The Oberon Slicing Tool

Christoph Steindl
steindl@ssw.uni-linz.ac.at
Institute for Practical Computer Science
Johannes Kepler University Linz

Abstract

Program slicing is a novel program analysis technique. It is mainly used during debugging and reengineering. It tries to eliminate all parts from the program that are not currently of interest to the programmer. Therefore, the programmer clicks on a statement (which is of interest to him, e.g. because the value of a variable is wrong) and the program slicer highlights all program statements that might have produced the erroneous value. The demonstration will show the Oberon Slicing Tool in action and how the programmer can benefit from a programming environment that is flow-sensitive because it has knowledge of the control flow and data flow of the edited programs.

1 Introduction

We have implemented the Oberon Slicing Tool ([OST], [Ste98] and [Ste99a]). Program Slicing [Wei84] is a program analysis technique that reduces programs to those statements that are relevant for a particular computation. A slice provides the answer to the question "What program statements potentially affect the value of variable *v* at statement *s*?" Mark Weiser introduced program slicing because he made the observation that programmers have some abstractions about the program in mind during debugging. When debugging a program one follows the dependences from the erroneous statement *s* back to the influencing parts of the program. These statements may influence *s* (the slicing criterion) either because they decide whether *s* is executed or because they define a variable that is used by *s*. Program slicing computes these dependences automatically and thus assists the programmer in a lot of error prone tasks, such as debugging, program integration, software maintenance, testing, and software quality assurance.

We visualize the control flow and data flow information computed by the Oberon Slicing Tool with *Active Text Elements* ([Szy92], [MöKo96]). The editor framework of the Oberon System allows arbitrary objects derived from an abstract base class to be inserted into the text like ordinary characters. These so-called *text elements* are active because they react on mouse clicks. For example, a text element representing a hypertext link will cause the editor to scroll to another text position when the user clicks on the link. Other active text elements may support folding or annotating or they may represent a menu that pops up in reaction to a click.

2 Features of the Oberon Slicing Tool

The Oberon Slicing Tool computes program slices of Oberon-2 programs. There are no restrictions on the programs, i.e. they may use structured types (records and arrays), global variables of any type, objects on the heap. Functions may have arbitrary side-effects. Procedures may be nested. There may be recursion and dynamic binding due to type-bound procedures (methods) and procedure variables (function pointers). Finally, the program can consist of modules.

It is extremely fast. Slicing information for a module is computed within a few seconds. The slices are computed and visualized without any delay at all.

It uses static slicing. Static slicing derives the necessary information by inspection of the source code. The information must be valid for all possible executions of the program. Dynamic slicing derives the necessary information by monitoring the execution of the program. Therefore, the information is only valid for the particular execution but not in general. Static information is necessarily less precise than dynamic information. On the other hand, it can be computed once for all possible executions, whereas dynamic information must be computed again and again.

It can compute intraprocedural, interprocedural and intermodular slices. Intraprocedural slices are restricted to the procedure containing the slicing criterion. Interprocedural slices span over all involved

procedures, considering statically and dynamically bound procedure calls. Intermodular slicing handles calls of external procedures of imported modules precisely.

It computes backward slices. This means that the slices consist of all parts that may have contributed to the slicing criterion and not of all parts that may be influenced themselves by the slicing criterion.

It can handle procedural and object-oriented programs. Object-oriented programming languages have attracted more and more attention during the last years since they allow one to write programs that are more flexible, reusable and maintainable. However, the concepts of inheritance, dynamic binding and polymorphism represent new challenges for static program analysis. In [Ste99b] we show how we integrate the dynamic aspects of polymorphism and dynamic binding into static analysis.

It uses expression-oriented slicing. The parts of the program that are considered to be included into the slice are parts of expressions and not whole statements or even procedures. The derived slices are finer grained and therefore more precise than those derived by statement-oriented slicing.

It uses user-feedback to restrict the effects of aliases and dynamic binding. Static analysis is not immediately applicable to object-oriented programs. We have developed techniques to account for the run-time effects of dynamic binding and aliasing. Starting from conservative assumptions about the effects of dynamic binding and aliases, we use user feedback to restrict the dynamic effects. Data flow information can be re-computed, leading to more precise information. Only a few cycles of restricting and recomputing are usually necessary to model the use case that the programmer currently investigates.

It uses a repository to store the computed slicing information which can be re-used later when importing already sliced modules in order to handle calls of external procedures precisely.

It implements the Model-View-Controller concept (i.e. it is possible to have multiple views of the same slice that are kept consistent).

3 Visualization of Control Flow and Data Flow Information

3.1 Data Dependences

Data dependences are links from the usage of a variable to all definitions of the same variable that might reach the usage. Data dependences are visualized by small buttons (see Fig. 1) that behave like popup menus showing all reaching definitions. One can select a reaching definition which is then highlighted in the program source code. Potential problems (e.g. usage of a potentially uninitialized variable) are also made obvious to the programmer. In order to visualize data dependences, exact data flow information has to be computed. This is not possible by simply parsing the source code on a syntactical level. One has to pay attention to the smallest details of the programming language (like short-circuit evaluation of Boolean expressions, side-effects of functions within expressions, scoping, etc.).

3.2 Links Between the Call Site and the Called Procedure

The call sites and the called procedures are linked with bidirectional hypertext links (see Fig. 1). Again, one can select a call destination or call site from popup elements, leading to the respective procedure or call site being highlighted in the program source code. Direct and indirect recursion due to static and dynamic binding are handled. For dynamically bound procedure calls due to method calls or calls via procedure variables, all possible call destinations are computed.

3.3 Parameter Usage

For each parameter of a procedure, the information about its usage is computed and visualized properly. At each call site, a small arrow element is inserted before each actual parameter indicating the usage of the parameter within the called procedure: a downward arrow for input parameters whose value is used in the called procedure, an upward arrow for output parameters that are defined by the called procedure, an up-and-down arrow for parameters whose values are used and that are defined. In the formal parameter list of each procedure, even more precise information is available about each parameter. By clicking on the respective arrows, a description of potential alias problems and data flow problems is displayed. For dynamically bound procedure calls, the usage of each parameter is combined over all possible call destinations (see Fig. 2).

3.4 Additional Parameters

If a procedure accesses non-local variables that are not part of the parameter list, these variables are added to the (formal and actual) parameter lists as additional parameters (see Fig. 2). This makes the total set of possibly used and changed variables of a procedure call obvious to the programmer at one glance. If the programmer is not aware of the additional parameters he will often overlook these side effects.

3.5 Aliases and Dynamic Types

Two variables are aliases if they may denote the same memory cell. Not only unexperienced programmers are subject to problems with aliases, so it can be a big help to make potential aliases visible as early as possible. In object-oriented programming languages, pointers may refer to objects of different types. It is not always obvious to the programmer what dynamic types a variable may have. Therefore, we collect the set of potential dynamic types into another kind of popup elements. We have extended the notion of dynamic types also to procedure variables where the dynamic types represent the set of possibly called procedures (see Fig. 2).

4 Bidirectional Feedback

So far, we have described how the Oberon Slicing Tool visualizes the derived information, i.e. we have explained the flow of information from the program slicer to the programmer. However, the user can request the computation of a program slice with respect to some point of the program by a simple mouse-click e.g. on an assignment statement. All parts of the program that belong to the slice with respect to the statement and the variables used and defined by that statement are then visualized in a different color. Furthermore, the control flow and data flow information is visualized with active text elements that react to user input. The programmer can e.g., enable or disable some potential aliases if he knows (by some oracle) that two variables will never be aliases. This information is then re-used by the program slicer in order to compute more precise control flow and data flow information. Likewise, the programmer can enable and disable some of the potential dynamic types of polymorphic variables, therefore limiting the set of methods that may be called at polymorphic call sites with these polymorphic variables as receivers of the method calls. In Fig. 2, procedure *ForAll* contains a dynamically bound call with two possible call destinations (*Inc* and *PrintNode*). They are both initially enabled, leading to their additional parameters (namely parameter *count* of procedure *Inc*) being shown at the call site. When the programmer disables the call destination *Inc*, control and data flow information are computed only for the restricted set of possible call destinations. Only the relevant additional parameters (i.e. parameter *head*) are shown.

Bibliography

[MöKo96] Hanspeter Mössenböck, Kai Koskimies: *Active Text for Structuring and Understanding Source Code*. SOFTWARE - Practice and Experience, 26(7), July 1996.

[OST] *Oberon Slicing Tool*. http://www.ssw.uni-linz.ac.at/General/Staff/CS/Research/Projects/ProgramSlicing/index.html

[Ste98] Christoph Steindl: *Intermodular Slicing of Object-Oriented Programs*. International Conference on Compiler Construction, LNCS 1383, Springer-Verlag, 1998.

[Ste99a] Christoph Steindl: *Program Slicing for Object-Oriented Programming Languages*. Dissertation, Johannes Kepler University Linz, 1999.

[Ste99b] Christoph Steindl: *Static Analysis of Object-Oriented Programs*. 9th ECOOP Workshop for PhD Students in Object-Oriented Programming, Lisbon, Portugal, June 1999.

[Szy92] Clemens Szyperski: *Write-ing Applications: Designing an Extensible Text Editor as an Application Framework*. Proceedings of the 7th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'92), Dortmund, Germany, Prentice Hall, March 1992.

[Wei84] Mark Weiser: *Program Slicing*. IEEE Transactions on Software Engineering, 10(4), July 1984.

```
MODULE Test;

IMPORT In, Out;

CONST
   TAB = 09X;
TYPE
   Node = POINTER TO NodeDesc;
VAR
   head: Node; global: INTEGER;
TYPE
   NodeDesc = RECORD i: INTEGER; next: Node END ;

   PROCEDURE Push (↓ i: INTEGER    (* VAR ↕ head: Node *)); called at
      VAR n: Node;
   BEGIN                                  data dependences
      NEW(n);
      n.i := i DDs;
      n.next := head DDs;
      head := n DDs
   END Push;

   PROCEDURE Insert* ((* VAR ↕ head: Node; VAR ↓In.Done: BOOLEAN *));
      VAR i: INTEGER;
   BEGIN
      In.Open calling;                                  bidirectional
      REPEAT                                            links between
         In.Int(↕i DDs!) calling;                       call site and
         IF In.Done DDs THEN                            destination
            Push(↓i DDs!  (* ↕head DDs *)) calling
         END
      UNTIL ~In.Done DDs
   END Insert;

   PROCEDURE Show* ((* VAR ↓ head: Node *));
      VAR n: Node; i: INTEGER;
   BEGIN
      n := head DDs;                          two reaching
      i := 1;                                 definitions (one for
      WHILE n DDs # NIL DO                    first iteration and
         Out.String(↓"Position ") calling;    one for all successive
         Out.Int(↓i DDs, ↓0) calling;         iterations)
         Out.String(↓": ") calling;
         Out.Char(↓TAB) calling;
         Out.Int(↓n DDs.i, ↓0) calling;
         Out.Ln calling;
         n := n DDs.next
      END
   END Show;

END Test.
```

Fig. 1 - Data dependences and links for procedure calls

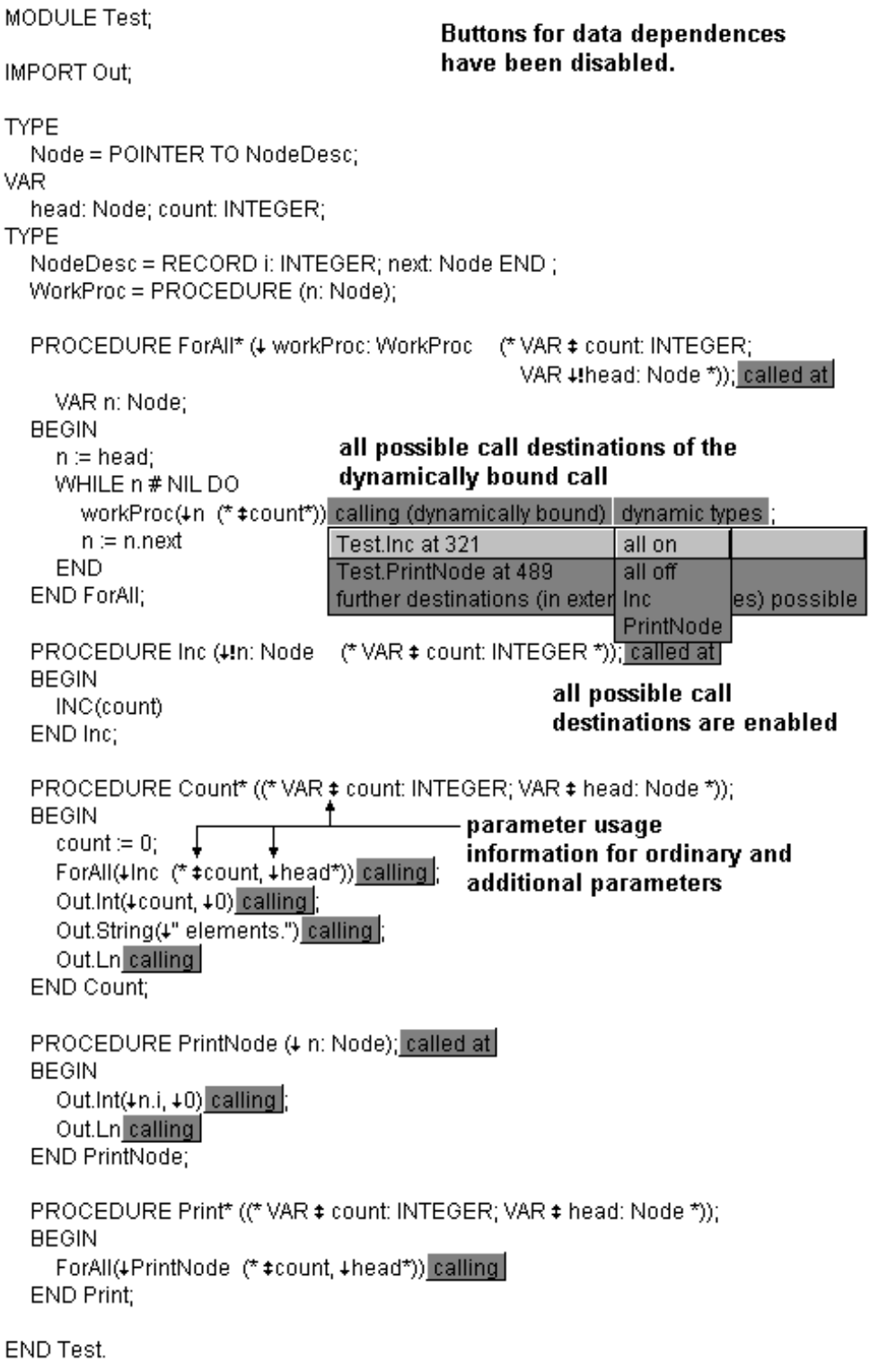

Fig. 2 - Parameter usage, additional parameters, dynamic types