Dependence Analysis Based on Dynamic Slicing for Debugging

Chen Zhen-qiang, Xu Bao-wen
1. Department of Computer Science & Engineering, Southeast University, Nanjing 210096, China; 2. State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China

Abstract: Dynamic program slicing is an effective technique for narrowing the errors to the relevant parts of a program when debugging. Given a slicing criterion, the dynamic slice contains only those statements that actually affect the variables in the slicing criterion. This paper proposes a dynamic slicing method based on static dependence analysis. It uses the program dependence graph and other static information to reduce the information needed to be traced during program execution. Thus, the efficiency is dramatically improved while the precision is not depressed. The slicing criterion is modified to fit for debugging. It consists of filename and the line number at which the statement is.

Key words: program slice; slicing criterion; program dependence graph

CLC number: TP 311

0 Introduction

Program slicing[1-2] is an effective technique for narrowing the focus of attention to the relevant parts of a program. It is widely used in various software engineering activities such as program understanding, testing, debugging, maintenance, and complexity measurement[3-4]. A slice consists of statements and predicates that have influence on the variables at a program point. There are two kinds of slicing methods: static slicing and dynamic slicing. Static slicing computes slices by taking all the possible executions into consideration, and dynamic slicing is with respect to the given input sets. Static slicing is mainly used in program understanding and maintenance because the information is global. A dynamic slice contains fewer statements than the static one, thus it is useful for debugging and localizing errors[9].

In this paper, we focus our attention on the dynamic slicing methods. The main difficulty of dynamic slicing is that to get the running-time information, the execution of the program has to be traced. It is expensive. Therefore techniques, that improve the efficiency and reduce the size of dynamic slice, are needed. Different dynamic slicing methods have been proposed in literature[3-5, 8-10]. The major shortcoming is either imprecision or inefficiency.

We develop a new dynamic slicing method based on dependence analysis. It uses the program dependence graph and other static information to reduce the information needed to be traced during program execution. Thus the efficiency is greatly improved, while the precision is not depressed.

The remainder of this paper is organized as follows. Section 1 provides an overview of the basic notions of dependence graph and previous dynamic methods. Section 2 provides a detail description of our dynamic slicing criterion. Our dynamic slicing algorithm is presented in Section 3. An example is given in Section 4. Section 5 shows how to apply our dynamic slicing method to program debugging. Conclusion remarks are given in Section 6.
1 Basic Notions and Related Work

This section introduces the program dependence graph and the related notions. These are the bases of our slicing method, although they are static.

1.1 Program Dependence Graph

Program dependences are dependence relationships between statements in a program that are implicitly determined by the control and data flows in the program. Program dependence analysis is an analysis technique to identify and determine various program dependences in program source codes and then represent them in some explicit forms convenient for various applications\cite{2,7,11,14}. There are two types of dependence: data dependence and control dependence.

Informally, for two statements $s_1$ and $s_2$, if the execution of $s_1$ influences the execution of $s_2$, then $s_2$ is control dependent on $s_1$, denoted by $CD(s_2, s_1)$; if $s_1$ defines a variable and $s_2$ uses it, then $s_2$ is data dependent on $s_1$, denoted by $DD(s_2, s_1)$.

The PDG of procedure $P$ is a directed graph, $G = (S_S, S_E)$, where $S_S$ are statements or predicate expressions, $S_E$ is an edge set, in which each edge $e = (s_1, s_2)$ represents $DD(s_1, s_2)$ or $CD(s_1, s_2)$.

PDG describes the dependences in a single subprogram. To model the dependences among subprograms, system dependence graph (SDG) is introduced\cite{2}. A SDG consists of a set of procedure dependence graphs each representing a procedure. Nowadays, most dependence analysis approaches are based on control flow graph (CFG). CFG is a common way to represent programs. It represents the execution orders of statements and modules. The CFG of a procedure is a direct graph, $G = (S, E, s_{entry}, s_{exit})$, where $S$ are statements or predicate expressions, $E$ is an edge set, in which each edge $e = (s_1, s_2)$ represents $\langle s_1, s_2 \rangle$ and $s_2$ might be executed just after the execution of $s_1$. $s_{entry}$ is the entry node of $P$, and $s_{exit}$ is the exit node of $P$.

CFG is a static representation for program, the path in a CFG might not be an actual executing path. Thus in dynamic analysis, execution history is often introduced to represent the execution path\cite{16}.

1.2 Execution History and Slicing Criterion

An execution history is a feasible path in the CFG that has actually been executed, denoted by $EH$. The statements contained in the execution history are in the same order as they have been executed. As the example shown in Fig. 1, let the input be $a = 1, c = 3$, the corresponding execution history is $EH_1(1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 7, 8, 9, 11, 12, 7, 13)$. $EH(j)$ gives the serial number of the statement executed at jth step of $EH$, such as $EH_1(17) = 13$. To distinguish the different occurrence of the same statement in the execution history, a number pair $(i, j)$, denoted by $i'$, is introduced, where $i$ is the serial number of the statement and $j$ is the step number of $i$ in the $EH$.

```
1) read(a)
2) read(c)
3) if $a > 0$ then
   4) $i = 2$
   5) $s = 0$
   else
   6) $i = 1$
   7) while $i <= c$ do
   8) if $i > 0$ then
  9) $x = 2$
   10) $x = s + 1$
   end if
  11) $s = s + x$
end if
 12) $i = i + 1$
 13) write(s)
```

Fig. 1 A sample program

Every program slice is based on a rule, named as slicing criterion. In static slicing, the slicing criterion has the form $(i, V)$ where $i$ is the serial number of a statement in a program point, $V$ is a variable set. Dynamic slicing criterion is a triple $(INPUT, i', V)$, where INPUT denotes the actual input during an execution of the program. When execution history is introduced, the dynamic slicing criterion will be $(i', EH, V)$, where $EH$ is the execution history on a given input. $i'$ denotes the $i$ statement at position $j$ in the $EH$. If only the variables referred at $i$ are concerned, $V$ might be omitted. In the following discussion, $V$ is omitted. For the execution history $EH_1$ above, slicing criterion $C1=(137)$, $ER1$ means slicing on statement 13 in fact. After the slicing criterion is determined, dynamic slices will be computed by different kinds of slicing methods.

1.3 Related Dynamic Slicing Methods

There have been several dynamic slicing methods proposed in literature. In this section, only three methods related to our algorithm are fully discussed, i.e. mark nodes or edges\cite{10} methods.