was part of work that investigated notions of control dependencies suitable for handling arbitrary CFGs. The main result was to propose one control dependence ($\overset{ntscd}{\rightarrow}$) designed to preserve termination properties, and another ($\overset{nticd}{\rightarrow}$) which allows the termination domain to increase; both coincide with classical definitions on CFGs with unique end nodes.

In [9] it is shown, using (weak) bisimulation as is known from concurrency [7] and first proposed for slicing purposes (for multi-threaded programs) in [4], that slicing based on $\overset{ntscd}{\rightarrow}$ preserves observable behavior, in particular termination, provided the CFG is reducible (with or without a unique end node). To handle also irreducible CFGs (as is needed to model state charts), [10] proposed several notions of "order dependence", like $\overset{dod}{\rightarrow}$ ("decisive") and $\overset{wod}{\rightarrow}$ ("weak"), and proved that for an *arbitrary* CFG, slicing based on $\overset{ntscd}{\rightarrow}$ and on $\overset{dod}{\rightarrow}$ preserves observable behavior. Still, [9,10] does *not* attempt to prove the correctness (modulo termination properties) of slicing based on definitions like $\overset{nticd}{\rightarrow}$.

The main contribution of this paper is to provide a result yet missing in the literature: a provably correct slicing technique which is able to handle arbitrary CFGs, including those needed to model reactive systems (and/or state charts), and which allows loops not influencing relevant values to be sliced away.

Our approach explores the abovementioned notion of weak order dependence, to be motivated in Section 3 which argues that it also captures control dependence. Its key virtue is to ensure that each node has a unique "next observable", with an *observable* being a node relevant to the slicing criterion. This allows (Section 4) a crisp correctness proof, establishing a (one-way) simulation property which states that if the original program can do some observable action then so can the slice program. (The reverse does not hold, as the original program may get stuck in some unobservable loop.) First we briefly summarize concepts important to program slicing, most of which are standard (see, e.g., [8,2]) but with a twist similar to [9,10].

2. Standard definitions

A control flow graph G is a labeled directed graph, with nodes representing statements in a program, and with edges representing control flow. A node is either a *statement node*, having at most one successor, or a *predicate node*, having two successors—one labeled T, and another labeled F. There is a distinguished *start node*, with no[3] incoming edges, from which all nodes are reachable. An *end node* is a node with no outgoing edges; if there is exactly one end node n_e and n_e is reachable from all other nodes, we say that G satisfies *the unique end node property*.

To relate a procedure (method) body to its CFG we use a "code map" *code* that maps each CFG node to the code for the corresponding program statement. The function *def* (*ref*) maps each node to the set of program variables defined (referenced) at that node. For example, a statement that branches on the boolean expression B is represented as a predicate node n with $code(n) = B\mathbf{?}$ and $def(n) = \emptyset$.

A *path* π in G from n_1 to n_k, written as $[n_1..n_k]$, is a sequence of nodes $n_1, n_2, \ldots, n_k$ where for all $i \in 1 \ldots k-1$, G contains an edge from n_i to n_{i+1}; here k ($\geqslant 1$) is the length of the path which is non-trivial if $k > 1$. If there is a path of length k from n to m, but no shorter path, we write $dist^G(n, m) = k$.

To illustrate the standard notions of dependence, consider the CFGs in Fig. 1 which all have a unique end node e, chosen as our slicing criterion.

In (I), e is *data dependent* on b, according to

Definition 1. Node n is data dependent on m, written $m \overset{dd}{\rightarrow} n$, if there exists a variable v such that $v \in def(m) \cap ref(n)$, and there exists a non-trivial path $[n_1..n_k]$ with $n_1 = m$ and $n_k = n$ where for all $i \in 2 \ldots k-1$, $v \notin def(n_i)$.

But neither b or e are data dependent on a or c which thus are irrelevant to the slicing criterion, so we may safely update *code* so that $code(a) = code(c) = \mathbf{skip}$. The resulting program and the original program behave identically on e.

In (II), it holds that $b \overset{dd}{\rightarrow} e$ so the slice must include b —and also the predicate node a, since otherwise slicing would update $code(a)$ to either *true***?** or *false***?**, causing b to be possibly either improperly executed or improperly bypassed; in all cases, y might end up with a wrong

[3] To save space, not all our examples satisfy this, but they can easily be transformed so as to do.

Download English Version:

https://daneshyari.com/en/article/427976

Download Persian Version:

https://daneshyari.com/article/427976

Daneshyari.com