Dynamic Slicing of Object-Oriented Programs

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
zhao@cs.fit.ac.jp

Abstract

Program slicing has many applications such as program debugging, testing, maintenance, and complexity measurement. A static slice consists of all statements in program $P$ that may affect the value of variable $v$ at some point $p$, and a dynamic slice consists only of statements that influence the value of variable occurrence for specific program inputs. In this paper, we concern the problem of dynamic slicing of object-oriented programs which, to our knowledge, has not been addressed in the literature. To solve this problem, we present the dynamic object-oriented dependence graph (DODG) which is an arc-classified digraph to explicitly represent various dynamic dependences between statement instances for a particular execution of an object-oriented program. Based on the DODG, we present a two-phase algorithm for computing a dynamic slice of an object-oriented program.

1 Introduction

Program debugging is the activity of analyzing the program to locate and correct errors in a program by reasoning about causal relation between bugs and the error detected in the program. Program debugging tools are essential for any programming environments. During the debugging process, once a symptom of error has been detected, we always hope to use some strategies to reduce the amount of code which can not have produced the error symptom. Such strategies are usually called filtering techniques [7].

The most important filtering technique is program slicing. Program slicing is the task of computing program slices which consist of the parts of a program that (potentially) affect the values computed at some point of interesting, referred to as a slicing criterion. The parts of a program which have a direct or indirect effect on the values computed at a slicing criterion are called the program slice with respect to the criterion. The original concept of a program slice was introduced by Weiser [13]. After that, a number of slightly different notions of program slices and a number of algorithms to compute slices have been proposed for imperative programs [1, 2, 6, 9, 12]. Program slicing can be divided into static slicing and dynamic slicing. Static slicing computes program slices through static data flow and control flow analysis and is valid for all possible executions of the program, whereas dynamic slicing computes slices through dynamic data flow and control flow analysis and is valid only for one set of input data to the program. While static slicing is mainly used in program understanding and software maintenance, dynamic slicing is particularly useful for program debugging and testing.

However, although a number of approaches have been proposed for slicing procedural programs, slicing object-oriented programs is just starting. Although researchers have extended the concept of program slicing to static slicing of object-oriented programs [6, 14, 16, 19, 21, 23], the dynamic slicing of object-oriented programs is still being missing until now.

Object-oriented programming languages present unique opportunities and problems for program analysis schemes such as slicing and testing. For example, to slice an object-oriented program, features such as dynamic binding, encapsulation, inheritance, message passing, and polymorphism must be considered carefully. Although the concepts of inheritance and polymorphism provide the great strengths of object-oriented programming languages, they also introduce difficulties in program analysis.

In this paper, we present the first algorithm for dynamic slicing of object-oriented programs. The main feature of the approach is to compute slices of an object-oriented program using a
dependence-based representation named dynamic object-oriented dependence graph (DODG). The DODG is an arc-classified digraph to explicitly represent various dynamic dependences between statement instances for a particular execution of an object-oriented program.

As the first attempt to study the dynamic slicing of object-oriented programs, our motivation is to build a powerful, yet efficient debugging tool for object-oriented programs by combining slicing technique as a main step to filter the program during bug location.

The rest of the paper is organized as follows. Section 2 introduces a motivation example. Section 3 describes some notions of dynamic slices of object-oriented programs. Section 4 presents the dynamic object-oriented dependence graph, and describes how to construct the graph. Section 5 shows how to find a dynamic slice of an object-oriented program. Concluding remarks are given in Section 6.

2 Motivation Example

We use a C++ program in Figure 1 as our target program. The program is taken from [16] and claimed to implement an elevator controller.

For the input data argv[1]=3 the program produces an incorrect output current_floor = 3, rather than 2, caused by the incorrect statement s12 which should have read return current_floor. By using the static slicing algorithm proposed by [16], we can obtain a static slice on the slicing criterion C = (s39, current_floor) which consists of the statements {e2, s3, s4, s5, s3, e11, s12, e15, s16, s17, c18, s19, c20, e21, s22, e24, e25, s26, e31, s32, c33, c14, s35, s36, s37, c38, s39}. The slice is shown in Figure 2 (a) in more detail. However, by carefully examining the execution trace of the program with input argv[1]=3, we can observe that:

1. Since statement s36 has not been executed, it can be removed from the slice.

2. Since s36 is a statement that creates an object of class AlarmElevator, it might call to the constructor of class AlarmElevator. Therefore statements e24, e25, s26 contained in the constructor can be removed also.

3. Since Statement s36 has not been executed, statement c38 will call only the method go() in class Elevator, and therefore statements c31, s32, s33 contained in method go() of class AlarmElevator can be removed.

4. Since the value of variable current_direction in statement s16 has its value UP, statements s19, c20 have not been executed. Therefore they can be removed from the slice.

Therefore, we can obtain a dynamic slice of the program that contains statements {e2, s3, s4, s5, e11, s12, e15, s16, s17, c18, e21, s22, c34, s37, c38, s39} shown in Figure 2 (b). The size of the resulting dynamic slice has been reduced sig-

Figure 1: A sample C++ program.
nificantly compared with its corresponding static slice. The above example shows that taking into account a particular program execution might significantly reduce the size of the slice. By applying dynamic analysis it is easier to identify those statements in an object-oriented program which do have influence on the variables of interest.

3 Dynamic Slices of Object-Oriented Programs

3.1 Preliminaries

A digraph is an ordered pair \((V, A)\), where \(V\) is a finite set of elements called vertices, and \(A\) is a finite set of elements of the Cartesian product \(V \times V\), called arcs, i.e., \(A \subseteq V \times V\) is a binary relation on \(V\). For any arc \((v_1, v_2) \in A\), \(v_1\) is called the initial vertex of the arc and said to be adjacent to \(v_2\), and \(v_2\) is called terminal vertex of the arc and said to be adjacent from \(v_1\). A predecessor of a vertex \(v\) is a vertex adjacent to \(v\), and a successor of \(v\) is a vertex adjacent from \(v\). A simple digraph is a digraph \((V, A)\) such that no \((v, v) \in A\) for any \(v \in V\).

An arc-classified digraph is an n-tuple \((V, A_1, A_2, \ldots, A_{n-1})\) such that every \((V, A_i)\) \((i = 1, \ldots, n - 1)\) is a digraph and \(A_i \cap A_j = \phi\) for \(i = 1, 2, \ldots, n - 1\) and \(j = 1, 2, \ldots, n - 1\). A simple arc-classified digraph is an arc-classified digraph \((V, A_1, A_2, \ldots, A_{n-1})\) such that no \((v, v) \in A_i\) \((i = 1, \ldots, n - 1)\) for any \(v \in V\).

A path in a digraph \((V, A)\) or an arc-classified digraph \((V, A_1, A_2, \ldots, A_{n-1})\) is a sequence of arcs \((a_1, a_2, \ldots, a_l)\) such that the terminal vertex of \(a_i\) is the initial vertex of \(a_{i+1}\) for \(1 \leq i \leq l - 1\), where \(a_i \in A_1(1 \leq i \leq l)\) or \(a_i \in A_1 \cup A_2 \cup \ldots \cup A_{n-1}(1 \leq i \leq l)\), and \(l \geq 1\) is called the length of the path. If the initial vertex of \(a_1\) is \(v_f\) and the terminal vertex of \(a_l\) is \(v_f\), then the path is called a path from \(v_f\) to \(v_f\), or path \(v_f - v_f\) for short.

The flow graph of an object-oriented program \(P\) is a digraph \((V, A)\) where \(V\) is the set of vertices that correspond to statements and control predicates, and \(A\) is the set of arcs between vertices in \(V\). If there is an arc from vertex \(u\) to vertex \(v\) it means that control can pass from vertex \(u\) to vertex \(v\) during program execution. A path is called feasible path if there exists input data which causes the path that has actually been executed for some input will be referred to as an execution trace. For example, \(c34, s35, s37, c2, s3, s4, s5, c38, c15, s16, s17, c18, c21, s22, s17, c18, c21, s22, s17, s39, c11, s12\) is the execution trace when the program in Figure 1 is executed on input data \(\text{argv[1]}=3\). This execution trace is presented in Figure 3 in a more detail. Note that we use 0, 1, 2, etc. contained in the brackets to distinguish between multiple occurrences of the same statement in the execution trace.

3.2 Dynamic Slices

Generally, dynamic slicing of an object-oriented program is similar to dynamic slicing of multi-procedural programs since both can be solved by interprocedural dynamic control-flow and data-flow analysis. However, due to the introduction of inheritance and dynamic binding in object-oriented programs, the process of tracing dependences in an object-oriented program becomes more complex than that in a procedural program.

In the following we informally define some notions of dynamic slicing of object-oriented programs.

- A slicing criterion for an object-oriented program is of the form \((s, v, t, i)\), where \(s\) is a statement in the program, \(v\) is a variable used at \(s\), and \(t\) is an execution trace of the program with input \(i\).

Notice that we restrict the slicing criterion to contain only a single variable \(v\) at a statement \(s\), rather than a set of variables since we can easily combine each slice with respect to a single variable of a statement to form a slice with respect to a set of variables of the statement.

- A dynamic slice of an object-oriented program on a given slicing criterion \((s, v, t, i)\) consists of all statements in the program that actually affected the value of a variable \(v\) at statement \(s\).

Note that our dynamic slice of an object-oriented program is not necessarily executable. This is in contrast to that presented in [13] which they defined a dynamic slice as an executable subprogram. For program debugging and testing, a non-executable dynamic slice can also supply enough information as an executable slice, but can be computed more easily.
class Elevator {
  public:
    int top_floor;
    Direction current_direction;
    protected:
      int current_floor;
      Direction current_direction;
      int top_floor;
  }

public:


Figure 2: A static slice (a) and a dynamic slice (b) on C = (s39, current_floor) of Figure 1.

4 The Dynamic Object-Oriented Dependence Graph

This section shows how to construct the dynamic object-oriented dependence graph of an object-oriented program on which dynamic slices can be computed efficiently.

To find a dynamic slice of an object-oriented program, we construct a dependence-based representation named dynamic object-oriented dependence graph (DODG) for a particular execution trace of the program. The DODG is an arc-classified digraph (V, A) where V is the multi-set of flow-graph vertices, and A is the set of arcs representing dynamic control dependences and data dependences between vertices.

Usually there are two types of dependence relationships between statements, i.e., control dependences and data dependences.

Control dependences represent control conditions on which the execution of a statement or expression depends. Informally, a statement u is directly control-dependent on the control predicate v of a conditional branch statement (e.g., an if statement or while statement) if whether u is executed or not is directly determined by the evaluation result of v.

Data dependences reflect the data flow between
statements and expressions. Informally a statement $u$ is directly data-dependent on a statement $v$ if the value of a variable computed at $v$ has a direct influence on the value of a variable computed at $u$.

Our construction of the dynamic object-oriented program dependence graph of an object-oriented program is based on dynamic analysis of control flow and data flow of the program, and similar to those for constructing dynamic dependence graphs for procedural programs [1]. However, to construct the DODG of an object-oriented program, we must consider specific features of object-oriented programming languages carefully.

For example, in a procedural program, a call statement usually regards to a statement that calls a procedure or a statement that has function application. However, in an object-oriented program, in addition to these two kinds of statements, we have to consider classes and their instances, objects, and dynamic bindings. Therefore, we should give a more broad meaning for what a call statement is in an object-oriented program. In this paper, we regard a call statement in an object-oriented program as one of the following statements:

- a statement that calls a free standing procedure,
- a statement that has function application,
- a statement that creates an object,
- a statement that invokes a method, or
- a statement that returns a value to its caller.

Using similar techniques proposed by Agrawal et. al. [1], we can solve the problem of representing a call statement in the DODG.

Figure 4 shows the DODG of the program in Figure 1 with respect to the execution trace in Figure 3.

5 Computing Dynamic Slices of Object-Oriented Programs

The notions of dynamic slices introduced in Section 3 give only some general views of dynamic slicing of object-oriented programs and do not tell us how to compute them. In this section, we refine those notions based on the DODG of object-oriented programs and present an algorithm to compute a dynamic slice of an object-oriented program based on its DODG. Our algorithm consists of two phases:

1. Computing a dynamic slice over the DODG of an object-oriented program,
2. Mapping the slice over the DODG to the source code to obtain a dynamic slice of the program.

In the following we describe some notions of dynamic slicing of an object-oriented program based on the DODG of the program. Let $P$ be an object-oriented program and $G = (V, A)$ be the DODG of $P$.

- A dynamic slicing criterion for $G$ is of the form $(v, s, t, i)$ where $v \in V$ representing a statement occurrence for a particular execution trace $t$ with input $i$ of $P$.

- The dynamic slice $DS_G(v)$ of $G$ on a given dynamic slicing criterion $(v, s, t, i)$ is a subset of vertices of $G$, $DS_G(v, s, t, i) \subseteq V$, such that for any $v' \in V$, $v' \in DS_G(v, s, t, i)$ if and only if there exists a path from $v'$ to $v$ in $G$.

Note that once we have constructed the DODG for the given execution trace, we can easily obtain the dynamic slice by using a usual depth-first or breadth-first graph traversal algorithm to traverse the DODG of the program by taking the vertex corresponding to the statement of interest as the start point of traversal.

However, the above description of a dynamic slice over the DODG of an object-oriented program is only a set of vertices of the DODG. Since our aim is to obtain a dynamic slice of an object-oriented program, we should map a vertex in the DODG to a statement of the program to obtain a dynamic slice of an object-oriented program. By simply defining a mapping function, we can obtain such a dynamic slice straightforwardly.

6 Concluding Remarks

We presented the first algorithm for dynamic slicing of object-oriented programs. The main feature of the approach is to compute slices of an object-oriented program using a dependence-based representation named dynamic object-oriented dependence graph (DODG). The DODG is an arc-classified digraph to explicitly represent various dynamic dependences between statement instances for a particular execution of an object-oriented program. Although here we presented the approach in term of C++, other versions of this approach for other object-oriented programming languages such as Java and Ada95 are easily adaptable because they share their basic execution mechanisms with C++.

Now we are developing a debugging envi-
environment for C++ programs in which the dynamic slicing technique has been used as a filtering technique to aid bug location during debugging.

References


