Program Slicing for Object-Oriented Programming Languages

A dissertation submitted to the

JOHANNES KEPLER UNIVERSITY LINZ

for the degree of
Doctor of Technical Sciences

presented by
Dipl.-Ing. Dipl.-Ing. Christoph Steindl
Johannes Kepler University Linz

accepted on the recommendation of
Prof. Dr. H. Mössenböck, examiner
Prof. Dr. T. Gyimóthy, co-examiner

1999

Program Slicing for Object-Oriented Programming Languages

Christoph Steindl

Copyright (c) Christoph Steindl, 1999

Program Slicing for Object-Oriented Programming Languages

A dissertation submitted to the

JOHANNES KEPLER UNIVERSITY LINZ

for the degree of
Doctor of Technical Sciences

presented by
Dipl.-Ing. Dipl.-Ing. Christoph Steindl
Johannes Kepler University Linz

accepted on the recommendation of
Prof. Dr. H. Mössenböck, examiner
Prof. Dr. T. Gyimóthy, co-examiner

1999

Acknowledgements

I want to thank my advisor Prof. H. Mössenböck for a liberal supervision of this project and for his ongoing encouragement and patience. The Oberon System was an excellent working tool and an appropriate base for the work presented in this thesis. Markus Hof and David Parsons proof-read earlier versions of this thesis and provided valuable comments and improvements. Last but not least, I wish to thank my colleagues at the department, my friends in Linz, my parents Grete and Alois, and my sisters Kathi and Ulli for their steady encouragement and help.

Contents

Abstract viii
Kurzfassung ix

1 Introduction 1
1.1 Motivation 1
1.2 Goals 2
1.3 Outline 2

2 Background Information 5
2.1 Oberon-2 5
2.2 Control Flow 7
2.2.1 Control Flow Graphs 7
2.2.2 Dominator and Post-dominator Trees 8
2.2.3 Control Dependences 10
2.3 Data Flow 12
2.3.1 Data Dependences 12
2.3.2 Computation of Used and Defined Variables 14
2.3.3 Computation of Reaching Definitions 16
2.4 Program Slicing 21
2.4.1 Variants of Program Slicing 22
2.4.2 Applications 26

3 Current Slicing Algorithms 31
3.1 Slicing as a Data Flow Problem 31
3.2 Slicing as a Graph-Reachability Problem 34
3.2.1 Program Dependence Graph 35
3.2.2 System Dependence Graph 35
3.2.3 Computation of Summary Edges 42
3.2.4 Enhancing Slicing Accuracy 45

4 Implementation 47
4.1 Overview 47
4.2 Algorithm 48
4.3 Data Structures 49
4.4 Computation of Control Flow Information 54
4.5 Computation of Data Flow Information 66
4.5.1 Computation of Used and Defined Variables 67
4.5.2 Computation of Reaching Definitions 78

4.6 Slicing 99
4.6.1 Intraprocedural Slicing 100
4.6.2 Interprocedural Slicing 101
4.6.3 Intermodular Slicing 102
4.7 Support of Object-Oriented Features 105
4.8 Modularization 106
4.8.1 Module Repository 107
4.8.2 Module Slicer 109

5 User Interface 113
5.1 Visual Elements 113
5.1.1 Bidirectional Links Between the Caller and the Callee 113
5.1.2 Data Dependences 115
5.1.3 Parameters 116
5.1.4 Aliases 119
5.1.5 Dynamic Types 120
5.2 User Feedback 124
5.3 Module SlicerFE 125
5.4 Model-View-Controller Concept 127

6 Comparison 129
6.1 Chopshop 129
6.2 Ghinsu 129
6.3 Spyder 130
6.4 Unravel 130
6.5 VALSOFT 131
6.6 Wisconsin Program-Slicing Project 131

7 Conclusions 133

8 Future Work 137
8.1 Integration into the Programming Environment 137
8.2 Other Variants of Slicing 138
8.3 Software Metrics 138

Appendix: Additional Module Definitions 141

Bibliography 151

Curriculum Vitae 157

Abstract

Program slicing 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* 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.

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.

The result of this thesis is the Oberon Slicing Tool, a fully operational program slicing tool for the programming language Oberon-2. It integrates state-of-the-art algorithms and applies them to a strongly-typed object-oriented programming language. It extends them to support intermodular slicing of object-oriented programs. Control and data flow analysis considers inheritance, dynamic binding and polymorphism, as well as side-effects of functions, short-circuit evaluation of Boolean expressions and aliases due to reference parameters and pointers. The algorithm for alias analysis is fast but effective by taking into account information about the type of variables and the place of their declaration. The result of static program analysis is visualized with active text elements: hypertext links connect the call sites with the possible call destinations, parameter information elements indicate the direction of data flow at calls. Since static program analysis must make conservative assumptions about actual program executions, the sets of possible aliases and call destinations due to dynamic binding are more general then necessary. We visualize these sets and allow the programmer to restrict them via user interaction. These restrictions are then used to compute more precise control and data flow information. In this way, the programmer can limit the effects of aliases and dynamic binding and bring in his knowledge about the program into the analysis.

Kurzfassung

Program Slicing ist eine Programmanalysetechnik, die Programme auf jene Anweisungen reduziert, die für eine bestimmte Berechnung relevant sind. Ein Slice ist die Antwort auf die Frage: "Welche Anweisungen im Programm können den Wert der Variable *v* bei der Anweisung *s* beeinflussen?" Mark Weiser erfand Program Slicing, weil er beobachtete, dass sich Programmierer während der Fehlersuche Gedanken über die Beziehungen zwischen Programmteilen machen. Bei der Fehlersuche verfolgt man solche Beziehungen von der fehlerhaften Anweisung *s* zurück zu den Programmteilen, die sich auf *s* auswirken. Diese Anweisungen können *s* beeinflussen, indem sie entscheiden, ob *s* ausgeführt wird, oder indem sie einer Variablen einen Wert zuweisen, der von *s* verwendet wird. Program Slicing berechnet diese Beziehungen automatisch und unterstützt dadurch den Programmierer bei vielen fehleranfälligen Tätigkeiten, wie Fehlersuche, Integration von Programmversionen, Software-Wartung, Testen und Software-Qualitätssicherung.

Objektorientierte Programmiersprachen haben sich in den letzten Jahren immer mehr durchgesetzt, da sie es erlauben, Programme zu schreiben, die flexibler, besser wiederverwendbar und besser wartbar sind. Die Konzepte der Vererbung, der dynamischen Bindung und des Polymorphismus stellen allerdings für die Programmanalyse neue Herausforderungen dar.

Das Ergebnis dieser Doktorarbeit ist das Oberon Slicing Tool, ein voll funktionsfähiges Werkzeug für das Slicen von Oberon-2 Programmen. Es kombiniert Algorithmen, die dem Stand der Technik entsprechen, und wendet sie auf eine objektorientierte Programmiersprache mit strenger Typprüfung an. Es erweitert sie, um intermodulares Slicen von objektorientierten Programmen zu unterstützen. Die Kontroll- und Datenflussanalyse berücksichtigt Vererbung, dynamische Bindung und Polymorphismus, sowie Nebeneffekte von Funktionen, Kurzschlussauswertung von Booleschen Ausdrücken und Aliase aufgrund von Referenzparametern und Zeigern. Der Algorithmus für die Analyse von Aliasen ist schnell, aber trotzdem effektiv, indem er Information über den Typ von Variablen und den Ort ihrer Deklaration in Betracht zieht. Die Ergebnisse der statischen Programmanalyse werden durch aktive Textelemente dargestellt: Hypertext-Verknüpfungen verbinden Prozeduraufrufe mit den möglichen Aufrufzielen, Parameter-Informations-Elemente zeigen die Richtung des Datenflusses bei Aufrufen an. Da bei statischer Programmanalyse konservative Annahmen über die tatsächlichen Programmabläufe gemacht werden müssen, sind die Mengen der möglichen Aliase und Aufrufziele aufgrund von dynamischer Bindung allgemeiner, als sie sein müssten. Wir stellen diese Mengen dar und erlauben dem Programmierer, die Mengen durch Benutzerinteraktion einzuschränken. Die Restriktionen werden anschließend verwendet, um genauere Kontrollund Datenflussinformation zu berechnen. Auf diese Weise kann der Programmierer die Auswirkungen von Aliasen und dynamischer Bindung einschränken und sein Wissen in die Analyse einbringen.

1 Introduction

1.1 Motivation

Program slicing [Wei84] is a program analysis and reverse engineering technique that reduces a program to those statements that are relevant for a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the value of variable *v* at statement *s*?"

Program slicing was introduced by Mark Weiser because he made the observation that programmers have some abstractions about the program in mind during debugging. The process of debugging consists of following dependences from the erroneous statement *s* back to the influencing parts of the program. These statements may influence *s* either because they decide whether *s* is executed at all (*control dependence*) or because they define a variable that is used by *s* (*data dependence*). A program slicer can be used to automatically compute and visualize the slice of the program with regard to the statement *s* and the variables used or defined at *s*. It allows the programmer to focus his attention on the statements that are part of the slice and that might therefore contribute to the fault. Additionally, the programmer sees any statements that are not part of the slice although he knows that they should be.

Program slicing can be used to assist the programmer in a lot of tedious and error prone tasks, such as debugging, program integration, software maintenance, testing, and software quality assurance. Several variants of program slicing have been proposed for these purposes, including static slicing, dynamic slicing, backward slicing, forward slicing, chopping, interface slicing, etc.

A survey of existing program slicing tools shows that most of them are written for the programming language C, some for COBOL and FORTRAN. Most of these program slicers have problems with dynamic binding which is a cornerstone of object-oriented programming. Neither do they address the concepts of inheritance and polymorphism. Another problem is the performance of the program slicers. Since program slicing is an interactive method intended to assist the programmer, the results should ideally be presented immediately. It is unsatisfactory to wait for several minutes before the slice can be viewed.

1.2 Goals

The purpose of this work is to investigate whether static program slicing can be efficiently implemented for an object-oriented language such as Oberon-2. Although Oberon-2 is a small language, it is powerful and we may encounter many difficulties. We used the following goals as guidelines for the implementation of the Oberon Slicing Tool:

- The program slicer should be able to analyze the entire language which includes user-declared data types, structured types (records and arrays), global variables, functions with side-effects, nested procedures, type extension, dynamic binding via type-bound procedures (also called methods) and procedure variables (also called function pointers), recursion, and modules.
- The object-oriented features of Oberon-2 such as inheritance, dynamic binding and polymorphism should be fully supported. Object-oriented programs make heavy use of dynamic binding and pointers which are both difficult to handle by static analysis. Necessary conservative assumptions shall be restricted by feedback from the user.
- The internal data structures of the program slicer should closely model the semantics of the program.
- The computation of the slices should be fast, but the resulting slices should still be as precise as possible.
- The program slicer should support slicing of modular systems. Information that has already been computed for a module should be reused when slicing dependent modules.
- The program slicer should be an interactive tool. It should visualize all the information that has been computed during slicing and that could be useful for the programmer in order to understand the program.
- As the main fields of application of the program slicer we envisage assistance to the programmer, debugging, code understanding, maintenance, program testing and software metrics.

1.3 Outline

Chapter 2 gives an overview of the programming language Oberon-2, some background information about the computation of control flow and data flow as well as an overview of program slicing and a survey of the variants and applications of program slicing.

Chapter 3 describes current slicing algorithms together with their data structures, ranging from the original approach where slicing is seen as a data flow problem to the state-of-the-art where slicing is seen as a graph-reachability problem.

Chapter 4 describes the implementation of the Oberon Slicing Tool, its data structures and the algorithms used for the computation of control flow and data flow information as well

as for slicing itself.

Chapter 5 describes the user interface of the Oberon Slicing Tool, its visual elements and how user feedback is used to bridge the gap between static and dynamic slicing.

Chapter 6 compares the Oberon Slicing Tool with existing program slicers.

Chapter 7 gives a summary of the contributions of this thesis.

Chapter 8 outlines some areas for future work.

2 Background Information

Since significant parts of this thesis refer to the programming language Oberon-2, Section 2.1 will briefly summarize its features. Then we will concentrate on the main problem when implementing a program slicing tool: the construction of an intermediate representation of the program that closely models its semantics. The flow of control and the flow of data are the two main concepts for modeling the semantics of a program. In sections 2.2 and 2.3 we will give an overview of the techniques that have been used to model the flow of control and the flow of data. In Section 2.4 we will give an overview of program slicing and a survey of the variants and applications of program slicing.

2.1 Oberon-2

Oberon-2 [MöWi91] is a general-purpose programming language in the tradition of Pascal and Modula-2 with block structure, modularity, separate compilation, static typing with strong type checking (also across module boundaries), type extension (object-orientation with single inheritance) and type-bound procedures (methods). In the following subsections we will give an overview of various language constructs.

Language constructs for structured control flow

There are three language constructs to express selection and three to express iteration:

- The IF statement for conditional execution of statement sequences.
- The CASE statement for the selection and execution of a statement sequence according to the value of an expression.
- The WITH statement for the execution of a statement sequence depending on the result of a run-time type test. The tested type is applied to every occurrence of the tested variable within the guarded statement sequence.
- The WHILE statement for the repeated execution of a statement sequence while a condition (specified as a Boolean expression, the guard of the loop) is satisfied.
- The REPEAT statement for the repeated execution of a statement sequence until a condition specified by a Boolean expression is satisfied.
- The FOR statement for a fixed number of executions of a statement sequence while an integer variable is incremented in every iteration.

Language constructs for unstructured control flow

There are three language constructs for moderately unstructured control flow:

- The LOOP statement for the repeated execution of a statement sequence with possibly multiple EXITs from the nested statement sequence.
- The EXIT statement for termination of the enclosing loop statement and continuation with the statement following that loop statement.
- The RETURN statement for the termination of a procedure (also specifying the return value of functions).

Language constructs for the declaration of user-declared data types

There are several language constructs for the declaration of user-declared data types:

- Predefined data types include numeric, Boolean and character types. Pointers can point to arrays and to records. References to objects may be polymorphic (i.e. they may point to an object whose type is an arbitrary extension of the pointer's static type. The object's type is then called the pointer's dynamic type.)
- Data types can be defined as arrays or records of other data types.
- Data types can be defined as extensions (subtypes) of other data types (single inheritance).
- Procedures can be associated with types (type-bound procedures, also called methods).

Language constructs for abstraction and stepwise refinement

There are several language constructs to support abstraction and stepwise refinement:

- A program (module) can be built out of procedures. Procedures can be (directly or indirectly) recursive, they can declare and use local procedures. Parameters of procedures can be passed by value or by reference. Procedures may return values but these values must not be arrays or records.
- A module defines its interface by exporting items such as constants, types, variables, and procedures. It can import other modules. Although modules are compiled separately, strong type-checking is performed across module boundaries.
- A module can have multiple entry points. In an interactive environment, these entry points (also called commands) can be activated directly by the user.

Further Remarks

Some further remarks are necessary to conclude the overview of the programming language Oberon-2:

- Short-circuit evaluation is used for Boolean expressions.
- Objects can be allocated on the heap with the predefined function NEW, they can also be allocated automatically on the stack or statically as global variables of modules.
- A garbage collector finds the blocks of memory that are not used any more and makes them available for allocation again.
- Run-time type tests and type guards can be used to perform safe casting.
- Modules can be loaded dynamically. The body of a module is guaranteed to be executed upon loading of the module.
- Reference parameters as well as pointers to dynamically allocated objects on the heap may introduce aliases.
- Procedure calls can be either statically bound or dynamically bound: ordinary procedure calls and super calls of methods can be bound statically, calls of type-bound procedures and calls via procedure variables must be bound dynamically.

2.2 Control Flow

In high-level languages, control structures (such as IF, WHILE and RETURN) express the flow of control. For example the Boolean expression of an IF decides which branch will be executed. These control structures can be translated into the conditional and unconditional jumps in low-level languages. Several data structures have been proposed to model the semantics of control flow at different levels of abstraction.

The sections about control flow graphs, dominator and post-dominator trees, and control dependences partly follow the explanations of Brandis [Bra95], Aho et al. [ASU86] and Ferrante et al. [FeOW87].

2.2.1 Control Flow Graphs

Control flow graphs [ASU86] have been used as a basis for data flow analysis and for many optimizing code transformations such as common subexpression elimination, copy propagation, and loop-invariant code motion. The definition of control flow graphs builds on the concept of basic blocks:

Definition: A *basic block* is a sequence of consecutive statements in which flow of control

enters at the beginning and leaves at the end without halt or possibility of branching except at the end. A basic block is either executed in its entirety or not at all.

Definition: A *control flow graph* is a directed graph whose nodes are basic blocks with a unique *entry* node *START* and a unique *exit* node *STOP*. There is a directed edge from node *A* to node *B* if control may flow from block *A* directly to block *B*. This is the case if the last statement in *A* is a branch to *B*, or when *B* is on the fall-through path from *A*. We assume that for any node *N* in the graph there exists a path from *START* to *N* and a path from *N* to *STOP*. If an edge is labeled *T* (or *F*), then the target node of the edge will be executed if the predicate at the origin of the edge evaluates to *TRUE* (or *FALSE*).

Fig. 2.1 shows a piece of source code with the corresponding control flow graph.

```
p := head; cnt := 0
WHILE p # NIL DO
   IF p.val > 0 THEN
      INC(cnt)
   END
   p := p.next
END
```

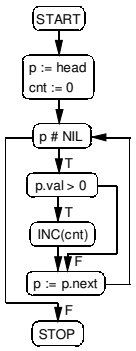

Fig. 2.1 - A piece of source code and its corresponding control flow graph

Control flow graphs accurately model the branching structure of the program and collate all statements between two branches into basic blocks. They can be built while parsing the source code with algorithms that have linear time complexity in the size of the program.

2.2.2 Dominator and Post-dominator Trees

Dominator trees represent the dominance relation between the nodes of directed graphs.

Definition: In a directed graph with entry node *START*, we say that a node *A dominates* node *B*, iff for all paths *P* from *START* to *B*, *A* is a member of *P*. *A* is called a *dominator* of *B*.

The dominance relation is

- reflexive: Each node dominates itself.
- transitive: If node *A* dominates node *B* and node *B* dominates node *C*, then node *A* dominates node *C*.
- anti-symmetric: If node *A* dominates node *B* and node *B* dominates node *A*, then node *A* must be equal to *B*.

Definition: We call *A* the *immediate dominator* of *B*, iff *A* is a dominator of *B*, *A* # *B*, and there is no other node *C* that dominates *B* and is dominated by *A*.

Definition: The *dominator tree* of a directed graph *G* with entry node *START* is the tree that consists of the nodes of *G*, has the root *START*, and has an edge between nodes *A* and *B* if *A* immediately dominates *B*.

Each node in the dominator tree has exactly one parent (except for the entry node *START*). All nodes being predecessors of some node *A* are dominators of *A*. If a basic block *A* dominates basic block *B*, *A* is on every path from *START* to *B*, and thus the statements in *A* have always been executed when control reaches *B*. Fig. 2.2 shows the dominator tree for the piece of source code shown in Fig. 2.1.

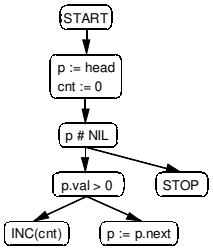

Fig. 2.2 - Dominator tree for the source code shown in Fig. 2.1

The dominator tree can be computed from the control flow graph. The algorithm due to Lengauer and Tarjan [LeTa79] runs in time $O(N * \alpha(N))$, where N is the number of nodes in the control flow graph and α is the inverse of the Ackermann function. For structured languages such as Oberon-2 the dominator tree can be computed in linear time [BrMö94].

Definition: In a directed graph with exit node *STOP*, we say that a node *A* *post-dominates* node *B*, iff for all paths *P* from *B* to *STOP*, *A* is a member of *P*. We call *A* a *post-dominator* of *B*.

Definition: We call *A* the *immediate post-dominator* of *B*, iff *A* is a post-dominator of *B*, *A* # *B*, and there is no other node *C*, for which *A* is a post-dominator and that is itself a post-dominator of *B*.

Definition: The *post-dominator tree* of a directed graph *G* with exit node *STOP* is the tree that consists of the nodes of *G*, has the root *STOP*, and has an edge between nodes *A* and *B* if *A* immediately post-dominates *B*.

If a basic block *A* post-dominates basic block *B*, *A* is on every path from *B* to *STOP*, and thus the statements in *A* will always be executed when control reaches *B*. Fig. 2.3 shows the post-dominator tree for the piece of source code shown in Fig. 2.1.

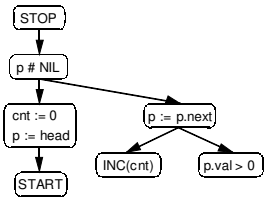

Fig. 2.3 - Post-dominator tree for the source code shown in Fig. 2.1

2.2.3 Control Dependences

Ferrante et al. [FeOW87] introduced the notion of control dependences to represent the relations between program entities due to control flow.

Definition: Let *G* be a control flow graph. Let *A* and *B* be nodes in *G*. *B* is *control dependent* on *A* iff all of the following hold:

1. There exists a directed path *P* from *A* to *B*.
2. *B* post-dominates any *C* in *P* (excluding *A* and *B*).
3. *B* does not post-dominate *A*.

If *B* is control-dependent on *A*, then *A* must have multiple successors. Following one path from *A* results in *B* being executed, while taking others may result in *B* not being executed.

Definition: The *control dependence graph* over the control flow graph *G* is the graph over all nodes of *G*, in which there is a directed edge from node *A* to node *B*, iff *B* is control dependent on *A*.

The control dependence graph compactly encodes the required order of execution of the

program's statements due to control flow. A node evaluating a condition on which the execution of other nodes depends has to be executed first. The latter nodes are therefore control dependent on the condition node.

The control dependence graph can be built from the control flow graph and the post-dominator tree using an algorithm with time complexity $O(N^2)$, where N is the number of nodes in the control flow graph [FeOW87].

For structured programming languages, control dependences reflect a program's nesting structure [HoRB90].

Definition: Let *G* be an abstract syntax tree of a structured program. The nodes of *G* represent statements and expressions of the program as well as pseudo nodes. The *control dependence graph* over *G* contains a control dependence edge from node *A* to node *B* iff one of the following holds:

1. *A* is the entry node and *B* represents a component that is not nested within any loop or conditional. These edges are labeled *T*.
2. *A* represents a control predicate and *B* represents a component immediately nested within the loop or conditional whose predicate is represented by *A*. The edge is labeled *T* if *B* is executed if the predicate *A* evaluates to *TRUE*, otherwise *F*.

The direction of the dependence indicates the flow of control. Fig. 2.4 shows the control dependences according to the latter definition for the piece of source code shown in Fig. 2.1.

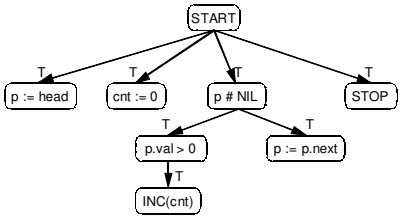

Fig. 2.4 - Control dependences for the source code shown in Fig. 2.1

2.3 Data Flow

Data flow describes the flow of the values of variables from the points of their definitions to the points where their values are used. In the following sections we describe how data flow information can be computed for structured programming languages (following [ASU86]).

2.3.1 Data Dependences

A data dependence from a node *A* to another node *B* means that the program's computation might be changed if the relative order of the nodes were reversed.

Definition: A *data dependence graph* over the abstract syntax tree of a program contains a *data dependence* (also called *flow dependence*) from node *D* to node *U* iff all of the following hold:

1. Node *D* defines variable *x*.
2. Node *U* uses *x*.
3. Control can reach *U* after *D* via an execution path along which there is no intervening definition of *x*.

The direction of the data dependence indicates the flow of the value of the defined variable. The value computed at *U* depends on all definitions *D* that may reach *U*.

Aho et al. [ASU86] use the term reaching definition to express that the value defined at a node may be used at another node.

Definition: If node *U* is data dependent on node *D* then *D* is a *reaching definition* for *U*.

The precise computation of reaching definitions is the goal of data flow analysis.

Fig. 2.5 shows a procedure that computes the greatest common divisor of two numbers along with its control dependence graph. Control dependences are shown as thin lines with small arrows.

```
PROCEDURE GCD (u, v: INTEGER): INTEGER;
  VAR t: INTEGER;
BEGIN
  REPEAT
    IF u < v THEN
      t := u; u := v; v := t
    END ;
    u := u MOD v
  UNTIL u = 0;
  RETURN v
END GCD;
```

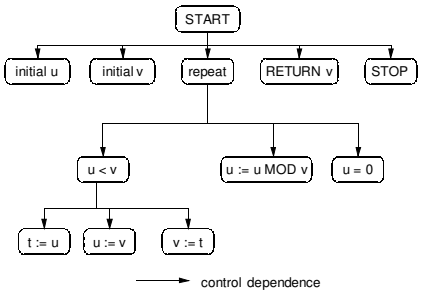

Fig. 2.5 - A program and its control dependence graph

Fig. 2.6 shows the data dependence graph. Data dependences are shown as thick lines with large arrows. The exit node is data dependent on the return value

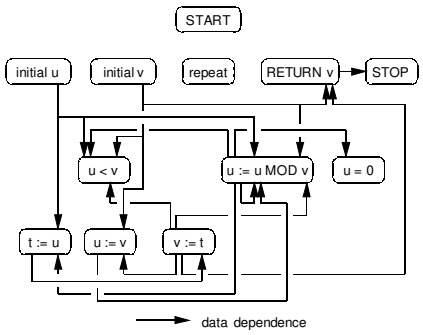

Fig. 2.6 - The data dependence graph for the program of Fig. 2.5

Fig. 2.7 shows the program dependence graph with control and data dependences.

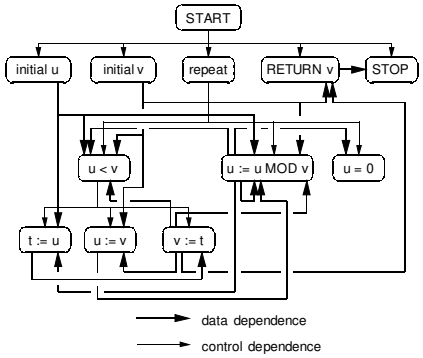

Fig. 2.7 - Program dependence graph for the program of Fig. 2.5

2.3.2 Computation of Used and Defined Variables

A first step in computing reaching definitions is to compute for each statement of the program the set of variables that are used and the set of variables that are defined by the statement.

Uses

The set of variables that are used by a statement of the dependence graph is easily computed by a traversal of the graph representation of the program. Table 2.1 shows a few examples.

Source code	Used	Defined
a := b + c	b, c	a
r.i := 5	r	i
p.next.i := a	p, next, a	i

Table 2.1 - Used and defined objects

Definitions

There are only two possibilities for changing a variable's value:

- First, the variable can be assigned a value with an assignment statement. Such a definition is unambiguous since the variable on the left-hand side is always given a new value. It is therefore also called a *killing definition* since the old value of the variable is always replaced by the new one, illustrated by the following example:

```
x := 4;          (* generates a definition of x                              *)
y := 5;          (* generates a definition of y                              *)
x := 3;          (* generates a new definition of x, kills the first definition of x *)
sum := x + y     (* only the definition of y and the last definition of x are reaching *)
```

- Second, a variable can be passed at a call as a reference parameter (VAR parameter). If the called procedure assigns to the corresponding formal parameter, the variable that has been passed as actual parameter is changed. Such a definition is in general uncertain since the actual parameter is not necessarily changed by the procedure call. It is therefore also called a *non-killing definition* since a previous definition is not killed by the new one.

There are some additional problems with definitions of array elements and record fields:

- Definition of Array Elements
  A definition of an array element must not be regarded as a killing definition of the entire array, since only one element is changed. A simple approach is to treat definitions of array elements as both a definition of the entire array (since one element is changed) and a use of the entire array (since the other elements remain their old values).

  In the following example the definition of the *i*-th array element is only killed by the subsequent definition of the *j*-th array element if *i = j*. Since one cannot deduce in general whether *i = j* or *i # j*, one has to assume that *i* may be equal to *j*. In other words, the second assignment generates a new definition but does not kill the first one. Both can reach the usage of the *i*-th array element in the last assignment. Assignments to array elements must therefore be considered as non-killing definitions.


```
a[i] := 0;   (* reaches the last statement if i # j *)
a[j] := 1;   (* reaches the last statement if i = j *)
y := a[i];
```

- Definition of Record Fields
  A record field has a constant position within the record. When accessing the field, its offset can be added to the address of the record in order to get the address of the record field. Therefore, an assignment to a record field is unambiguous as long as the address of the record is known at compile time and as long as there are no aliases. For statically allocated records, assignments to record fields can be considered as killing definitions as long as there are no aliases. In general, assignments to fields of heap-allocated records must not be considered as killing definitions.

  In the following example *p* and *q* are pointers to heap-allocated records. The definition of *p^.f* is only killed by the subsequent definition of *q^.f* if *p = q*. Since one cannot deduce in general whether *p = q* or *p # q*, one has to assume that *p* may be equal to *q*. In other words, the second assignment generates a new definition but does not kill the first one. Both can reach the usage of the field *f* in the last assignment.

```
p^.f := 0;   (* reaches the last statement if p # q *)
q^.f := 1;   (* reaches the last statement if p = q *)
y := p^.f;
```

2.3.3 Computation of Reaching Definitions

Once the sets of used and defined variables have been computed for every statement, reaching definitions can be computed for each usage of a variable. Therefore, all definitions are labeled. This label is used to identify the definition. In the following we will use the notions *definition set*, *gen set*, *kill set*, *in set*, and *out set*, which we define as:

Definition: The *definition set* of variable *x* contains as its elements the labels of all definitions that define *x*.

Definition: The *gen set* of statement *S* contains as its elements the labels of all definitions that are generated by *S*. The *kill set* of statement *S* contains as its elements the labels of all definitions that are killed by *S*.

Definition: The *in set* of statement *S* contains as its elements the labels of all definitions that reach *S*. The *out set* of statement *S* contains as its elements the labels of all definitions that leave *S*.

Algorithm for the computation of reaching definitions:

- In a first traversal, one computes the *definition set* of each variable that has been defined and the *gen* and *kill sets* for each statement.

- In another traversal, one computes the reaching definitions in a syntax-directed manner and inserts links from the usage nodes of variables to all its reaching definitions. (Remark: This is only possible for languages with structured control flow. For languages with unconstrained control flow (e.g., with gotos), an iterative approach must be chosen to compute the reaching definitions rather than a syntax-directed one.)

In order to compute the gen and kill sets as well as the reaching definitions, one has to solve the data flow equations for all statements of the program.

Data Flow Equations for Assignments

An assignment to a variable generates a definition. If the assignment is unambiguous, the definition is a killing one with a non-empty *kill* set, otherwise it is a non-killing one with an empty *kill* set. Fig. 2.8 shows a killing assignment with the associated data flow equations.

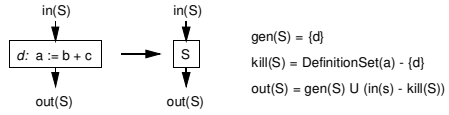

Fig. 2.8 - Data flow equations for a killing assignment

Each assignment is given a label *d*. The *gen* set of the statement has this label as its only element, meaning that it generates the definition *d* for variable *a*. On the other hand, it kills all other definitions of *a*. The *out* set consists of all definitions that are generated by *S* (i.e., *gen(S)*), since they surely reach the end of the statement. Furthermore, definitions that reach the statement *S* (i.e., *in(S)*) and are not killed by *S* (i.e., *kill(S)*) reach the end of the statement. If the assignment were non-killing, the *kill* set would be empty.

Data Flow Equations for Statement Sequences

When two statements are executed in sequence, their effects can be combined. Fig. 2.9 shows how the effects of the statements *S1* and *S2* can be combined to give the effects of the sequence *S*.

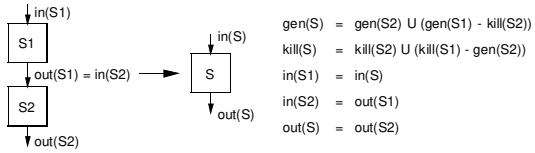

Fig. 2.9 - Data flow equations for a sequence of two statements

The compound statement *S* generates everything that is generated by *S2* (i.e. *gen(S2)*). Furthermore, all definitions that are generated by *S1* (i.e. *gen(S1)*) and are not killed by *S2* (i.e. *kill(S2)*) are generated by the compound statement *S*. Likewise, the compound statement *S* kills everything that is killed by *S2* (i.e. *kill(S2)*). Furthermore, all definitions that are killed by *S1* (i.e. *kill(S1)*) and are not generated by *S2* (i.e. *gen(S2)*) are killed by the compound statement *S*.

Data Flow Equations for Selective Statements

Fig. 2.10 shows how the effects of the branches of selective statements (such as IF and CASE) can be combined to the effects of the compound statement *S*.

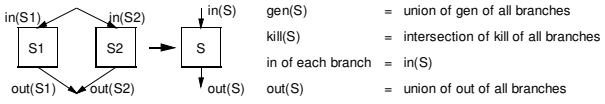

Fig. 2.10 - Data flow equations for a selection of two statements

A definition that is generated by any branch of the selective statement can be thought of as being generated by the compound statement *S*. On the other hand, a definition is only killed by the compound statement *S* if it is killed by each branch. If a definition is killed in one branch, but not in the other, the conservative assumption for the compound statement must be that the definition is not killed (since one cannot determine statically which branch will actually be executed).

Data Flow Equations for Iterative Statements

Fig. 2.11 shows the data flow equations for iterative statements (such as WHILE and REPEAT).

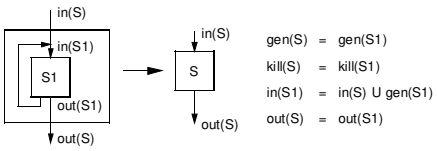

Fig. 2.11 - Data flow equations for an iterative statement

The *gen* and *kill* sets of the compound statement are the same as for the nested statement sequence: If a definition is generated during the first iteration of the loop, it will also be generated during the second iteration and so on. The proof why *in(S1)* can be regarded as the union of *in(S)* and *gen(S1)* and not (as obvious from the figure) as the union of *in(S)* and *out(S1)* is given in Section 4.5.2

Aho et al. [ASU86] describe a two-phase algorithm that can be used to solve the data flow equations for structured programming languages:

- The *gen* and *kill* sets that have been computed in the previous step for each defining node can be composed in a bottom-up manner for each statement sequence.
- For each statement, the *out* set is computed as a function of the *gen* and *kill* sets as well as of the *in* set by applying the equation

  out(S) = gen(S) ∪ (in(S) - kill(S))

Fig. 2.12 will illustrate this algorithm for the computation of reaching definitions with a small example.

```
          MODULE ComputeGenKill;                 Definition Sets:
                                                   u: {0, 3, 6, 8}
          VAR u, v, t: INTEGER;                    v: {1, 4, 7}
          (* initial definitions:                  t: {2, 5}
(* 0: *)     u := 0;
(* 1: *)     v := 0;                  Node   gen    kill
(* 2: *)     t := 0;                    0:   {0}    {3, 6, 8}
          *)                            1:   {1}    {4, 7}
          BEGIN                         2:   {2}    {5}
(* 3: *)     u := 10;                   3:   {3}    {0, 6, 8}
(* 4: *)     v := 2;                    4:   {4}    {1, 7}
            IF u < v THEN                5:   {5}    {2}
(* 5: *)        t := u;                 6:   {6}    {0, 3, 8}
(* 6: *)        u := v;                 7:   {7}    {1, 4}
(* 7: *)        v := t                  8:   {8}    {0, 3, 6}
            END ;
(* 8: *)     u := u MOD v
          END ComputeGenKill.
```

Fig. 2.12 - Example for the computation of the *gen* and *kill* sets

First, the *gen* and *kill* sets of the individual statements are composed for each statement sequence in a bottom-up manner.

Sequence 3-4:	gen(3-4)	= gen(4) ∪ (gen(3) - kill(4))	= {4} ∪ ({3} - {1, 7})
			= {3..4}
	kill(3-4)	= kill(4) ∪ (kill(3) - gen(4))	= {1, 7} ∪ ({0, 6, 8} - {1, 7})
			= {0..1, 6..8}
Sequence 5-6:	gen(5-6)	= gen(6) ∪ (gen(5) - kill(6))	= {6} ∪ ({5} - {0, 3, 8})
			= {5..6}
	kill(5-6)	= kill(6) ∪ (kill(5) - gen(6))	= {0, 3, 8} ∪ ({2} - {6})
			= {0, 2..3, 8}

Sequence 5-7:	gen(5-7)	= gen(7) ∪ (gen(5-6) - kill(7))	= {7} ∪ ({5..6} - {1, 4})
			= {5..7}
	kill(5-7)	= kill(7) ∪ (kill(5-6) - gen(7))	= {1, 4} ∪ ({0, 2..3, 8} - {7})
			= {0..4, 8}
Selection IF:	gen(IF)	= gen(5-7) ∪ gen(ELSE)	= {5..7} ∪ {}
			= {5..7}
	kill(IF)	= kill(5-7) ∩ kill(ELSE)	= {0..4, 8} ∩ {}
			= {}
Sequence 3-4-IF:	gen(3-4-IF)	= gen(IF) ∪ (gen(3-4) - kill(IF))	= {5..7} ∪ ({3..4} - {})
			= {3..7}
	kill(3-4-IF)	= kill(IF) ∪ (kill(3-4) - gen(IF))	= {} ∪ ({0..1, 6..8} - {5..7})
			= {0..1, 8}
Sequence 3-8:	gen(3-8)	= gen(8) ∪ (gen(3-4-IF) - kill(8))	= {8} ∪ ({3..7} - {0, 3, 6})
			= {4..5, 7..8}
	kill(3-8)	= kill(8) ∪ (kill(3-4-IF) - gen(8))	= {0, 3, 6} ∪ ({0..1, 8} - {8})
			= {0..1, 3, 6}

Then the *out* sets are computed as a function of the *in* sets as well as the *gen* and *kill sets* (out = gen ∪ (in - kill). The *in* set for statement 3 consists of the initial definitions. At each node *U*, reaching definitions are inserted to all nodes *D* whose labels are included in *in(U)* and which define the variable that is used at *U*.

Node 3:	in(3)	= {0..2}		
	out(3)	= gen(3) ∪ (in(3) - kill(3))	= {3} ∪ ({0..2} - {0, 6, 8})	= {1..3}
Node 4:	in(4)	= out(3)	= {1..3}	
	out(4)	= gen(4) ∪ (in(4) - kill(4))	= {4} ∪ ({1..3} - {1, 7})	= {2..4}
IF:	in(IF)	= out(4)	= {2..4}	
			node 3 is a reaching definition for usage of *u*	
			node 4 is a reaching definition for usage of *v*	
	out(IF)	= gen(IF) ∪ (in(IF) - kill(IF)	= {5..7} ∪ ({2..4} - {})	= {2..7}
Node 5:	in(5)	= in(IF)	= {2..4}	
			node 3 is a reaching definition for usage of *u*	
	out(5)	= gen(5) ∪ (in(5) - kill(5))	= {5} ∪ ({2..4} - {2})	= {3..5}
Node 6:	in(6)	= out(5)	= {3..5}	
			node 4 is a reaching definition for usage of *v*	
	out(6)	= gen(6) ∪ (in(6) - kill(6))	= {6} ∪ ({3..5} - {0, 3, 8})	= {4..6}
Node 7:	in(7)	= out(6)	= {4..6}	
			node 5 is a reaching definition for usage of *t*	
	out(7)	= gen(7) ∪ (in(7) - kill(7))	= {7} ∪ ({4..6} - {1, 4})	= {5..7}
Node 8:	in(8)	= out(IF)	= {2..7}	
			node 3 is a reaching definition for usage of *u*	
			node 6 is a reaching definition for usage of *u*	
			node 4 is a reaching definition for usage of *v*	
			node 7 is a reaching definition for usage of *v*	
	out(8)	= gen(8) ∪ (in(8) - kill(8))	= {8} ∪ ({2..7} - {0, 3, 6})	= {2, 4..5, 7..8}

2.4 Program Slicing

Program slicing [Wei84] is a program analysis and reverse engineering technique that reduces a program to those statements that are relevant for a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the value of variable *v* at statement *s*?"

Program slicing was originally introduced by Mark Weiser as a "method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a *slice*, is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior." [Wei84] He defined a slice with respect to a program point *p* and a subset of the program variables *V* to consist of all statements in the program that may affect the values of the variables in *V* at point *p*. In other words, a program slice consists of all parts of the program that (potentially) affect the values of the interesting variables at some point of the program.

Program slicing usually requires access to the source code of the program, since it can be seen as a source code to source code transformation. However, program slicing algorithms work on an internal representation of the program. If this internal representation can be derived from the object code or bytecode of a compiled program (e.g., via decompilation), and if the internal representation can again be visualized as program source code, then access to the source code is not necessary. Fig. 2.13 shows a piece of source code and three slices computed for different slicing criteria: The first slice is derived for line 12 and variable *z*, the second for line 9 and variable *x*, the third for line 12 and variable *total*.


```
1   BEGIN                          BEGIN
2     Read(x, y);                    Read(x, y);
3     total := 0;                    IF x <= 1 THEN
4     sum := 0;                      ELSE
5     IF x <= 1 THEN                   Read(z);
6       sum := y                     END
7     ELSE                         END
8       Read(z);
9       total := x * y             BEGIN
10    END ;                          Read(x, y);
11    Write(total, sum)            END
12  END

                                   BEGIN
                                     Read(x, y);
                                     total := 0;
                                     IF x <= 1 THEN
                                     ELSE
                                       total := x * y
                                     END
                                   END
```

(12, {z} → 9, {x} → 12, {total})

Fig. 2.13 - Piece of source code with 3 examples of slices

The following sections describe variants of program slicing and their applications. We partly follow the survey of Binkley and Gallagher [BiG96].

2.4.1 Variants of Program Slicing

Static Slicing and Dynamic Slicing

Static slicing [Wei84] uses static analysis to derive slices, i.e. the source code of the program is analyzed and the slices are computed for all possible input values. No assumptions may be made about the input values, predicates may evaluate either to true or false. Therefore, conservative assumptions have to be made, which may lead to relatively big slices. A static slice contains all statements that *may* effect the value of a variable at a program point for *every possible* execution.

Dynamic slicing (introduced by Korel and Laski [KoL88]) makes use of the information about a particular execution of a program. The execution of the program is monitored, and the dynamic slices are computed with respect to the execution history. A dynamic slice contains all statements that *actually* affect the value of a variable at a program point for *that particular* execution.

Fig. 2.14 demonstrates the difference between static and dynamic slicing in a simple example: Depending on an operation code entered by the user, different values are computed. The result is printed. In both cases, the slice with respect to the output statement is shown in bold face.


```
MODULE StaticSlicing;

IMPORT Math, In, Out;

VAR
  x, y: REAL;
  op: ARRAY 10 OF CHAR;

BEGIN
  In.Open;
  In.String(op); In.Real(x);
  IF op = "sin" THEN
    y := Math.Sin(x)
  ELSE
    y := Math.Cos(x)
  END ;
  Out.Real(y)
END StaticSlicing.
```

```
MODULE DynamicSlicing;

IMPORT Math, In, Out;

VAR
  x, y: REAL;
  op: ARRAY 10 OF CHAR;

BEGIN
  In.Open;
  In.String(op); In.Real(x);
  IF op = "sin" THEN
    y := Math.Sin(x)
  ELSE
    y := Math.Cos(x)
  END ;
  Out.Real(y)
END DynamicSlicing.
```

Fig. 2.14 - Static slice computed for the last statement (left) and dynamic slice for the input *op = "sin"* (right)

Backward Slicing and Forward Slicing

Program slices, as originally introduced by Weiser [Wei84], are now called backward slices, because they contain all parts of the program that might have influenced the variable at the statement under consideration. On the other hand, *forward slices* contain all parts of the program that might be influenced by the variable. Fig. 2.15 shows the backward and forward slices for the statement *x := 3*.

```
MODULE BackwardSlicing;

VAR
  x, y, z: INTEGER;

BEGIN
  x := 3;
  y := x + 4;
  z := y + 3
END BackwardSlicing.
```

```
MODULE ForwardSlicing;

VAR
  x, y, z: INTEGER;

BEGIN
  x := 3;
  y := x + 4;
  z := y + 3
END ForwardSlicing.
```

Fig. 2.15 - Backward slice and forward slice computed for the statement "y := x + 4"

Intraprocedural Slicing and Interprocedural Slicing

Intraprocedural slicing computes slices within one procedure. Calls to other procedures are either not handled at all or handled conservatively. If the program consists of more than one procedure, interprocedural slicing can be used to derive slices that span multiple procedures.

Interprocedural slicing raises a new problem: When a procedure is called at different places, the calling context must be considered, in order to correctly model the run-time

execution at compile time. Interprocedural data flow analysis has a similar goal to only consider paths that correspond to legal call/return sequences. Such paths are called realizable, valid, or feasible. Fig. 2.16 shows a module where procedure *Add* is called at two places: once in procedure *Increment*, another time in procedure *A*.

```
MODULE CallingContext;

PROCEDURE Add (VAR a: INTEGER; b: INTEGER);
BEGIN a := a + b
END Add;

PROCEDURE Increment (VAR z: INTEGER);
BEGIN Add(z, 1)
END Increment;

PROCEDURE A (VAR x, y: INTEGER);
BEGIN
  Add(x, y);
  Increment(y)
END A;

PROCEDURE Main;
  VAR sum, i: INTEGER;
BEGIN
  sum := 0;
  i := 1;
  WHILE i < 11 DO
    A(sum, i)
  END
END Main;

END CallingContext.
```

Fig. 2.16 - Example module with two call sites of procedure *Add*

Fig. 2.17 shows a trace of the procedure activations during the execution of procedure *Main*.

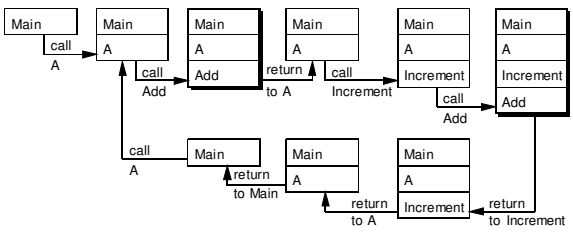

Fig. 2.17 - Trace of procedure activations (activations of *Add* are shaded)

When computing the slice, "regarding the calling context" means that the slicing algorithm correctly models the execution. When the call of *Add* in procedure *Increment* is encountered, it is necessary to continue the analysis with procedure *Add*, but when returning from procedure *Add*, analysis of procedure *Increment* must be continued. It would not be a precise model of the run-time execution to call *Add* in procedure *Increment* but to return to procedure *A*, as shown in Fig. 2.18.

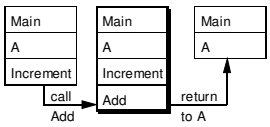

Fig. 2.18 - Trace of procedure activations equivalent to wrong handling of calling context

If procedure *Increment* is sliced for the output parameter *z* without regarding the calling context, the slice will contain the whole program. This is imprecise, if one allows not only the deletion of entire statements from the original program but also of smaller parts such as individual parameters: the call of *Add* within procedure *A*, the first actual parameter of *A* at its call in procedure *Main* and the initialization of *sum* are not relevant. The imprecision is introduced because *Add* is (necessarily) included into the slice (since it is called in *Increment*) and all call sites of *Add* are then (unnecessarily) included into the slice. The call of *Add* in *Increment* must be included into the slice, but the call of *Add* in *A* should not be included into the slice. The inclusion of the call of *Add* within *A* into the slice necessitates the inclusion of the formal parameter *x* of *A*, the corresponding actual parameter *sum* and the initialization of *sum*. On the other hand, if the calling context is regarded, the run-time execution of the program is modeled correctly and the slice will only contain the relevant parts, as shown in Fig. 2.19.

```
MODULE CallingContext;

PROCEDURE Add (VAR a: INTEGER; b: INTEGER);
BEGIN a := a + b
END Add;

PROCEDURE Increment (VAR z: INTEGER);
BEGIN Add(z, 1)
END Increment;

PROCEDURE A (VAR x, y: INTEGER);
BEGIN
   Add(x, y);
   Increment(y)
END A;

PROCEDURE Main;
   VAR sum, i: INTEGER;
BEGIN
   sum := 0;
   i := 1;
   WHILE i < 11 DO
      A(sum, i)
   END
END Main;

END CallingContext.
```

Fig. 2.19 - Slice computed for the output parameter *z* of procedure *Increment* with regarding the calling context

Slicing Granularity

Program slices can be computed at different abstraction levels. The parts of the program that are considered to be included into the slice can be as big as procedures or as small as nodes of the syntax tree of the program (e.g., statements, expressions, variables, parameters, etc.). Slicing at the level of syntax tree nodes (also called at the expression

level) gives the most detailed and precise information. However, the resulting slices are no longer executable programs. In Fig. 2.20 we show the slice for the last statement computed either by expression-oriented slicing or by statement-oriented slicing.

```
MODULE ExpressionSlicing;

IMPORT Out;

VAR i, j, k, l: INTEGER;

PROCEDURE F (VAR i: INTEGER): INTEGER;
BEGIN INC(i); RETURN i
END F;

BEGIN
   i := 1; j := 2; k := 3;
   l := i + F(j) * k;
   Out.Int(j, 0)
END ExpressionSlicing.
```

```
MODULE StatementSlicing;

IMPORT Out;

VAR i, j, k, l: INTEGER;

PROCEDURE F (VAR i: INTEGER): INTEGER;
BEGIN INC(i); RETURN i
END F;

BEGIN
   i := 1; j := 2; k := 3;
   l := i + F(j) * k;
   Out.Int(j, 0)
END StatementSlicing.
```

Fig. 2.20 - Slicing at the expression level (left) and at the statement level (right) computed for the last statement

2.4.2 Applications

Program slicing can be used to assist the programmer in a lot of tedious and error prone tasks. In the following we give a brief survey.

Debugging

During debugging, a programmer usually has a test case in mind which causes the program to fail. A program slicer that is integrated into the debugger can be very useful in discovering the reason for the error by visualizing control and data dependences and by highlighting the statements that are part of the slice. Variants of program slicing have been developed to further assist the programmer: *Program dicing* [LyW86] identifies statements that are likely to contain bugs by using information that some variables fail some tests while others pass all tests. Several slices are combined with each other in different ways: e.g. the intersection of two slices contains all statements that lead to an error in both test cases; the intersection of slice *a* with the complement of slice *b* excludes from slice *a* all statements that do not lead to an error in the second test case. Another variant of program slicing is *program chopping* [JaR94]. It identifies statements that lie between two points *a* and *b* in the program and will be affected by a change at *a*. This can be useful when a change at *a* causes an incorrect result at *b*. Debugging should be focused on the statements between *a* and *b* that transmit the change of *a* to *b*.

Program Integration

Programmers frequently face the problem of integrating several variants of a base program. It is only a first step to simply look for textual differences. Semantics-based program integration is a technique that attempts to create an integrated program that incorporates the changed computations of the variants as well as the computations of the base program that are preserved in all variants.

Berzins [Be86] addresses a part of the program-integration problem from the semantic perspective. Given two programs, his method attempts to find a merged program that is the least (semantic) extension that subsumes both versions, that is, a merged program that incorporates the whole behavior of the two versions. However, as software evolves, not only extensions but also modifications (such as bug fixes) are made to the base program. Modifications are not addressed by his method.

Horwitz et al. [HoPR89] presented an algorithm for semantics-based program integration that creates the integrated program by merging certain program slices of the variants. Their integration algorithm takes as input three programs *Base, A,* and *B*, where *A* and *B* are variants of *Base*. The integrated program is produced by (1) building graphs that represent *Base, A,* and *B*, (2) combining program slices of the program dependence graphs of *Base, A,* and *B* to form a merged graph, (3) testing the merged graph for certain interference criteria, and (4) reconstituting a program from the merged graph.

Yang [Yan90] extends the algorithm of Horwitz et al.: The new algorithm is extendible in that it can incorporate any techniques for detecting program components with equivalent behaviors (components with isomorphic slices, see [HoR91]) and it can accommodate semantics-preserving transformations. He classifies the nodes of *A* into the classes:

- *NewA* is the class of all nodes of *A* that have no corresponding nodes in *Base*. These nodes represent program components that have been added to *Base* to create *A*, or have been moved to a context that has changed their execution behaviors (similar for *NewB*).
- *ModifiedA* is the class of all nodes of *A* that have a corresponding node in *Base*, but the node's text in *A* differs from the text of the corresponding node in *Base*. These nodes represent components of *A* whose texts have been changed but whose execution behaviors remain the same.
- *ModifiedB* is the class of all nodes of *A* that have corresponding nodes in *Base* and *B*, for which the node's text in *A* is the same as the text of the corresponding node in *Base*, but whose text differs from the text of the corresponding node in *B*.
- *IntermediateA* is the class of all nodes of *A* that have a corresponding node in *Base* and whose text in *A* is the same as the text of the corresponding node in *Base*, but there is no corresponding node in *B* (either because the node was deleted from *B,* or because the node's execution behavior was changed, or because the node assigns to a different variable in *B*).
- *Unchanged* is the class of all nodes of *A* that have corresponding nodes in *Base* and *B*. All three nodes have the same text. These nodes represent components whose texts and behaviors are identical in all three programs.

Likewise, the nodes of *B* are classified into the sets *NewB, ModfiedB, ModifiedA, Unchanged*, and *IntermediateB*. The nodes of *Base* are similarly classified into the sets *ModifiedA, ModifiedB, IntermediateA, IntermediateB, Unchanged*, and *Deleted*. A node in *Base* is in *Deleted* if neither *A* nor *B* contains a corresponding node. The classification process may discover that *A* and *B* interfere with respect to *Base* by identifying corresponding nodes *nodeA* and *nodeB* in *A* and *B* such that:

- The text of *nodeA* differs from the text of *nodeB*.
- If there is a corresponding node *nodeBase* in *Base*, the texts of *nodeA* and *nodeBase*, and the texts of *nodeB* and *nodeBase* are unequal.

Since a node in the merged graph can have only one text, it is not possible to preserve the changed text of this component from both *A* and *B*. This can occur either for a node in *NewA* (with a corresponding node in *NewB*), or for a node in *ModifiedA* (with a corresponding node in *ModifiedB*).

Software Maintenance

The main challenges in software maintenance are to understand existing software and to make changes without introducing new bugs. A *decomposition slice* [GaL92] is useful in making a change to a piece of software without unwanted side effects. It captures all computations of a variable and is independent of a program location. The decomposition slice for a variable *v* is the union of slices taken at *critical nodes* with respect to variable *v*. Critical nodes are the nodes that output the value of *v* and the last node of the program. The decomposition slices are computed for all variables of the program. The decomposition slice for variable *v* partitions the program into three parts:

- The *independent part* contains all the statements of the decomposition slice (taken with respect to *v*) that are not part of any decomposition slice taken with respect to another variable.
- The *dependent part* contains all statements of the decomposition slice (taken with respect to *v*) that are part of another decomposition slice taken with respect to another variable.
- The *complement* contains all statements that are not in the decomposition slice (taken with respect to *v*). The statements of the complement may nevertheless be part of some other decomposition slice taken with respect to another variable. The complement must remain fixed after changing a statement of the decomposition slice.

Likewise, variable *v* can be categorized as

- *changeable* if all assignments to *v* are within the independent part.
- *unchangeable* if at least one assignment to *v* is in a dependent part. If the maintainer modifies this assignment, the new value will flow out of the decomposition.
- *used* if it is not used in the dependent or independent parts but in the complement. The maintainer may not declare new variables with the same name.

Several conclusions can be drawn for modifications:

- Statements of the independent part may be deleted from a decomposition slice since they do not affect the computation of the complement.
- Assignments to changeable variables may be added anywhere in the decomposition slice.
- New control statements that surround any statements of the dependent part will cause the complement to change.

The maintainer who tries to change the code only has to regard the dependent and independent parts of the program. After the modification, only the dependent and independent parts will have to be retested. The complement is guaranteed to be unaffected by the change, it will not have to be retested [Gal91].

Testing

Software maintainers are also faced with the task of regression testing: retesting software after a modification. Even after the smallest change, extensive tests may be necessary, running a large number of test cases. While decomposition slicing eliminates the need for regression testing on the complement, there may still be a substantial number of tests to be run on the dependent, independent and changed parts. A lot of work has been done in order to test incrementally [BaH93], to simplify testing [HaD95], to apply program slicing to regression testing [GuHS96] and to test path selection [Bi95, FoB97].

Software Quality Assurance

Software quality assurance auditors have to locate safety critical code and to ascertain its effects throughout the system. Program slicing can be used to locate all code that influences the values of variables that might be part of a safety critical component. But beforehand these critical components still have to be determined by domain experts.

One possibility to assure high quality is to make the system redundant. If two output values are critical, then these output values should be computed independently. They should not depend on the same internal functions, since the same error might manifest in both output values in the same way, thereby hiding the error. One technique to defend against such errors is to use functional diversity, where multiple algorithms are used for the same purpose. Thus the critical output values depend on different internal functions. Program slicing can be used to determine the logical independence of the slices computed for the two output values [Ly+95].

Functional Cohesion

Cohesion measures the relatedness of some component. A highly cohesive software module is a module that has one function and is indivisible - it is difficult to split a cohesive module into separate components. Cohesion has been categorized as *coincidental* (weakest form), *logical, procedural, communicational, sequential* and *functional* (strongest form) [YoC79].

Bieman and Ott [BiO94] define *data slices* that consist of data tokens (instead of statements). Data tokens may be variable and constant definitions and references. A data slice for a data token *v* is the sequence of all data tokens in the statements that comprise the backward and forward slices of *v*. Fig. 2.21 shows a piece of source code and the data slice for *sum*. The parts of the data slice are shown in bold face.

```
PROCEDURE SumAndProduct (n: INTEGER; VAR sum, prod: INTEGER);
  VAR i: INTEGER;
BEGIN
  sum := 0;
  prod := 1;
  FOR i := 0 TO n - 1 DO
    sum := sum + i;
    prod := prod * i
  END
END SumAndProduct;
```

Fig. 2.21 - A piece of source code and the data slice for *sum*

Data slices are computed for each output of a procedure (e.g., output to a file, output parameter, assignment to a global variable). The tokens that are common to more than one data slice are the connections between the slices, they are the "glue" that binds the slices together. The tokens that are in every data slice of a function are called *super-glue*, tokens that are in more than one slice are called *glue*. *Strong functional cohesion* can be expressed as the ratio of super-glue tokens to the total number of tokens in the slice, whereas *weak functional cohesion* may be seen as the ratio of glue tokens to the total number of tokens. The *adhesiveness* of a token is another measure expressing how many slices are glued together by that token.

3 Current Slicing Algorithms

This chapter describes current slicing algorithms together with their data structures ranging from the original approach where slicing is seen as a data flow problem to the state-of-the-art where slicing is seen as a graph-reachability problem.

Weiser used a control flow graph as an intermediate representation for his slicing algorithm. He computed slices by solving the data flow problem of relevant nodes. He gave algorithms for intraprocedural and interprocedural slicing. However, the interprocedural version did not account for the calling context and therefore produced imprecise slices. Ottenstein et al. [OtO84, FeOW87] recognized that intraprocedural backward slices could be efficiently computed using dependence graphs as intermediate representations by traversing the dependence edges backwards (from target to source). Horwitz et al. [HoRB90] introduced system dependence graphs for interprocedural slicing. They also developed a two-phase algorithm that computes precise interprocedural slices. With the help of summary edges they accounted for the transitive effects of procedure calls without descending into called procedures. They computed these summary edges by a variation on the technique to compute the subordinate characteristic graphs of an attribute grammar's nonterminals [Ka80]. Livadas et al. [LivC94, LivJ95] proposed a simpler method of computing the summary edges. In the following sections we will discuss these algorithms and their data structures in more detail.

3.1 Slicing as a Data Flow Problem

Weiser used a control flow graph as an intermediate representation for his slicing algorithm. Computing a slice from a control flow graph requires computation of the data flow information about the set of *relevant variables* at each node. The sets of relevant variables for the slice taken with respect to node *n* and variables *V* can be computed as follows:

1. Initialize the relevant sets of all nodes to the empty set.
2. Insert all variables of *V* into *relevant(n)*.
3. For *n*'s immediate predecessor *m*, compute *relevant(m)* as:

```
relevant(m) := relevant(n) - def(m)          (* exclude all variables that
                                                are defined at m          *)
if relevant(n) ∩ def(m) # {} then            (* if m defines a variable that
                                                is relevant at n          *)
    relevant(m) := relevant(m) ∪ ref(m)      (* include the variables that
                                                are referenced at m       *)
    include m into the slice
end
```


4. Work backwards in the control flow graph, repeating step 3 for *m*'s immediate predecessors until the entry node is reached or the relevant set is empty.

Table 3.1 shows an example for the computation of the relevant sets (taken from [BiG96]). The slice is computed for the last statement and the variable *a*. The nodes that are finally part of the slice are shown with bold node numbers.

n	Statement	ref(n)	def(n)	relevant(n)
1	b = 1		b	
2	c = 2		c	b
3	d = 3		d	b, c
4	a = d	d	a	b, c
5	d = b + d	b, d	d	b, c
6	b = b + 1	b	b	b, c
7	a = b + c	b, c	a	b, c
8	print a	a		a

Table 3.1 - Source code with relevant sets, slice for <8, {a}>

```
Step 2: relevant(8) = {a}
Step 3: relevant(7) = relevant(8) - def(7)      = {a} - {a}      = {}
        relevant(7) = relevant(7) ∪ ref(7)      = {} ∪ {b, c}    = {b, c}
        Since node 7 defines a variable relevant at node 8, it is included into the slice.
Step 3: relevant(6) = relevant(7) - def(6)      = {b, c} - {b}   = {c}
        relevant(6) = relevant(6) ∪ ref(6)      = {c} ∪ {b}      = {b, c}
        Since node 6 defines a variable relevant at node 7, it is included into the slice.
Step 3: relevant(5) = relevant(6) - def(5)      = {b, c} - {d}   = {b, c}
Step 3: relevant(4) = relevant(5) - def(4)      = {b, c} - {a}   = {b, c}
Step 3: relevant(3) = relevant(4) - def(3)      = {b, c} - {d}   = {b, c}
Step 3: relevant(2) = relevant(3) - def(2)      = {b, c} - {c}   = {b}
        relevant(2) = relevant(2) ∪ ref(2)      = {b} ∪ {}       = {b}
        Since node 2 defines a variable relevant at node 3, it is included into the slice.
Step 3: relevant(1) = relevant(2) - def(1)      = {b} - {b}      = {}
        relevant(1) = relevant(1) ∪ ref(1)      = {} ∪ {}        = {}
        Since node 1 defines a variable relevant at node 2, it is included into the slice.
```

For structured programs, a statement can have multiple predecessors. The algorithm outlined above must therefore be extended:

- to compute the control sets for each node,
- to combine the relevant sets at points where the control flow merges (by unioning the relevant sets), and
- to compute the relevant sets iteratively until there are no further changes.

The *control set* associates with each node the set of predicate statements that directly control its execution. Whenever a statement is added to the slice, the members of its control set are included into the slice. New slices are computed with respect to the nodes that have been included due to the control set and the variables referenced at these nodes. All statements of the new slices are considered to be part of the original slice.

The following example (taken from [BiG96]) shows the computation of the relevant sets. The slice is computed for node 11 with respect to variable *a*. Table 3.2 shows the source code of the example and for each statement the sets of referenced and defined variables as well as the control set and the relevant set.

n	Statement	ref(n)	def(n)	control(n)	relevant(n)
1	b = 1		b		
2	c = 2		c		b
3	d = 3		d		b, c
4	a = d	d	a		b, c, d
5	if a then	a			b, c, d
6	d = b + d	b, d	d	5	b, d
7	c = b + d	b, d	c	5	b, d
	else				
8	b = b + 1	b	b	5	b, c
9	d = b + 1	b	d	5	b, c
	endif				b, c
10	a = b + c	b, c	a		b, c
11	print a	a			a

Table 3.2 - Source code with relevant sets, slice for <11, {a}>

Step 2: relevant(11) = {a}
Step 3: relevant(10) = relevant(11) - def(10) = {a} - {a} = {}
relevant(10) = relevant(10) ∪ ref(10) = {} ∪ {b, c} = {b, c}
Since node 10 defines a variable relevant at node 11, it is included into the slice.
Step 3: relevant(9) = relevant(10) - def(9) = {b, c} - {d} = {b, c}
Step 3: relevant(8) = relevant(9) - def(8) = {b, c} - {b} = {c}
relevant(8) = relevant(8) ∪ ref(8) = {c} ∪ {b} = {b, c}
Since node 8 defines a variable relevant at node 9, it is included into the slice.
Since control(8) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.
Step 3: relevant(7) = relevant(10) - def(7) = {b, c} - {c} = {b}
relevant(7) = relevant(7) ∪ ref(7) = {b} ∪ {b, d} = {b, d}
Since node 7 defines a variable relevant at node 10, it is included into the slice.
Since control(7) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.
Step 3: relevant(6) = relevant(7) - def(6) = {b, d} - {d} = {b}
relevant(6) = relevant(6) ∪ ref(6) = {b} ∪ {b, d} = {b, d}
Since node 6 defines a variable relevant at node 7, it is included into the slice.
Step 3: relevant(5) = relevant(6) ∪ relevant(8) = {b, d} ∪ {b, c} = {b, c, d}
Step 3: relevant(4) = relevant(5) - def(4) = {b, c, d} - {a} = {b, c, d}
Step 3: relevant(3) = relevant(4) - def(3) = {b, c, d} - {d} = {b, c}
relevant(3) = relevant(3) ∪ ref(3) = {b, c} ∪ {} = {b, c}
Since node 3 defines a variable relevant at node 4, it is included into the slice.
Step 3: relevant(2) = relevant(3) - def(2) = {b, c} - {c} = {b}
relevant(2) = relevant(2) ∪ ref(2) = {b} ∪ {} = {b}
Since node 2 defines a variable relevant at node 3, it is included into the slice.
Step 3: relevant(1) = relevant(2) - def(1) = {b} - {b} = {}
relevant(1) = relevant(1) ∪ ref(1) = {} ∪ {} = {}
Since node 1 defines a variable relevant at node 2, it is included into the slice.

Table 3.3 shows how the slice is computed for node 5 with respect to variable *a*:

n	Statement	ref(n)	def(n)	control(n)	relevant(n)
1	b = 1		b		
2	c = 2		c		
3	d = 3		d		{}
4	a = d	d	a		{d}
5	if a then	a			{a}
6	d = b + d	b, d	d	5	
7	c = b + d	b, d	c	5	
	else				
8	b = b + 1	b	b	5	
9	d = b + 1	b	d	5	
	endif				
10	a = b + c	b, c	a		
11	print a	a			

Table 3.3 - Source code with relevant sets, slice for <5, {a}>

Step 2: relevant(5) = {a}
Step 3: relevant(4) = relevant(5) - def(4) = {a} - {a} = {}
Since node 4 defines a variable relevant at node 5, it is included into the slice.
relevant(4) = relevant(4) ∪ ref(4) = {} ∪ {d} = {d}
Step 3: relevant(3) = relevant(4) - def(3) = {d} - {d} = {}
Since node 3 defines a variable relevant at node 4, it is included into the slice.
relevant(3) = relevant(3) ∪ ref(3) = {} ∪ {} = {}
Since the relevant set is empty, no more nodes will be included into the slice.

The complete slice contains the nodes 10, 8, 7, 6, 5, 4, 3, 2, and 1.

When the program contains loops, iteration over parts of the control flow graph is necessary until the relevant sets and the slice stabilize. The maximum number of iterations is the same as the number of assignment statements in the loop. Weiser describes a method to derive interprocedural slices, essentially by in-line replacement of each procedure with appropriate substitutions for the parameters. However, his method does not account for the calling context and yields imprecise slices.

A big disadvantage of computing program slices this way is that the relevant sets have to be computed for each slice and that this information cannot be reused for other slices.

3.2 Slicing as a Graph-Reachability Problem

Another approach to compute program slices is to first derive an intermediate representation of the program that models the dependences among the program entities and to compute slices simply by traversing the dependences of this intermediate representation. The big advantage of this approach is that data flow analysis only has to be performed once and that the information can be reused for deriving all kinds of slices, such as forward and backward slices, as well as intraprocedural and interprocedural slices.

3.2.1 Program Dependence Graph

Ferrante et al. [FeOW87] introduced *program dependence graphs* which combine control dependences and data dependences into a common framework. The nodes of a program dependence graph represent statements and predicate expressions of the program. Each node of the graph has references to the nodes that it is control dependent on and to the nodes that define its operands. The set of all dependences induce a partial ordering on the statements and predicates in the program that must be followed in order to preserve the semantics of the original program.

They propose to use program dependence graphs for optimizations. Since both the essential control relationships and the essential data relationships are present in the program dependence graph, a single traversal of these dependences is sufficient to perform many optimizations.

3.2.2 System Dependence Graph

Horwitz et al. [HoRB90] enhance the program dependence graph to facilitate interprocedural slicing. They add vertices for the program entry, for the initial definition of variables and for the final use of variables to the graph. They label control dependences with either *true* or *false*, meaning that when the predicate at the origin of the dependence evaluates to *true* (or *false*), the target node of the dependence labeled *true* (or *false*) will eventually be executed. They classify data dependences as *loop carried* and *loop independent*. However, for slicing purposes the distinction between different kinds of data dependences is not necessary. Fig. 3.1 shows the program dependence graph for a small program.

```
MODULE PDG;

VAR i: INTEGER;

BEGIN
  i := 1;
  WHILE i < 11 DO
    i := i + 1
  END
END PDG.
```

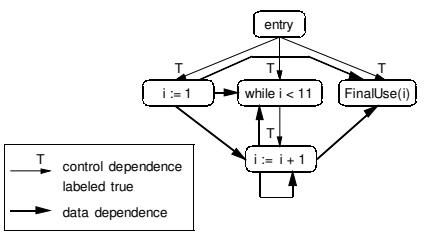

Fig. 3.1 - A small program with its program dependence graph

Using a program dependence graph, program slicing can be seen as a graph reachability problem. The slice of a dependence graph with respect to a node *n* of the graph, is the graph that contains all nodes that (directly or indirectly) reach *n* via a data dependence or control dependence. Horwitz et al. [HoRB90] give a simple worklist algorithm for deriving intraprocedural slices. Fig. 3.2 shows a recursive implementation.


```
PROCEDURE SliceNodeIntraproc (node: Node);
BEGIN
  IF node is not marked THEN
    mark node as visited
    FOR all nodes pred on which node depends DO
      SliceNodeIntraproc(pred)
    END
  END
END SliceNodeIntraproc;
```

Fig. 3.2 - Intraprocedural backward slicing algorithm

For programs that consist of several procedures, Horwitz et al. define the *system dependence graph* that contains one program dependence graph for each procedure of the program. They introduce several nodes to model procedure calls and parameter passing, where parameters are passed by value-result and accesses to global variables are modeled via additional parameters of the procedure:

- *Call-site nodes* represent the call sites.
- *Actual-in* and *actual-out nodes* represent the input and output parameters at the call sites. They are control dependent on the call-site node.
- *Formal-in* and *formal-out nodes* represent the input and output parameters at the called procedure. They are control dependent on the procedure's entry node.

They also introduce additional edges to link the program dependence graphs together:

- *Call edges* link the call-site nodes with the procedure entry nodes.
- *Parameter-in edges* link the actual-in nodes with the formal-in nodes.
- *Parameter-out edges* link the formal-out nodes with the actual-out nodes.

Finally, *summary edges* are used to represent the transitive dependences due to calls. A summary edge is added from an actual-in node *A* to an actual-out node *B*, if there exists a path of control, data and summary edges in the called procedure from the corresponding formal-in node *A'* to the formal-out node *B'*. Fig. 3.3 shows how the summary edges can be used to simulate the effects of a call without descending into the subgraph of the called procedure. The graph contains a summary edge from the actual-in node of *z* at the call site of *Add* to the actual-out node of *z* because there is a path (via data dependences) in the called procedure from the formal-in node of *a* to the formal-out node of *a*. Likewise there is a summary edge from the second actual-in node to the actual-out node of *z*. When computing the slice for the actual-out node of *z*, it suffices to follow the summary edges backwards in order to visit the actual-in nodes on which the value of *z* depends. It is not necessary to descend into the graph of procedure *Add*.

```
PROCEDURE Add (VAR a: INTEGER; b: INTEGER);
BEGIN
   a := a + b
END Add;

call of Add in Increment: Add(z, 1)
```

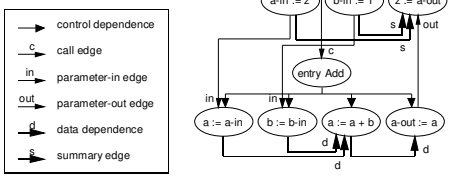

Fig. 3.3 - Summary edges

Summary edges permit movement across call sites without having to descend into the called procedures, while still regarding the effects of the called procedure. It is therefore not necessary to keep track of the calling context explicitly to ensure that only legal execution paths are traversed.

Horwitz et al. used a variation on the technique to compute the subordinate characteristic graphs of an attribute grammar's nonterminals [Ka80] in order to compute these summary edges. Livadas et al. [LivC94, LivJ95] proposed a simpler method of computing the summary edges.

Fig. 3.4 shows the system dependence graph of the program shown in Fig. 2.16. Control dependences are drawn with thin lines and broad arrows, call edges, par-in and par-out edges with thin lines and small arrows, and data dependences as well as summary edges are drawn with thick lines and big arrows.

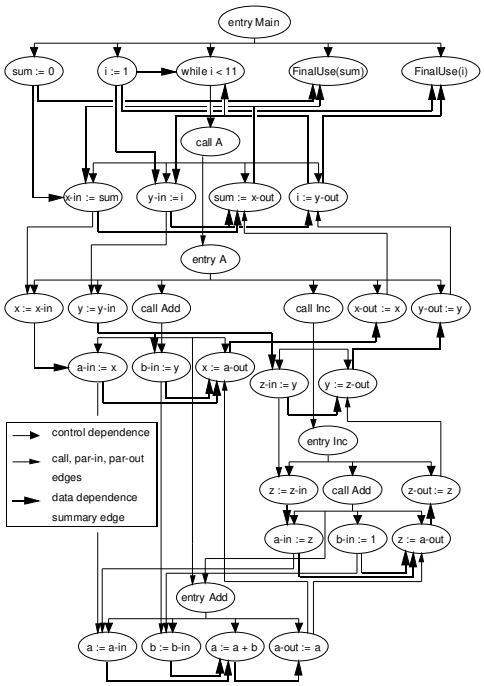

Fig. 3.4 - The system dependence graph for the program shown in Fig. 2.16

Interprocedural slicing can be implemented as a reachability problem over the system dependence graph. The transitive closure over all dependences yields a slice that does not regard the calling context and therefore contains irrelevant nodes. Horwitz et al. [HoRB90] developed a two-phase algorithm that computes precise interprocedural slices. In the following we give a brief outline of this algorithm (the slice shall be computed with respect to node *n* in procedure *P*):

- In the first phase, all edges except parameter-out edges (i.e., control and data dependences, summary, parameter-in and call edges) are followed backwards starting with node *n* in procedure *P*. All nodes are marked, that either reach *n* and are in *P* itself or in procedures that (transitively) call *P*, i.e. the traversal ascends from

procedure *P* upwards to the procedures that called *P*. Since parameter-out edges are not followed, phase 1 does not "descend" into procedures called by *P*. The effects of such procedures are not ignored, however; summary edges from actual-in nodes to actual-out nodes cause nodes to be included into the slice that would only be reached through the procedure call, although the graph traversal does not actually descend into the called procedure (see Fig. 3.3). The marked nodes represent all nodes that are part of the calling context of *P* and may influence *n*.

- In the second phase, all edges except parameter-in and call edges (i.e., control and data dependences, summary and parameter-out edges) are followed backwards starting from all nodes that have been marked during phase 1. Because parameter-in edges and call edges are not followed, the traversal does not "ascend" into calling procedures. Again, the summary edges simulate the effects of the calling procedures. The marked nodes represent all nodes in called procedures that induce summary edges.

Fig. 3.5 shows how the slice is computed for the formal-out parameter *z* in procedure *Inc*. Only the nodes and edges that are traversed during the first phase are shown.

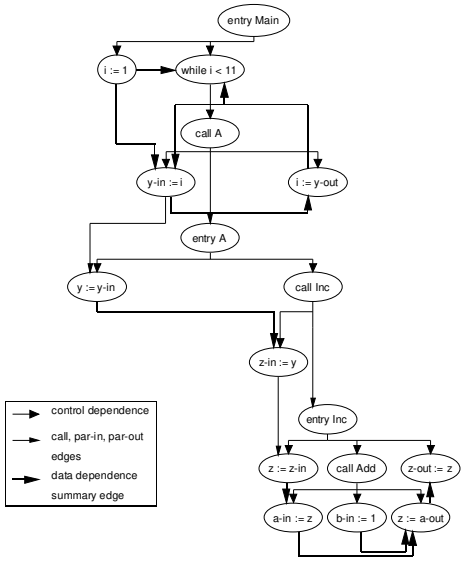

Fig. 3.5 - Nodes and edges visited during the first phase of computing the slice for the formal-out *z* in procedure *Inc*

Fig. 3.6 adds the nodes that are marked in the second phase (shown in bold). The complete slice consists of the nodes and edges visited during the two phases and the edges between them.

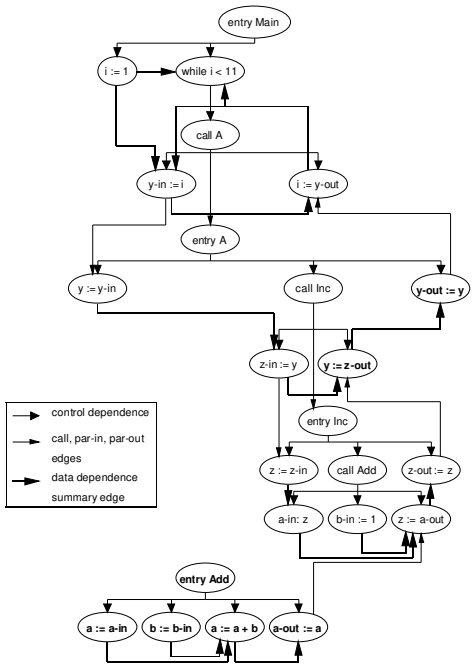

Fig. 3.6 - Nodes and edges visited during the second phase (shown in bold) of computing the slice for the formal-out *z* in procedure *Inc*

Fig. 3.7 shows a recursive implementation of the two-phase algorithm.

```
PROCEDURE SliceNodeInterproc (node: Node; excludeEdges: Set; visitedNodes: NodeSet);
BEGIN
  IF node is not marked THEN
    mark node as visited
    insert node into visitedNodes
    FOR all edges e leading from other nodes n to node DO
      IF kind of e is not in excludeEdges THEN
        SliceNodeInterproc(n, excludeEdges, visitedNodes)
      END
    END
  END
END SliceNodeInterproc;
```



```
PROCEDURE ComputeSlice (node: Node);
BEGIN
  (* phase 1: traverse control and data dependences, follow summary, par-in and call edges *)
  SliceNodeInterproc (node, {par-out}, visitedNodes);
  (* phase 2: traverse control and data dependences, follow summary, par-out edges *)
  FOR all nodes n in visitedNodes DO
    SliceNodeInterproc (n, {par-in, call}, visitedNodes)
  END
END ComputeSlice;
```

Fig. 3.7 - Interprocedural backward slicing algorithm

Forward Slicing

Horwitz et al. [HoRB90] showed that interprocedural forward slicing can be implemented in a very similar way to backward slicing where the edges are traversed from source to target. The first phase ignores parameter-in and call edges but follows parameter-out edges (thus ascends into calling procedures), whereas the second phase ignores parameter-out edges but follows parameter-in and call edges (thus descends into called procedures).

Dynamic Slicing

Agrawal and Horgan presented the first algorithm for finding dynamic slices using dependence graphs [AgH90]. The first approach to compute dynamic slices is to mark nodes and edges as the corresponding parts of the program are executed. After execution the slice is computed by applying the static slicing algorithm restricted to only marked nodes and the edges that connect them. Since multiple executions of a particular node are summarized by marking it once for all executions, these executions cannot be distinguished during analysis which makes the slices not as precise as possible. Another approach is to produce a *dynamic dependence graph* from the execution history that contains a node for each occurrence of a statement in the execution history along with only the executed edges. However, the dynamic dependence graph may be unbounded in length. Therefore Agrawal and Horgan also introduced the more economical version of a reduced dynamic dependence graph.

3.2.3 Computation of Summary Edges

Livadas et al. [LivC94, LivJ95] proposed a simpler method for the computation of summary edges. The basic idea is that for leaf procedures (procedures that do not call any other procedures) the summary edges can be computed via intraprocedural slicing, i.e. by following data dependences and control dependences backwards from the formal-out node. Summary edges are necessary from all formal-in nodes that are visited by this traversal to the formal-out node from which the traversal started. As long as there is no recursion, this idea can be applied to programs with procedure calls by analyzing the procedures

recursively when they are encountered. The summary edges of a procedure are computed as soon as the procedure is encountered:

- If a procedure *P* contains a call to another procedure *Q*, processing of *P* is suspended, and *Q* is processed. This process is continued until a procedure *R* is encountered that is either a leaf procedure or that has already been solved.
- If *R* is a leaf procedure, summary edges are computed directly via intraprocedural slicing. The processing of the calling procedure is then resumed.
- If *R* has already been solved, there is no reason to descend into the procedure again; subsequent calls to a solved procedure need only have the summary edges reflected (i.e. copied) to the call site.

We will illustrate this method by considering a program that consists of four procedures *M, A, B*, and *C* with calls as indicated in Fig. 3.8 (*M* calls *A* and *C, A* calls *B*, *B* calls *C, C* is a leaf procedure).

```
M  =  A C.
A  =  B.
B  =  C.
C  =  .
```

Fig. 3.8 - Sample program abstraction

The computation of the summary edges starts with processing procedure *M*. The first call encountered is the call of procedure *A*. Since procedure *A* has not been solved, we descend into *A*. During processing *A*, the call to procedure *B* is encountered. Therefore we descend into the unsolved procedure *B*. Again, procedure *B* is not a leaf procedure and we descend into procedure *C*. But *C* is a leaf procedure and is solved immediately. The summary edges from *C* are reflected onto the call site of *C* in *B*. Similarly *B* can now be solved since it contains no more calls; its summary edges are reflected back to the call site of *B* in *A*. Now *A* can be solved and its summary edges are reflected back to the call site of *A* in *M*. Processing of *M* is resumed until the call to *C* is encountered. But *C* has already been solved; therefore its summary edges are simply reflected. Since *M* contains no more calls, *M* can be solved. Fig. 3.9 shows the call trace for this process.


```
Compute(M)
call A          =>  Compute(A)
                    call B          =>  Compute(B)
                                        call C          =>  Compute(C)
                                                            C is solved
                                                        <= reflect information of C
                                                           onto call in B
                                        B is solved
                                    <= reflect information of B onto call in A
                    A is solved
                <= reflect information of A onto call in M
call C
                <= reflect information of C onto call in M
M is solved
```

Fig. 3.9 - Call trace of the computation of summary edges for a program without recursion (shown in Fig. 3.8)

The algorithm just described does not work well in the case of recursive procedures. The reason is that in the absence of recursion, it is guaranteed that a leaf procedure will be encountered that can be solved completely and its information can be reflected to its caller. In the case of recursion, the obtained information may be incomplete even if one processes a procedure in its entirety.

Therefore, recursion must be detected upon calls. Then only the partial information is reflected back to the call site and one does not descend into the (only partially solved) procedure. Finally if a procedure has been processed entirely, its information is reflected back on all its call sites. If the procedure has been part of a recursion chain, one has to iterate over the set of procedures that were part of the recursion. An iteration over a procedure *P* merely reflects the summary edges of all procedures *Q* that are called in *P* but does not descend into *Q*. Iteration is performed over the set of procedures until no more changes to the calculated information are found. At this point, all procedures of the iteration set are solved. Fig. 3.10 shows the a sample program with recursion.

```
M  =  A B.
A  =  B D.
B  =  C E.
C  =  C A.
D  =  .
E  =  F.
F  =  .
```

Fig. 3.10 - Sample program abstraction

Fig. 3.11 shows the call trace of the sample program of Fig. 3.10.

```
Compute(M)
call A          => Compute(A)
                   call B          => Compute(B)
                                      call C          => Compute(C)
                                                         call C          => recursion detected (C)
                                                                            <= partial information of C
                                                                               reflected
                                                         call A          => recursion detected
                                                                               (A, B, C)
                                                                            <= partial information of A
                                                                               reflected
                                                         C partially solved
                                                      <= (partial) solution of C reflected
                                      call E          => Compute(E)
                                                         call F          => Compute(F)
                                                                            F solved
                                                                         <= solution of F reflected
                                                         E solved
                                                      <= solution of E reflected
                                   <= solution of B reflected
                   call D          => Compute(D)
                                      D solved
                                   <= solution of D reflected
                   A partially solved
                   Iterate over all procedures of the recursion (A, B, C) until there are no changes
                                   => Iterate over A
                                                      <= reflect information of B and D
                                   => Iterate over B
                                                      <= reflect information of C and E
                                   => Iterate over C
                                                      <= reflect information of C and A
                                      A, B and C are solved completely
                                   <= complete information of A has been reflected to
                                      all call sites of A
                                   <= complete information of B has been reflected to
                                      all call sites of B
                                   <= complete information of C has been reflected to
                                      all call sites of C
                <= solution of A reflected
call B
                <= solution of B reflected
M solved
```

Fig. 3.11 - Call trace of the computation of summary edges for a program with recursion (shown in Fig. 3.10)

3.2.4 Enhancing Slicing Accuracy

Horwitz et al. [HoRB90] noted that some imprecision is introduced if parameter nodes are generated for every parameter, regardless of whether it is changed by the called procedure or not. They use interprocedural data flow analysis to compute the sets of non-local variables and parameters that are used and modified by a procedure:

GMOD(P) is the set of non-local variables and parameters that might be modified by *P* itself or by a procedure (transitively) called from *P*.

GREF(P) is the set of non-local variables and parameters that might be referenced by *P* itself or by a procedure (transitively) called from *P*.

For each procedure *P*, there is one formal-in and one actual-in node for each variable or parameter in GMOD(P) ∪ GREF(P), and there is one formal-out and one actual-out node for each variable or parameter in GMOD(P).

Livadas et al. [LivC94, LivJ95] do not use interprocedural data flow analysis to derive this information but rather derive it during the construction of the system dependence graph. They further restrict the number of necessary parameter nodes depending on how the parameters are used within the called procedure:

- When a reference parameter is never modified (i.e., there is no path where the variable is defined), no formal-out and actual-out nodes are necessary.
- When a reference parameter is always modified (i.e., the variable is defined on every path), formal-out and actual-out nodes are necessary. At the call-site, a killing definition can be generated for the actual-out node.
- When a reference parameter is sometimes modified or when it is not known how it is used, formal-out and actual-out nodes are necessary. At the call-site a non-killing definition must be generated for the actual-out node.
- For value parameters no formal-out and actual-out nodes are necessary.

4 Implementation

This chapter describes the implementation of the Oberon Slicing Tool (OST), the underlying data structures and the algorithms for the computation of control flow and data flow information and for slicing itself. The computation of precise control flow and data flow information is a prerequisite of precise interprocedural slicing. In fact, it is the most difficult part, since slicing itself is simply a traversal of the computed dependences.

4.1 Overview

The OST (see [OST] for information about the OST, and [Ste98a, Ste98b] for a technical description) can compute static backward slices of Oberon-2 programs. We did not restrict the language in any kind which means that we had to cope with structured types (records and arrays), global variables of any type, objects on the heap, side-effects of function calls, nested procedures, recursion, dynamic binding due to type-bound procedures and procedure variables, and modules.

The underlying data structures of the OST are the abstract syntax tree (AST) and the symbol table constructed by the front-end of the Oberon compiler [Cre90]. Additional information (such as control and data dependences) is added to the nodes of the syntax tree and the symbol table. The nodes of the AST represent the program at a fine granularity, i.e. one statement can consist of many nodes (function calls, operators, variable usages, variable definitions, etc.). The target and origin of control and data dependences are nodes of the AST, not whole statements. This allows for fine-grained slicing (cf. [Ern94]), therefore we call our slicing method expression-oriented in contrast to statement-oriented slicing.

Our slicing algorithm is based on the two-pass slicing algorithm of Horwitz et al. [HoRB90] where slicing is seen as a graph-reachability problem. This algorithm uses summary information at call sites to account for the calling context of procedures. We compute the summary information by a variation of the algorithm of Livadas et al. [LivC94, LivJ95]. In order to slice the program with respect to the starting node, the graph representation of the program is traversed backwards from the starting node along control and data dependence edges. All nodes that could be reached belong to the slice because they potentially affect the starting node.

We extended the notion of interprocedural slicing to intermodular slicing. Information that has been computed once can be stored and reused when slicing other modules that import previously sliced modules. Furthermore, we support object-oriented features such as inheritance, polymorphism, and dynamic binding. Since the construction of summary

information at call sites is the most costly computation, it is worthwhile to cache this information in a repository and to reuse as much information as possible from previous computations.

Zhang and Ryder showed that alias analysis in the presence of procedure variables is NP-hard in most cases [ZhR94]. This justifies to use safe approximations since exact algorithms would be prohibitive for an interactive slicing tool where the maximal response time must be in the order of seconds. In addition to conservative alias analysis we use feedback from the user to compute more precise data flow information. The user can for example restrict the dynamic type of polymorphic variables and thereby disable specific destinations at polymorphic call sites. He can also restrict the sets of possible aliases at definitions.

4.2 Algorithm

Before we can derive slices of a program, we have to build up a graph representation of the program that closely models its semantics. We want to derive precise information about the possible run-time executions of the program at compile time. This is not possible in general, since the values of input parameters are not known, just as it is not known which branches will be taken and how many times loops will be executed. But we can compute information that is useful for debugging and necessary for slicing, e.g. we can derive the call destinations of dynamically bound calls, as well as the usage of parameters and precise reaching definitions.

The following outline shows the necessary steps to compute the information that is necessary to perform slicing.

1) Build the abstract syntax tree and the symbol table of the program under consideration.
2) Build its class hierarchy.
3) Compute its control flow information.
   3.1) Compute the control dependences.
   3.2) Link the call sites with all possible call destinations.
4) Compute its data flow information.
   4.1) Compute the used and defined variables.
      4.1.1) Compute the used and defined variables of each node and of each procedure.
      4.1.2) Compute the additional parameters of each procedure.
      4.1.3) Add parameter edges between the actual and formal parameters.
      4.1.4) Handle definitions of possible aliases.
   4.2) Compute the reaching definitions.

4.2.1) Compute the definition sets of each variable.
4.2.2) Compute the *gen/kill* sets of each defining node.
4.2.3) Combine the *gen/kill* sets of the particular nodes to the *gen/kill* set of statement sequences.
4.2.4) Compute the reaching definitions for each using node.
4.2.5) Compute the parameter usage information.
4.2.6) Compute the summary edges for each procedure.

The first step is accomplished by a slightly modified version of the front end of the Oberon-2 compiler [Cre90]. The second step traverses the symbol table and collects for each class all its direct extensions, as well as the set of all its fields. The third step computes control flow information for each procedure of the program. This is explained in Section 4.4. The fourth step computes data flow information for each procedure of the program. This is explained in Section 4.5, where Section 4.5.1 describes the computation of used and defined variables with definitions via assignments and reference parameters at calls, definitions of record fields and array elements and handling of aliases. Section 4.5.2 describes the computation of reaching definitions by first computing the definition sets of all variables and the *gen/kill* sets. Then we explain in detail the data flow equations adapted for our fine-grained representation. Finally we describe the computation of the parameter usage information and the computation of summary edges. Section 4.6 describes the algorithms used for slicing itself. Section 4.7 explains how object-oriented features are supported. Section 4.8 shows the modularization of the Oberon Slicing Tool and describes the interfaces of the most important modules.

4.3 Data Structures

The underlying data structures of the OST are the abstract syntax tree and the symbol table constructed by the front-end of the Oberon compiler (for a technical description see [Cre90] and [Ste98a]). In the following we will explain the most important internal data structures:

- Global and local variables, value and reference parameters, constants, record fields, named types, all kinds of procedures, modules, and scopes are represented by objects of type *Object*.
- Named and anonymous type structures are represented by objects of type *Struct*.
- Nodes of the abstract syntax tree are of type *Node*.
- Information about procedures is represented by objects of type *ProcInfo*.

We will not explain the auxiliar data structures *Nodes, ObjArr, StructArr, Dependences, SetArr, NodeArr, HashTable, Definitions,* and *AccessArr* here (see [Ste98a] for details).

The object declaration is as follows (fields added for slicing purposes are shown in bold face):

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD
   left, right: Object;          (* for binary search tree structure                *)
   link, scope: Object;          (* link for sequence of objects, declaring scope   *)
   name: OPS.Name;               (* name of the object, under which it is found in the
                                    binary search tree                              *)
   mode: SHORTINT;               (* Var for global or local variables and value parameters
                                    VarPar for reference parameters
                                    Con for constants
                                    Fld for record fields
                                    Typ for named types
                                    LProc, XProc, SProc, TProc, CProc for local,
                                      external, standard, type-bound, and code
                                      procedures
                                    Mod for modules
                                    Head for scope anchors *)
   vis: SHORTINT;                (* internal, external, external read-only          *)
   typ: Struct;                  (* type of the object                              *)
   ...
   nodes: Nodes;                 (* AST nodes that use or define the object         *)
   procInfo: ProcInfo;           (* for procedure only: additional information,
                                    see below                                       *)
   assignedToProcVar: BOOLEAN;
   mark: SHORTINT;               (* marked during slicing if slice.mark = obj.mark  *)
   level: SHORTINT;              (* 0 for global scope, 1 for scope of local procedures,
                                    etc.                                            *)
   mod: Object;                  (* containing module object                        *)
   expanded: BOOLEAN;            (* TRUE for arrays and records that have been
                                    expanded for data flow analysis                 *)
   components: ObjArr;           (* expanded components: fields of a record or
                                    elements of an array                            *)
   ...
END ;
```

The symbol table is organized as binary search trees that are linked together. Each scope (global scope of a module, local scope of procedures) is represented by a scope anchor. When looking up objects by name, the scopes are traversed from the innermost scope outwards. Fig. 4.1 shows the scopes with the accessible objects for the local procedure *ProcessStatSeq* of procedure *Slicer.ControlFlow*, beginning with the scope of local variables of procedure *ProcessStatSeq*, then the scope of intermediate variables (declared in the outer procedure *ControlFlow*) and finally the scope of global variables.

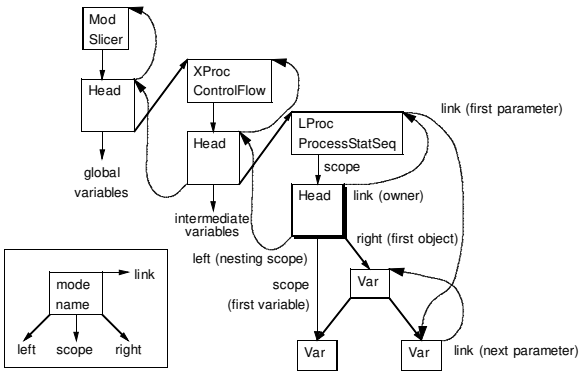

Fig. 4.1 - Scope of accessible objects of procedure *Slicer.ControlFlow.ProcessStatSeq*

Named and anonymous types are described by records of type *StrDesc* (fields added for slicing purposes are shown in bold face):

```
StrDesc = RECORD
   form: SHORTINT;               (* Undef, Byte, Bool, Char, SInt, Int, LInt, Real, LReal,
                                    Set, String, NilTyp, NoTyp, Pointer, ProcTyp, Comp*)
   comp: SHORTINT;               (* Basic, Array, DynArr, Record *)
   BaseTyp: Struct;              (* extended type for records, element type for arrays,
                                    base type for pointers, return type for procedures *)
   link: Object;                 (* link for sequence of objects (parameter list or field
                                    list)                                           *)
   strobj: Object;               (* for named types: object, struct.strobj.typ = struct *)
   ...
   mod: Object;                  (* containing module object                        *)
   extensions: StructArr;        (* direct extensions                               *)
   fields: ObjArr;               (* for records: all fields (including fields of base
                                    classes)                                        *)
   mark: SHORTINT                (* marked during slicing if slice.mark = str.mark  *)
   ...
END ;
```

Oberon-2 allows single inheritance. Therefore each class can have at most one base class. The field *BaseTyp* of a structure node is used to model the inheritance relationship in the upwards direction. Additionally, each extending type is registered at the extended base type. The field *extensions* of a structure holds all direct extensions of the type. The following methods operate on the extension relation between classes:

```
PROCEDURE IsExtended (typ: Struct): BOOLEAN;
PROCEDURE FindMethod (name: ARRAY OF CHAR; typ: Struct): Object;
PROCEDURE FindOverriddenMethod (name: ARRAY OF CHAR; typ: Struct): Object;
PROCEDURE IsOverridden (name: ARRAY OF CHAR; typ: Struct): BOOLEAN;
```


- *IsExtended(t)* returns TRUE if there are extensions of type *t*.
- *FindMethod(n, t)* returns the method object for the method with the name *n* of type *t*. If such a method does not exist, it returns NIL.
- *FindOverriddenMethod(n, t)* returns the method object for the method with the name *n* of type *t* or any subtype of *t*. If such a method does not exist, it returns NIL.
- *IsOverridden(n, t)* returns TRUE if any extension of type *t* overrides the method with the name *n*.

The front end of the Oberon-2 compiler translates the source code into a binary tree of elements of type *Node*, all having the same form (fields added for slicing purposes are shown in bold face):

```
NodeDesc = RECORD
   left, right: Node;            (* for binary tree structure of the AST *)
   link: Node;                   (* for sequence of nodes (statement sequence,
                                    list of parameters)                             *)
   class: SHORTINT;              (* Nvar, Nvarpar,... Nifelse, Nwhile,... Nfpar,
                                    NcallSite,...                                   *)
   subcl: SHORTINT;              (* subclass, e.g. if class = Nassign: incfn, decfn,
                                    newfn,...                                       *)
   ...
   typ: Struct;                  (* type of the node                                *)
   obj: Object;                  (* e.g. for Nvar: used or defined object           *)
   conval: Const                 (* position in the source code or other constant
                                    value                                           *)
   mark: SHORTINT;               (* marked during slicing if slice.mark = node.mark *)
   procInfo: ProcInfo;           (* for procedure entry nodes: additional information *)
   usedObjs: ObjArr;             (* set of objects used at this node                *)
   definedObjs: ObjArr;          (* set of objects defined at this node             *)
   dependences: Dependences;     (* sets of dependences onto other nodes            *)
   gen, kill, in: SetArr;        (* gen/kill and in sets of the node                *)
   choice: SetArr;               (* set of enabled dynamic types                    *)
   aliases: ObjArr;              (* set of aliases                                  *)
   enabledAliases: SetArr;       (* bitset of enabled aliases                       *)
END ;
```

The dependences between nodes are implemented by pointers from the target to the origin, since they are traversed in this direction for backward slicing.

A *ProcInfo* object stores additional information about a procedure object:

```
ProcInfoDesc = RECORD
   fpars: Node;                  (* list of formal parameter nodes (formal-in nodes and
                                    formal-out nodes)                               *)
   callSites: Node;              (* list of actual call sites                       *)
   calls: NodeArr;               (* calls occurring in the described procedure      *)
   procExit: Node;               (* procedure exit node                             *)
   enter: Node;                  (* procedure entry node                            *)
   procObj: Object;              (* procedure or module object *)
   in, out: SetArr;              (* reaching definitions before first and after last
                                    statement                                       *)
   objs: Objects;                (* sets of used and defined variables              *)
   definitionsHT: HashTable;     (* hash table of definitions                       *)
   varDefs: Definitions;         (* sets of killing and non-killing definitions per object *)
   accesses: AccessArr;          (* sets of variable uses and definitions           *)
END ;
```

fpars is the list of formal parameter nodes:

- For ordinary input parameters (value parameters) there is a formal input parameter node (*node.class = Nfpar, node.subcl = inPar*).
- For ordinary reference parameters (VAR parameters) there is a pair of two formal parameter nodes which reference both the same object (*node.class = Nfpar, node.subcl* is once *inPar* and once *outPar*).
- For additional parameters due to accessed global or intermediate variables there is a pair of two formal parameter nodes which reference both the same object (*node.class = Nfpar, node.subcl* is once *additionalInPar* and once *additionalOutPar*). At the call sites, an additional actual parameter node (*node.class = Nvarpar, node.subcl = additionalPar*) is added to the list of actual parameters of the call node. All these nodes refer to the same symbol table entry for the parameter object.

callSites is the list of call sites calling this procedure (*callSite.class = NcallSite*). *calls* is an array of all calls contained in this procedure (*call.class IN {Ncall, Ndyncall}*). Fig. 4.2 shows the bidirectional call relation between procedures.

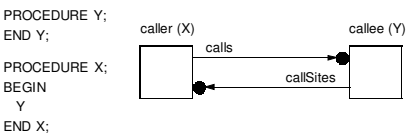

Fig. 4.2 - Bidirectional call relation between procedures

In Fig. 4.3, we show a procedure call with the list of actual parameter nodes and the called procedure with the list of formal parameter nodes. Symbol table entries are shown in rectangles with rounded corners.


```
PROCEDURE Print* (VAR str: ARRAY OF CHAR; i: INTEGER);
...
Print(s, 4);
```

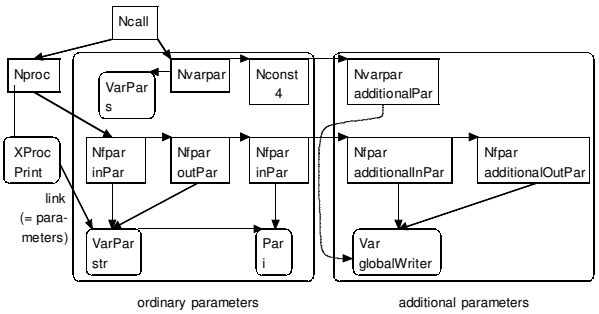

Fig. 4.3 - Procedure call with ordinary and additional parameters

procExit is the procedure exit node of this procedure (*procExit.class = NprocExit*). *enter* is the entry node of this procedure (*enter.class = Nenter*), i.e. a reference to the syntax tree. *procObj* is the procedure or module object (*procObj.mode IN {LProc, XProc, CProc, TProc, Mod}*), i.e. a reference to the symbol table. *in* is the set of definitions that reach the first statement of the procedure, *out* is the set of definitions that leave the last statement of the procedure. *objs* provides access to two collections: the set of variables that are used by this procedure and the set of variables that are defined by this procedure. *definitionsHT* is the hash table of definitions (each definition in this procedure is entered in this hash table; the elements of the *gen* and *kill* sets as well as the elements of the *in* and *out* sets are the indices of the definitions within this hash table). *varDefs* is an array of triplets *<o mustAssigns mayAssigns>* (representing the sets of killing definitions *mustAssigns* and the sets of all (killing or non-killing) definitions *mayAssigns* of object *o*). *accesses* is an array of tuples *<o n>* (representing an access to the object o at node *n*).

4.4 Computation of Control Flow Information

In Oberon-2, the computation of intraprocedural control flow information is - in most cases - very easy, since Oberon-2 contains mainly constructs for structured control flow. The control dependences therefore simply reflect the program's nesting structure. In Example 4.1 statements *stat* are control dependent on the guarding expressions *expr*. *stat2* and *stat3*

are control dependent on *expr2* which itself is again control dependent on *expr*.

Example 4.1:

```
IF expr THEN stat ELSIF expr2 THEN stat2 ELSE stat3 END ;
WHILE expr DO stat END ;
```

During slicing, we usually need to traverse the control dependences backwards, therefore, they are implemented as pointers from the destination to the source (in the opposite direction of the arrows of the figures in chapters 2 and 3). Fig. 4.4 shows the AST for the code in Example 4.2. Every node has 5 pointers. *left* and *right* point to the sons of the node. *link* points to the next statement in a statement sequence. Control dependences are drawn with thick lines and big arrows, whereas the other pointers are drawn with thin lines and small arrows. If a node refers to an object (e.g. *Nvar*, *Nproc*, and *Nfield* nodes), *object* points to the respective symbol table entry (e.g., *Read, val, p, left,* and *right*). The upward pointer of a node *n* points to the node on which *n* is control dependent.

Example 4.2:

```
Read(val);
p := tree;
WHILE (p # NIL) & (p.val # val) DO
   IF val < p.val THEN p := p.left ELSE p := p.right END
END ;
RETURN p
```

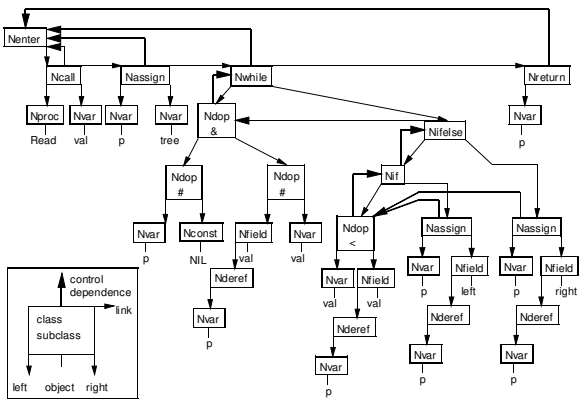

Fig. 4.4 - AST of the statement sequence in Example 4.2 with control dependences

We compute control dependences for a statement sequence by a recursively descending traversal. Each statement is handled in an appropriate way described below. After a short explanation of the language construct (usually given by a short quotation of the Oberon-2 language report [MöWi91]), a figure shows the syntax tree and control flow graph for a piece of source code. We do not construct or use the control flow graphs, but only show them to let the reader compare them with our representation. Finally, a table summarizes the control dependences.

Assignment

Assignment nodes in the AST represent ordinary assignments but also built-in functions such as NEW, INC, DEC, INCL, EXCL, COPY, SYSTEM.GET, SYSTEM.PUT, etc. No control dependences are inserted for assignments.

IF

"If statements specify the conditional execution of guarded statement sequences. The Boolean expression preceding a statement sequence is called its guard. The guards are evaluated in sequence of occurrence, until one evaluates to TRUE, whereafter its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the symbol ELSE is executed, if there is one."

Example 4.3:

```
IF expr1 THEN stat1
ELSIF expr2 THEN stat2
ELSIF expr3 THEN stat3
ELSE stat4
END
```

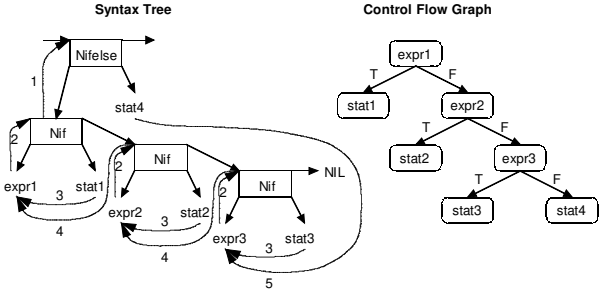

Fig. 4.5 - Syntax tree and control flow graph for an IF statement

We insert a control dependence labeled 1 from the Nif node of the first alternative to the statement node (Nifelse node). The control dependences labeled 2 point from the root of the expression trees to the Nif node. They are the destinations of the control dependences from the directly nested statements (labeled 3) and from the Nif node that represents the next alternative (labeled 4). The directly nested statements of the ELSE branch have also control dependences (labeled 5) on the last test. When slicing for *stat2* of Example 4.3, *stat2*, *expr2*, the guarding Nif node, *expr1*, the guarding Nif node and the *Nifelse* node would be reached via control dependences. Table 4.1 summarizes these control dependences.

	From	To
1	Nif	Nifelse
2	expr of Nif	Nif
3	directly nested statements of THEN	guarding expr of Nif
4	following Nif (representing ELSIF)	expr of preceding Nif
5	directly nested statements of ELSE	last expr

Table 4.1 - Control dependences of an IF statement

CASE

"Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then that statement sequence is executed whose case label list contains the obtained value. The case expression must either be of an integer type that includes the types of all case labels, or both the case expression and the case labels must be of type CHAR. Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is selected, if there is one, otherwise the program is aborted."

Example 4.4:

```
CASE expr OF
  case1: stat1
| case2: stat2
ELSE stat3
END
```


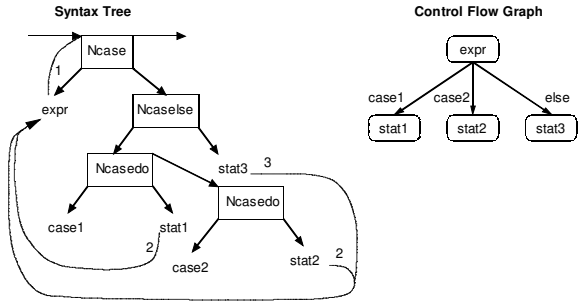

Fig. 4.6 - Syntax tree and control flow graph of a CASE statement

We insert a control dependence labeled 1 from the root of the expression of the CASE statement to the statement node (Ncase node). There are control dependences from the directly nested statements of all alternatives (labeled 2) and of the ELSE branch (labeled 3) to the expression of the CASE statement. Table 4.2 summarizes these control dependences.

	From	To
1	expr of Ncase	Ncase
2	directly nested statements of Ncasedo	expr of Ncase
3	directly nested statements of ELSE	expr of Ncase

Table 4.2 - Control dependences of a CASE statement

WITH

"With statements execute a statement sequence depending on the result of a type test and apply a type guard to every occurrence of the tested variable within this statement sequence."

Example 4.5:

```
WITH test1 DO stat1
| test2 DO stat2
ELSE stat3
END
```


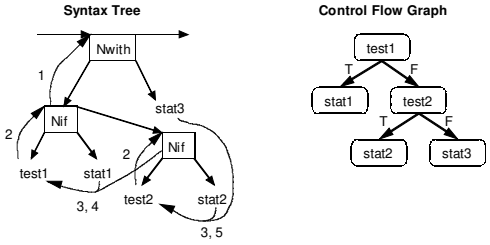

Fig. 4.7 - Syntax tree and control flow graph of a WITH statement

We insert a control dependence labeled 1 from the Nif node of the first alternative to the statement node (Nwith node). The control dependences labeled 2 point from the root of the expression trees to the Nif node. They are the destinations of the control dependences from the directly nested statements (labeled 3) and from the Nif node that represents the next alternative (labeled 4). The directly nested statements of the ELSE branch also have control dependences (labeled 5) on the last test. Table 4.3 summarizes these control dependences.

	From	To
1	Nif	Nwith
2	expr of Nif	Nif
3	directly nested statements of alternative	guarding expr of Nif
4	following Nif	expr of preceding Nif
5	directly nested statements of ELSE	last expr

Table 4.3 - Control dependences of an WITH statement

WHILE

"While statements specify the repeated execution of a statement sequence while the Boolean expression (its guard) yields TRUE. The guard is checked before every execution of the statement sequence."

Example 4.6:

```
WHILE expr DO
   stat
END
```

Fig. 4.8 - Syntax tree and control flow graph of a WHILE statement

We insert a control dependence labeled 1 from the root of the expression to the statement node (Nwhile node). There are control dependences from the directly nested statements (labeled 2) to the expression. The control dependence labeled 3 points from the statement node back to the root of the expression. Table 4.4 summarizes these control dependences.

	From	To
1	expr of Nwhile	Nwhile
2	directly nested statements of Nwhile	expr of Nwhile
3	Nwhile	expr of Nwhile

Table 4.4 - Control dependences of a WHILE statement

REPEAT

"A repeat statement specifies the repeated execution of a statement sequence until a condition specified by a Boolean expression is satisfied. The statement sequence is executed at least once."

Example 4.7:

```
REPEAT
   stat
UNTIL expr
```

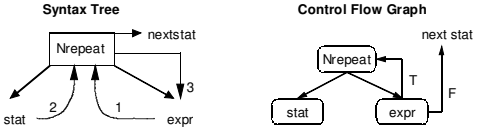

Fig. 4.9 - Syntax tree and control flow graph of a REPEAT statement

We insert a control dependence labeled 1 from the root of the expression to the statement node (Nrepeat node). There are control dependences from the directly nested statements (labeled 2) to the statement node. The control dependence labeled 3 points from the statement node back to the root of the expression. Table 4.5 summarizes these control dependences.

	From	To
1	expr of Nrepeat	Nrepeat
2	directly nested statements of Nrepeat	Nrepeat
3	Nrepeat	expr of Nrepeat

Table 4.5 - Control dependences of a REPEAT statement

FOR

"A for statement specifies the repeated execution of a statement sequence while a progression of values is assigned to an integer variable called the control variable of the for statement."

Since the FOR statement is represented internally by an equivalent WHILE statement, we do not have to treat it specially.

Call

"A procedure call activates a procedure. It may contain a list of actual parameters which replace the corresponding formal parameters defined in the procedure declaration."

Procedure calls occur at the statement level. They represent transfers of control from the call site to the called procedure. In order to represent this transfer of control, the AST contains references from the call sites (Ncall nodes) to the symbol table entries of the destination of the call (procedure object which has a reference to its Nenter node).

Functions are procedures that return a result value. Function calls can be used as factors in expressions. On the other hand, expressions can be used at various places in Oberon programs, e.g. as operands, as parameters of procedure or function calls, as the return value of functions, and in the expressions of IF, CASE, WHILE, REPEAT, and FOR statements.

Example 4.8:

```
PROCEDURE Print* (VAR str: ARRAY OF CHAR; i: INTEGER);
...

Print(s, 4);
```


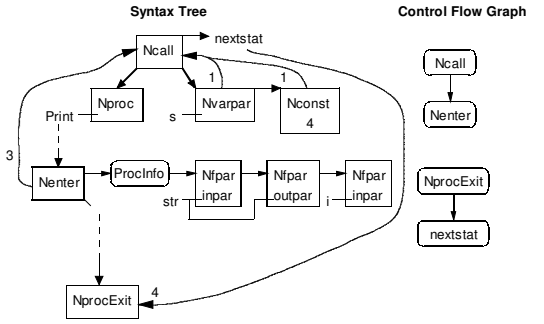

Fig. 4.10 - Syntax tree and control flow graph for a call statement

We insert control dependences labeled 1 from the actual parameters to the Ncall node. If the call is a function call (Ncall.typ # NoTyp), there is a control dependence labeled 2 from the Ncall node to the statement node. The control dependence labeled 3 from the Nenter node of the called procedure to the call node and the control dependence labeled 4 from the next statement to the procedure exit node of the called procedure are not represented explicitly, but are handled implicitly by the interprocedural slicing algorithm. Table 4.6 summarizes these control dependences.

	From	To
1	actual parameters	Ncall
2	for function calls: Ncall	statement node
3	Nenter node of called procedure	Ncall
4	next statement	NprocExit node of called procedure

Table 4.6 - Control dependences of a call

For dynamically bound calls, Ndyncall nodes are used to represent all possible call destinations. Links are inserted from the actual Ncall node to all call destinations (Ndyncall nodes). Each of these links can be enabled or disabled via user interaction. Fig. 4.11 shows a call site of a procedure variable whose call destinations have been computed. Of the two call destinations *P* and *Q* only *Q* is enabled, *P* has been disabled via user interaction (see Section 5.2).

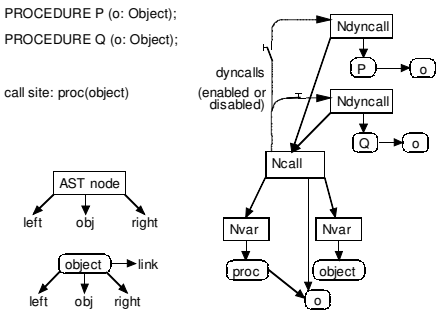

Fig. 4.11 - Dynamically bound procedure call with *Ndyncall* nodes

The set of possible call destinations is computed as follows:

- For method calls, one can distinguish between methods that can be bound statically and methods that must be bound dynamically.
  - If the receiver of a method call is a monomorphic variable (a variable that will always refer to objects of the same class at run time) and the called method can be determined at compile time (e.g. by *Class Hierarchy Analysis* [DGC94]), it can be bound statically. In Oberon-2, the type of the actual receiver can be a record or a pointer: Pointers can in general refer to different objects at run time, whereas records can refer to different objects at run time only if they are VAR parameters (reference parameters); receivers of a record type that are not VAR parameters (e.g. locally declared records) are known to be monomorphic.
  - One can bind method calls statically, if one can guarantee at compile time that there is only one call destination (this corresponds to an empty *Override* set of the method in the terms of [Bac97]). Since the analyzed programs are usually incomplete programs, one can guarantee this only in the following case: If the record type is not exported and one determines only one call destination, then this call destination will always remain the only one since the record type cannot be extended in another module.
  - Otherwise, one has to find all possible destinations of the calls. These are simply the methods of the statically known class of the receiver and all subclasses (this is a conservative assumption and can be improved by fast techniques such as *Rapid Type Analysis* [Bac97] or by other flow-sensitive techniques [PaR93]). One can either determine all call destinations (e.g. if the record type is not exported, but there are several destinations because the record type has been extended within the same module several times and the method has been overridden more than once) or not (e.g. if the record type has been exported and will potentially be extended in other modules).

- For calls of procedure variables, one can either determine the set of possible destinations by flow-sensitive analysis (e.g. by propagating the assigned procedures along all possible paths in the invocation graph [EGH94]) or one can approximate the set of possible destinations (which can be done much faster) with the following restrictions:
  1. A procedure must be assigned somewhere to a procedure variable. Otherwise it can never be the destination of a call via a procedure variable.
  2. The type of the procedure and the type of the procedure variable must *match*. This depends on the semantics of the programming language. In Oberon-2, two parameter lists only match if (see [MöWi91]),
     a) they have the same number of parameters, and
     b) they have either the same function result type or none, and
     c) parameters at corresponding positions have equal types, and
     d) parameters at corresponding positions are both either value or reference parameters.

RETURN

"A return statement indicates the termination of a procedure. It is denoted by the symbol RETURN, followed by an expression if the procedure is a function procedure. The type of the expression must be assignment compatible with the result type specified in the procedure heading.
Function procedures must be left via a return statement indicating the result value. In proper procedures, a return statement is implied by the end of the procedure body. Any explicit return statement therefore appears as an additional (probably exceptional) termination point."

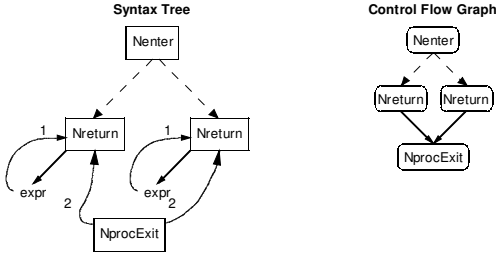

Fig. 4.12 - Syntax tree and control flow graph of a RETURN statement

We insert control dependences labeled 1 from the roots of the expressions to the statement node (Nreturn node), if the RETURN statement appears in a function procedure. The control dependences labeled 2 point from the procedure exit node to all Nreturn nodes of the procedure. Table 4.7 summarizes these control dependences.

	From	To
1	expr of Nreturn	Nreturn
2	NprocExit	Nreturn

Table 4.7 - Control dependences of a RETURN statement

LOOP / EXIT

"A loop statement specifies the repeated execution of a statement sequence. It is terminated upon execution of an exit statement within that sequence."
"An exit statement is denoted by the symbol EXIT. It specifies termination of the enclosing loop statement and continuation with the statement following that loop statement. Exit statements are contextually, although not syntactically associated with the loop statement which contains them."

Example 4.9:

```
LOOP
   stat1
   IF expr THEN EXIT END ;
   stat2
END
```

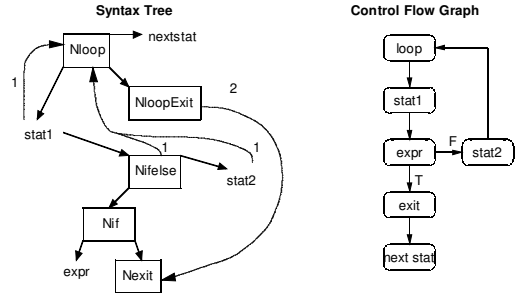

Fig. 4.13 - Syntax tree and control flow graph of a LOOP statement

We insert control dependences labeled 1 from the directly nested statements to the statement node (Nloop node). The control dependences labeled 2 point from the loop exit node to all Nexit nodes of the LOOP. Table 4.8 summarizes these control dependences.

	From	To
1	directly nested statements of Nloop	Nloop
2	NloopExit	Nexit

Table 4.8 - Control dependences of a LOOP statement

Whenever an exit node is encountered, a control dependence is inserted from the loop exit node of the enclosing loop to the exit node.

ASSERT and HALT

HALT statements explicitly terminate the program. ASSERT statements test a Boolean expression at run time. If this expression is not TRUE, the program is terminated. On the other hand, run-time type checks are performed as type tests, type guards, and as part of the WITH-statement and the assignment statement (when assigning to a VAR-record parameter). If these run-time type checks fail, the program is also terminated. In the AST, the HALT statement is represented by a Ntrap node, the ASSERT statement is resolved by an IF statement (e.g., ASSERT(b, 55) corresponds to IF ~b THEN HALT(55) END). Control dependences are inserted from the (global) halt node to all trap nodes of the program.

Other Sources for Traps

Other sources for traps are not handled; these include "division by 0", dereferencing a NIL-pointer, heap overflow, FPU error etc.

4.5 Computation of Data Flow Information

The goal of data flow analysis is precise information about which variable definitions reach which points in the program, i.e. we want to derive information about the flow of data at run time by static analysis. Conservative assumptions must be taken if the program uses conditional branches and iteration since we do not know at compile time which branches will be taken at run time and how many iterations there will be.

We insert *data dependence edges* from a node n_1 to a node n_2 of the syntax tree of the program iff all of the following conditions hold (similar to the definition of [HoRB90]):

1) n_1 defines variable *x*.
2) n_2 uses *x*.
3) Control can reach n_2 after n_1 via a path along which there is no intervening definition of *x*.

Additionally, we insert data dependence edges because of the fine granularity of our program representation

1) for assignment statements from the definition node on the left-hand side to all usage nodes and function call nodes of the statement,
2) for expressions of conditional and iterative statements from the guarding AST nodes (e.g., Nif, Nwhile, Nrepeat) to all usage nodes and function call nodes within the expression tree.

In Fig. 4.14, we show the AST of the statement sequence in Example 4.10 with encircled definitions of variables and data dependences labeled DD to the reaching definitions. The left-hand side of an assignment statement (i.e. variable *j* of the last statement) is data dependent on all used variables of the right-hand side of the assignment (in this case variable *i*). These variable nodes on the right-hand side are again data dependent on all reaching definitions. Nif nodes are data dependent on all variable usage nodes in the expression sub-tree.

Example 4.10:

```
Read(i);
IF i < 0 THEN i := -i END ;
j := i * 3;
```

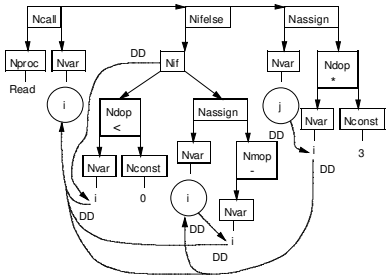

Fig. 4.14 - AST of the statement sequence in Example 4.10
with encircled definitions of variables, and data dependences (DD)

4.5.1 Computation of Used and Defined Variables

Before we can compute reaching definitions, we first have to determine the exact sets of used and defined variables of a procedure. We perform this in the following way:

1) First we compute the used and defined variables per node in the AST. This information is also collected for the whole procedure. A procedure may use and define a subset of the variables of its scope (including the additional parameters, see Section 4.2).
2) When we encounter a procedure call, we append additional formal parameter nodes for accessed intermediate and global variables to the list of formal parameters of the calling

procedure and corresponding additional actual parameter nodes to the list of actual parameters at the call site. We add control and data dependences for the actual parameter nodes and edges for parameter passing.
3) Finally, we handle aliases by inserting non-killing definitions of all possible aliases at definition nodes.

We compute this information for one procedure at a time. We handle recursion due to static and dynamic binding similar to the way described in Section 3.2.3.

Definition via Assignments

When a variable is defined by an assignment statement, the AST node representing the definition of the variable (in the following often called *defining node*) is given a new value by evaluating the right-hand side. It is data dependent on all variables and function calls whose values are used to compute the new value as illustrated by Fig. 4.15. Summary edges lead from the function call nodes to the input parameters that contribute to the return value of the function. The nodes upon which the variable is data dependent are collected by a top-down traversal of the sub-trees. The definition of the variable on the left-hand side is considered to be a killing definition. Additionally, non-killing definitions are generated for all variables that may be aliases of the defined variable. In Fig. 4.15, the left-hand side of the assignment is data dependent on the variable usage node of *m* and on the function call node of *Sum* on the right-hand side. The function call node of *Sum* has summary edges to both parameters *i* and *j*.

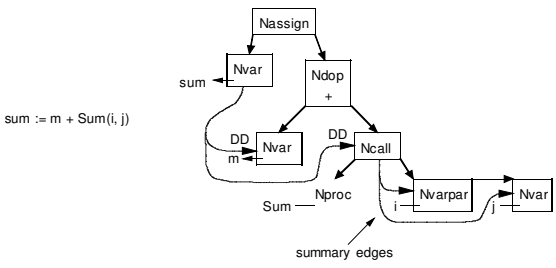

Fig. 4.15 - Data dependences for an assignment statement

Definition via Reference Parameters at Calls

There are several complications that have to be considered at a call site: First, a call may be bound statically or dynamically. Second, a call has ordinary parameters (as declared in the parameter list of the procedure) and additional parameters (see Fig. 4.3).

We combine the parameter usage information over all enabled call destinations in the following way:

- If a formal parameter is used by any of the call destinations, the value of the actual parameter is assumed to be used at the call site.
- If a formal reference parameter is defined by any of the call destinations (at least on some path), the actual parameter is assumed to be non-killingly defined at the call site.
- If a formal reference parameter is defined by all call destinations on all paths, the definition of the actual parameter is a killing definition instead of a non-killing definition.
- If the parameter usage information about a call destination is not available (e.g. because the procedure is a library function that cannot be processed because we do not have its source code), we have to take conservative assumptions: all parameters are assumed to be used, all reference parameters are assumed to be non-killingly defined.

If a formal parameter is assumed to be used, we traverse the expression tree at the call and insert data dependences from the variables and function calls used in the expression tree to all reaching definitions. When a formal reference parameter is modified, a definition is generated for the corresponding actual parameter. Since killing definitions lead to a more precise data flow information than non-killing definitions, we use the parameter usage information in order to generate as few definitions as possible, and if unavoidable as many killing definitions as possible, see Table 4.9:

Usage of Formal Reference Parameter	Kind of Definition of Corresponding Actual Parameter
not defined	no definition at all
defined on all control flow paths	killing definition
defined on some control flow paths	non-killing definition

Table 4.9 - Definitions via reference parameters

Definition of Records and Record Fields

Records can be seen as a whole or as the sum of their fields. Likewise, the definition of a record can be seen as a definition of the whole record or as the definition of all its fields. The symbol table stores structural information about records and their fields. Fig. 4.16 shows the symbol table objects for the declarations in Example 4.11. Variables *s* and *t* have the same type, they refer to the same structure node. The record fields *s.i*, and *t.i* are combined to the field *T.i*. This has the consequence that when field *s.i* is defined, field *t.i* is also considered to be defined.

Example 4.11:

```
TYPE T = RECORD i, j: INTEGER END ;
VAR s, t: T;
```


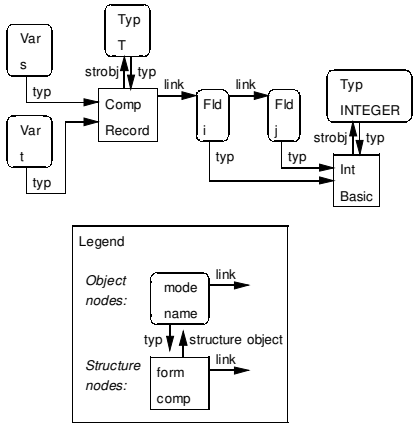

Fig. 4.16 - Symbol table entries for records and their fields

As outlined in Section 2.2.2, assignment to a record field is unambiguous as long as the address of the record is known at compile time and as long as there are no aliases. We can exclude the existence for aliases only for locally declared records. For VAR parameter records and for records that are allocated on the heap, aliases may exist. Therefore, we treat locally declared records in a different way than other records.

We expand locally declared records by copying the list of fields (including their base class fields) for each record. The nodes of the abstract syntax tree that formerly accessed the common fields *T.i* and *T.j* are patched to access the expanded fields *s.i, s.j, t.i* and *t.j*. Access to expanded records is handled as follows:

- For a killing/non-killing definition of the record (e.g. "s := ..."), we insert a killing/non-killing definition of the entire record (i.e. the symbol table object for *s*) and killing/non-killing definitions of all its expanded fields (i.e. the symbol table objects for *s.i* and *s.j*).
- For a use of the record (e.g. "... := s"), we insert a use of the entire record (i.e. *s*) and uses of all its expanded fields (i.e. *s.i* and *s.j*).
- For a killing/non-killing definition of a record field (e.g. "s.i := ..."), we insert a killing/non-killing definition of the expanded field (i.e. *s.i*) and a non-killing definition of the entire record (i.e. *s*), since the enclosing record is changed by the assignment.
- For a use of a record field (e.g. "... := s.i"), we insert a use of the expanded field (i.e. *s.i*).

Example 4.12 illustrates the handling of access to expanded records. Note that there are different symbol table objects for the fields *s.i* and *t.i*.

Example 4.12:

```
PROCEDURE ExpandedRecords;
   TYPE
      T0 = RECORD END ;
      T2 = RECORD i, j: INTEGER END ;
   VAR s0, t0: T0; s, t: T2; i: INTEGER;
BEGIN
   s0 := t0;          (* rhs: use of t0, t0.i, and t0.j; initial definitions are reaching
                         lhs: killing definition of s0, s0.i, and s0.j                    *)
   t0 := s0;          (* rhs: use of s0, s0.i, and s0.j; previous definitions are reaching
                         lhs: killing definition of t0, t0.i, and t0.j                    *)
   i := s.i;          (* rhs: use of s.i, initial definition of s.i is reaching           *)
   s.i := 0;          (* lhs: killing definition of s.i, non-killing definition of s      *)
   s := t;            (* rhs: use of t, t.i, and t.j; initial definitions are reaching
                         lhs: killing definition of s, s.i, and s.j; previous definition of s.i is killed *)
   s.i := 1;          (* lhs: killing definition of s.i, non-killing definition of s      *)
   t.i := 1;          (* lhs: killing definition of t.i, non-killing definition of t      *)
   s.j := 2;          (* lhs: killing definition of s.j, non-killing definition of s      *)
   i := s.i + s.j + t.i;  (* rhs: use of s.i, s.j and t.i, only last three definitions of fields are
                             reaching                                                     *)
   t := s             (* rhs: use of s, s.i, and s.j; last definitions of s.i and s.j are reaching.
                         The definitions due to record assignment s := t are no longer reaching
                         since all fields have been killingly defined.
                         lhs: killing definition of t, t.i, and t.j                       *)
END ExpandedRecords;
```

We do not expand all other records. Since the fields of those other records are combined by the fields of the record type, the definitions of the fields must no longer be killing. Access to non-expanded records is handled as follows:

- For a killing/non-killing definition of the record (e.g. "s := ..."), we insert a killing/non-killing definition of the entire record (i.e. *s*) and non-killing definitions of all fields of the record type (including base class fields; i.e. *T.i* and *T.j*), since the fields are changed by the assignment.
- For a use of the record (e.g. "... := s"), we insert a use of the entire record (i.e. *s*).
- For a killing/non-killing definition of a record field (e.g. "s.i := ..."), we insert a non-killing definition of the field of the record type (i.e. *T.i*) and a non-killing definition of the entire record (i.e. *s*), since the enclosing record is changed by the assignment.
- For a use of a record field (e.g. "... := s.i"), we insert a use of the field of the record type (i.e. *T.i*).

Example 4.13 illustrates the handling of non-expanded records.

Example 4.13:

```
TYPE T = RECORD i, j: INTEGER END ;

PROCEDURE NonExpandedRecords (VAR s, t: T);
   VAR i: INTEGER;
BEGIN
   i := s.i;          (* rhs: use of T.i, initial definition of T.i is reaching            *)
   s.i := 0;          (* lhs: non-killing definition of T.i and s,
                         lhs: non-killing definition of t (possible alias)                 *)
   s := t;            (* rhs: use of t;
                         lhs: killing definition of s, non-killing definition of T.i and T.j, previous
                         definition of T.i is not killed
                         lhs: non-killing definition of t (possible alias)                 *)
   s.i := 1;          (* lhs: non-killing definition of T.i and s
                         lhs: non-killing definition of t (possible alias)                 *)
   t.i := 1;          (* lhs: non-killing definition of T.i and t
                         lhs: non-killing definition of s (possible alias)                 *)
   s.j := 2;          (* lhs: non-killing definition of T.j and s
                         lhs: non-killing definition of t (possible alias)                 *)
   i := s.i + s.j + t.i;  (* rhs: use of T.i, T.j and T.i, all definitions of fields (including initial
                             definitions) are reaching                                     *)
   t := s             (* rhs: use of s, all definitions of record fields and records are reaching
                         lhs: killing definition of t, non-killing definition of T.i and T.j,
                         lhs: non-killing definition of s (possible alias)                 *)
END NonExpandedRecords;
```

Definition of Arrays and Array Elements

Arrays can be seen as a whole or as the sum of their elements. Likewise, the definition of an array can be seen as a definition of the whole array or as the definition of all its elements. As outlined in Section 2.2.2, assignments of array elements can be treated as both an assignment and a use of the entire array. This leads to non-killing definitions of the entire array for assignments to array elements. However, if the position of the element within the array is known, the particular array element can be changed and used.

We expand local arrays (including value parameters) of basic types up to a user-configurable size. The nodes of the abstract syntax tree that formerly accessed an array element at a constant position are patched to access the expanded array element. Access to expanded arrays is handled as follows:

- For a killing/non-killing definition of the array (e.g. "a1 := ..."), we insert killing/non-killing definitions of all its expanded elements (i.e. *a1[0]* and *a1[1]*).
- For a use of the array (e.g. "... := a1"), we insert uses of all its expanded elements (i.e. *a1[0]* and *a1[1]*).
- For a killing/non-killing definition of an array element at a constant position (e.g. "a1[0] := ..."), we insert a killing/non-killing definition of the expanded element (i.e. *a1[0]*).
- For a killing/non-killing definition of an array element at an arbitrary position(e.g. "a1[i] := ..."), we insert non-killing definitions of all expanded elements (i.e. *a1[0]* and

a1[1]), since any of them may be changed.

- For a use of an array element at a constant position (e.g. "... := a1[0]"), we insert a use of the expanded array element (i.e. *a1[0]*).
- For a use of an array element at an arbitrary position (e.g. "... := a1[i]"), we insert uses of all its expanded elements (i.e. *a1[0]* and *a1[1]*).

Example 4.14 illustrates the handling of access to expanded arrays.

Example 4.14:

```
PROCEDURE ExpandedArrays;
   VAR a1, a2: ARRAY 2 OF INTEGER; i: INTEGER;
BEGIN
   a1[0] := 0;          (* killing definition of a1[0] *)
   a1[1] := 1;          (* killing definition of a1[1] *)
   a2[0] := 2;          (* killing definition of a2[0] *)
   a2[1] := 3;          (* killing definition of a2[1] *)
   a2[i] := 4;          (* non-killing definition of a2[0] and a2[1] *)
   i := a1[0] + a2[1];  (* first definition reaches a1[0], fourth and fifth definitions reach a2[1] *)
   a1 := a2;            (* the three definitions for a2 are reaching,
                           kills definitions for a1[0] and a1[1] *)
   a2[0] := a1[1]       (* use of definition due to last assignment to a1*)
END ExpandedArrays;
```

We do not expand arrays that are either too big or whose elements are of a structured type. Access to non-expanded arrays is handled as follows:

- For a killing/non-killing definition of the array (e.g. "a1 := ..."), we insert a killing/non-killing definition of the entire array (i.e. *a1*).
- For a use of the array (e.g. "... := a1"), we insert a use of the array (i.e. *a1*).
- For a killing/non-killing definition of an array element (e.g. "a1[0] := ..."), we insert a non-killing definition of the array (i.e. *a1*).
- For a use of an array element (e.g. "... := a1[0]"), we insert a use of the array (i.e. *a1*).

Example 4.15 illustrates the handling of access to non-expanded arrays. The default limit for array expansion is 256 elements.

Example 4.15:

```
PROCEDURE NonExpandedArrays;
   VAR a1, a2: ARRAY 1000 OF INTEGER; i: INTEGER;
BEGIN
   a1[0] := 0;          (* non-killing definition of a1 *)
   a1[1] := 1;          (* non-killing definition of a1 *)
   a2[0] := 2;          (* non-killing definition of a2 *)
   a2[1] := 3;          (* non-killing definition of a2 *)
   a2[i] := 4;          (* non-killing definition of a2 *)
   i := a1[0] + a2[1];  (* use of all previous definitions (including initial definitions) *)
   a1 := a2;            (* use of all definitions of a2 (including initial ones),
                           kills definitions for a1 *)
   a2[0] := a1[1]       (* use of definition due to last assignment *)
END NonExpandedArrays;
```


Handling of Aliases

Two variables *a* and *b* are aliases if they refer to the same memory cell. Zhang and Ryder showed that alias analysis in the presence of procedure variables is NP-hard in most cases [ZhR94]. Exact determination of the sets of variables that are aliases is not possible under the timing restrictions for an interactive tool where the maximum response time must be in the order of seconds. However, less precise information can be computed much faster. The least precise alias information would be that any two variables may be aliases. In the following we will show how we can use type information and information about the place of the declaration of the variable in order to restrict the sets of possible aliases. Finally, we allow feedback from the user to restrict the sets of possible aliases.

When computing the sets of possible aliases for a procedure, we start with the set *S* of accessible objects. These include the local variables and parameters, as well as intermediate and global variables, and additional parameters. When a variable $a \in S$ is defined, all other variables $b \in S$ may also be changed. This means that the value of *b* is either affected by the definition of *a* (if *a* and *b* are aliases) or not (if *a* and *b* are not aliases).

Since Oberon-2 is a statically typed programming language with strong type checking, we can use type information in order to restrict the set of possible aliases of *a*. The type system guarantees that the memory of a variable of type *T* can only be accessed via variables of type *T*. This means that two variables *a* and *b* may only refer to the same memory cell if they have the *same type* [MöWi91]. In Example 4.16, the set *S* of accessible objects of procedure *X* contains the elements *global, i, j,* and *x*. For the assignment to *i*, all other objects of *S* with the same type (i.e., *global* and *j*) might be changed. Since the type of *x* (LONGINT) and the type of *i* (INTEGER) are not the same, *x* and *i* cannot be aliases. For the assignment of *x* there are no possible aliases.

Example 4.16:

```
MODULE Aliases;

VAR global: INTEGER;

PROCEDURE X (VAR i: INTEGER);
   VAR j: INTEGER; x: LONGINT;
BEGIN
   i := 0;       (* global and j may be changed, too *)
   x := i + j
END X;

END Aliases.
```

However, the sets of possible aliases is still too big, since *i* and *j* may never be aliases. We can deduce that if we consider where the two variables are declared. Therefore, we first recapitulate the different possibilities for declaring variables:

- Global variables are declared at the module level. They reside within the block of global data of a module. They are also called static data, since they are allocated when the module is loaded and they stay in memory until the module is unloaded.

Each global variable is allocated at a different address, no two global variables may refer to the same memory cell. They are accessible from anywhere within the declaring module. Additionally, they may be exported by the declaring module and imported into other modules. This export can be either read-only or read/write (thereby granting the importing module full access to the object).

- Local variables are declared within a procedure. They are allocated for each activation of a procedure (in most implementations on the stack). Each local variable is allocated at a different address, no two local variables may refer to the same memory cell. They are also called automatic data, since they are allocated when the procedure is called and their memory is automatically reclaimed when the procedure returns. They are only accessible within the declaring procedure and procedures that are nested within the declaring procedure.
- Intermediate variables are local variables of a procedure *P* that are accessed from within a procedure *Q* that is nested in *P*. When regarding these variables from procedure *P*, they are ordinary local variables, when regarding them from procedure *Q*, they are intermediate variables, since they are neither local to *Q*, nor global, but intermediate.
- Objects and arrays may be allocated on the heap. They are referenced by and accessible via pointer variables. Two pointers may reference the same object. Heap data is also called dynamic data, since it is allocated on demand (by a NEW statement) and its memory is automatically reclaimed by the garbage collector when it is no longer referenced.

There are two kinds of parameters in Oberon-2:

- Value parameters can be considered as local variables where the values of the expressions at the call sites are used as initial values of the formal parameters. Memory is automatically allocated and reclaimed as for local variables.
- Reference parameters can be considered as additional names for the actual parameters. No new memory is allocated for reference parameters. Reference parameters are the main source of aliases in Oberon-2.

With the information about the place of the declaration of a variable, one can restrict the sets of possible aliases. Table 4.10 contains one row and one column for local variables (including local value parameters), global read-only variables of imported modules, intermediate variables (including intermediate value parameters), global variables of the module under consideration, and global variables of imported modules that have been exported without access restrictions, and reference parameters. In each cell of the table, "no" indicates that two objects o1 (row) and o2 (column) may not be aliases, whereas "yes" indicates that the two objects may be aliases. The table is symmetric, i.e. *Cell(x1, y1) = Cell(y1, x1)*.

o1 \ o2	(1)	(2)	(3)	(4)	(5)	(6)
(1) local var	no	no	no	no	no	no
(2) global var (other mod.), read-only	no	no	no	no	no	no
(3) intermediate var	no	no	no	no	no	yes
(4) global var (this mod.)	no	no	no	no	no	yes
(5) global var (other mod.), r/w	no	no	no	no	no	yes
(6) reference par	no	no	yes	yes	yes	yes

Table 4.10 - Possible aliases

Local variables, intermediate and global variables may never be aliases since they are allocated at different addresses. Reference parameters are aliases of their actual parameters. At the call site, intermediate variables with smaller nesting level ("yes" in last column of row 3 in Table 4.10, see Example 4.17), global variables without access restrictions (rows 4 and 5 in Table 4.10), and other reference parameters with smaller nesting level (row 6 in Table 4.10) may be used as actual parameters.

Example 4.17:

```
MODULE Aliases;

PROCEDURE X;
   VAR i: INTEGER;

   PROCEDURE Local (VAR j: INTEGER);  (* The reference parameter j is an alias of the
                                         intermediate variable i.                    *)
      VAR x: LONGINT;
   BEGIN
      j := 0;                         (* i is changed, too                           *)
      x := i + j                      (* both i and j, access the same memory cell   *)
   END Local;

BEGIN
   Local(i)
END X;

END Aliases.
```

With these restrictions, the set of possible aliases at the assignment of *i* in procedure *X* of Example 4.16 is restricted to {*global*}. For a call of *X* with *global* as the actual parameter, *i* and *global* would actually be aliases. Therefore, this is the most precise set of possible aliases, as long as we do not regard the actual parameters at call sites. For incomplete programs, such as frameworks, we cannot further restrict the sets of possible aliases. If we analyze complete programs ("closed-world assumption"), then we could further reduce the sets of possible aliases.

Structured data types raise another problem: the actual parameter at a call site may be a part of a structured variable, e.g. a field of a record or an element of an array. The variable may be allocated statically, automatically, or dynamically. So it is not enough to test whether two variables *a* and *b* have the same type when computing the possible aliases for the definition of *a*. We must additionally test whether variable *a* can be contained in *b* or

whether *b* can be contained in *a*. *a* can be contained in *b* in the following cases:

- if *b* is a record of which *a* might be a field,
- if *b* is an array of which *a* might be an element,
- if *b* is a pointer that is dereferenced and *a* might be a field of the referenced record or an element of the referenced array, or
- if *b* is a record and its type is an extension of the type of *a*.

Example 4.18 shows possible calls of procedure *X*, where the variables that might contain each other are actual aliases.

Example 4.18:

```
MODULE Aliases;

TYPE
   T = RECORD i: INTEGER END ;
   P = POINTER TO T;
   T1 = RECORD (T) j: INTEGER END ;
   P1 = POINTER TO T1;

VAR
   t: T; t1: T1;
   p: P; p1: P1;

PROCEDURE X (VAR i: INTEGER);
BEGIN
   t.i := 0;                    (* changes i for the first call of X in the
                                   module body                          *)
   t1.j := 0;                   (* changes i for the second call of X   *)
   t := t1;                     (* changes i for the first call of X    *)
   p^.i := 0;                   (* changes i for the third call of X    *)
   p1^.j := 0;                  (* changes i for the fourth call of X   *)
   i := 0;                      (* changes t.i for the first call of X,
                                   changes t1.j for the second call of X,
                                   changes p^.i for the third call of X,
                                   changes p1^.j for the fourth call of X *)
END X;

BEGIN
   X(t.i);
   X(t1.j);
   X(p^.i);
   X(p1^.j)
END Aliases.
```

When we have narrowed the sets of possible aliases of a variable *a*, we insert non-killing definitions for all possible aliases *b* at the node defining *a*. These non-killing definitions lead to additional reaching definitions at places, where *b* is used.

The user interface of the Oberon Slicing Tool visualizes the sets of possible aliases. The user can disable some or all of the possible aliases. He can then initiate the re-computation of data flow information where only the enabled aliases lead to non-killing definitions.

4.5.2 Computation of Reaching Definitions

In this section we describe our algorithm for the computation of the definition sets and of the gen and kill sets of defining nodes. Then we show how these are combined to the gen and kill sets of the statement sequences, which are then used to compute the reaching definitions.

Computation of the Definition Sets of Variables and of the Gen and Kill Sets

We have to modify the procedure for the computation of the definition sets and of the gen and kill sets outlined in Section 2.3.2 for several reasons:

- We use a finer-grained intermediate representation than Aho et al. [ASU86].
- We allow multiple definitions at one node.
- We allow killing and non-killing definitions at one node.

In order to account for these requirements, we have to use a triplet in order to describe a definition. A definition consists of the defining AST node, the defined object, and the kind of the definition (killing or non-killing). Each definition must be associated with a number. Therefore we insert each definition into a hash table and use the index for this definition within the hash table to represent the definition in the bit sets *gen* and *kill*. If the definitions were simply numbered consecutively by inserting them into an array or a list, looking up a definition could necessitate a linear scan of all definitions.

ComputeDefinitionSets computes the definition set for each variable that is defined by the procedure. For each object the killing definitions are collected in a bit set called "must assigns" and all (killing and non-killing) definitions are collected in a bit set called "may assigns" (the Definition Set). The loop of *ComputeDefinitionSets* iterates over all definitions of the hash table. Initially, the set of killing definitions (must assigns) and the set of all definitions (may assigns) are empty. The indices of the definitions within the hash table are inserted into the set "may assigns" and into the set "must assigns" (if the definition is a killing one).

```
PROCEDURE ComputeDefinitionSets (varDefs: Definitions);
   VAR i: INTEGER; def: HashTableEntry; obj: Object; e: Definition;
BEGIN
   FOR i := 0 TO size of hash table of definitions of the current procedure DO
      def = definition i of the hash table
      obj := object of definition def
      IF varDefs does not yet contain an entry e for obj THEN
         insert entry e with empty sets "may assigns" and "must assigns" for obj in varDefs
      END ;
      include i into the set of "may assigns" of e
      IF def is a killing definition THEN
         include i into the set of "must assigns" of e
      END
   END
END ComputeDefinitionSets;
```


Fig. 4.17 shows how the definitions sets are computed for parameter *i* of procedure *X* in Example 4.19. First, an entry for parameter *i* is inserted into the variable definitions with empty sets "must assigns" and "may assigns". Whenever a definition is encountered, its index is included into the "may assigns" (indices 1, 3, 4, and 5). Whenever the definition is a killing one, its index is included into the "must assigns" (indices 1, 3, and 4).

Example 4.19:

```
PROCEDURE X (VAR i, j: INTEGER);
   (* Assignments for parameter passing:
      Nfpar/inPar: i                (* 5 *)
      Nfpar/inPar: j                (* 6 *)
   *)
   VAR k: LONGINT;
   (* Assignments for initialization of local variables:
      Nenter: k                     (* 7 *)
   *)
BEGIN
   i := 1;                          (* 1 *)
   j := 2;                          (* 2 *)
   i := i * 2;                      (* 3 *)
   k := i + j                       (* 4 *)
END X;
```

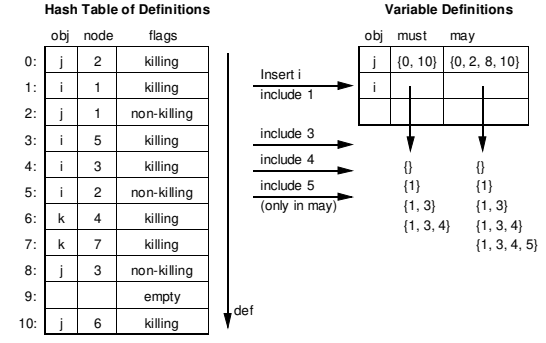

Fig. 4.17 - Computation of the definition sets of parameter *i*

ComputeGenKillSets computes for each defining node the gen and kill sets. The loop of *ComputeGenKillSets* iterates over all definitions of the hash table. Initially, the gen and kill sets are empty for each node. The indices of the definitions are included into the node's gen set. The kill set of the node is the union of the kill sets for each killing definition, where the kill set for a killing definition *d* is the definition set (may assigns) of the variable defined at *d* without the index of *d*.


```
PROCEDURE ComputeGenKillSets (varDefs: Definitions);
   VAR i: INTEGER; def: Definition; node: Node; e: Definition;
BEGIN
   FOR i := 0 TO size of hash table of definitions of the current procedure DO
      def = definition i of the hash table
      node := node of the definition def
      IF gen and kill sets of node have not yet been computed THEN
         allocate empty gen and kill sets for node
      END ;
      include i into the gen set of node
      IF def is a killing definition THEN
         e := entry of varDefs for object defined at def
         node.kill := node.kill ∪ ("may assigns" of e - i)
      END
   END
END ComputeGenKillSets;
```

Fig. 4.18 shows how the gen and kill sets are computed for the node 3. First, empty *gen/kill* sets are allocated for node 3. Whenever a definition at node 3 is encountered, its index is included in the gen set (indices 4 and 8). Whenever the definition is a killing one, the kill set is updated by the definition set of the defined object excluding the index of the definition.

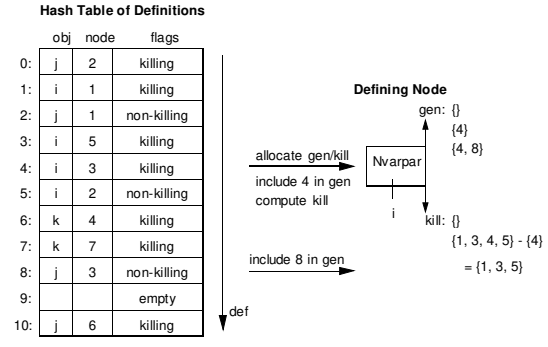

Fig. 4.18 - Computation of *gen* and *kill* for node 3

Data Flow Equations of Iterative Statements

In Section 2.2.3, we gave the data flow equations for iterative statements. For Oberon-2, the equations have to be adapted to model the WHILE, REPEAT and LOOP statements. Additionally, side-effects of function calls within expressions must be handled properly. But let us first examine the data flow equations once more:

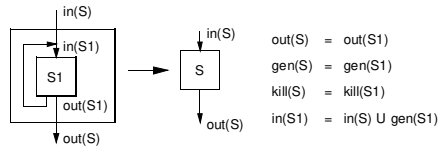

Fig. 4.19 - Data flow equations for an iterative statement

We have argued that *gen(S)* and *kill(S)* for the compound statement are the same as the gen and kill sets of the nested statement sequence. The following argument shall serve as an informal proof:

Let us suppose that the iterative statement *S* can be unfolded into a sequence of statements:

S = S1 S2 ... Sn with *S1 = S2 = Sn*

gen(S) can then be computed by applying the rule for statement sequences:

gen(S1;S2) = gen(S2) ∪ (gen(S1) - kill(S2)) =
= gen(S1) ∪ (gen(S1) - kill(S1)) = since S1 = S2
= gen(S1) since the second term is a subset of the first

Since *gen(S1;S2) = gen(S1)*, we see that executing the same statement several times does not generate new definitions.

We have further argued that *in(S1) = in(S) ∪ gen(S1)*. The following argument shall serve as an informal proof:

In general, *out(S)* can be computed as:
out(S) = gen(S) ∪ (in(S) - kill(S))

For an iterative statement, all definitions that leave the end of the nested statement sequence *S1* are again definitions that reach the beginning of the nested statement sequence *S1*. This (seemingly recursive) problem can only be solved by iteration until no new definitions are generated that leave the compound statement *S*.

For the Fig. 4.19, *out(S)* can be computed by the following equation:

Equation 1: out(S1) = gen(S1) ∪ (in(S1) - kill(S1))

Whereas *in(S1)* can obviously (see Fig. 4.19) be computed as:

Equation 2: in(S1) = in(S) ∪ out(S1)

After substituting *in(S1)* by *I*, *out(S1)* by *O*, *in(S)* by *J*, *gen(S1)* by *G*, and *kill(S1)* by *K* in Equations 1 and 2, we get the following two recurrence equations:

I = f(O, J) := J ∪ O
O = f(I, G, K) := G ∪ (I - K)

I and *O* can be seen as functions, whereas *J*, *G* and *K* are constants in these two equations. A solution can be found by starting from a conservative assumption ($O_0 = \{\}$) and then substituting one equation by the result of the other:

$I_1 = J \cup O_0 = J$

Then I_1 can be used to get a better approximation for *O*:

$O_1 = G \cup (I_1 - K) = G \cup (J - K)$

Again, we can compute *I* by using the better approximation for *O*:

$I_2 = J \cup O_1 = J \cup G \cup (J - K)$
$= J \cup G$ since last term *J - K* is a subset of *J*

The next approximation for *O* is:

$O_2 = G \cup (I_2 - K) = G \cup ((J \cup G) - K)$
$= G \cup ((J - K) \cup (G - K))$ since $(A \cup B) - C = (A - C) \cup (B - C)$
$= G \cup (J - K) \cup (G - K)$
$= G \cup (J - K)$ since the term *G - K* is a subset of *G*

But since $O_1 = O_2$, further iteration will not produce any new results and we see that the solution for the two recurrence equations can be written (after backwards substitution) as:

in(S1) = in(S) ∪ gen(S1)
out(S1) = gen(S1) ∪ (in(S) - kill(S1))

The first of these two equations has already been introduced above, the second one can be used to compute *out(S)*.

Combination of the Gen and Kill Sets and the Computation of the Reaching Definitions

Fig. 4.20 shows how the *out* set can be computed for procedure *X* of Example 4.19: The definitions are inserted into the hash table of definitions, the definition sets are computed for each variable and the *gen/kill* sets are computed for each defining node. The *in* set of the procedure contains the initial definitions of the parameters and the local variables. The *out* set is then computed by applying the equation

out = gen ∪ (in - kill)

to every node. The *out* set of the one node is the *in* set for the next node. The *out* set of the last node is the *out* set of procedure *X*.

Hash Table of Definitions

	obj	node	flags
0:	j	2	killing
1:	i	1	killing
2:	j	1	non-killing
3:	i	5	killing
4:	i	3	killing
5:	i	2	non-killing
6:	k	4	killing
7:	k	7	killing
8:	j	3	non-killing
9:			empty
10:	j	6	killing

Definition Sets

	obj	must	may
	j	{0, 10}	{0, 2, 8, 10}
	i	{1, 3, 4}	{1, 3, 4, 5}
	k	{6, 7}	{6, 7}

	gen	kill
1:	{1, 2}	{3, 4, 5}
2:	{0, 5}	{2, 8, 10}
3:	{4, 8}	{1, 3, 5}
4:	{6}	{7}
5:	{3}	{1, 4, 5}
6:	{10}	{0, 2, 8}
7:	{7}	{6}

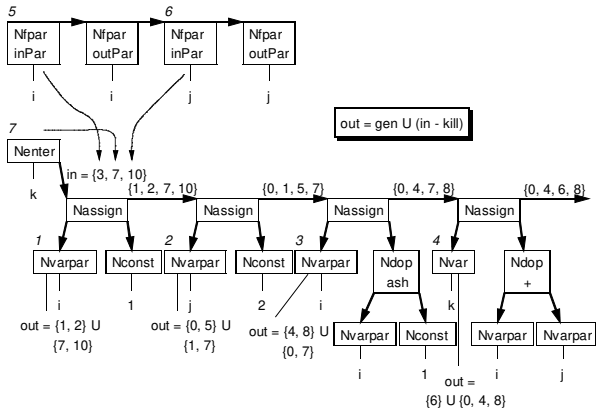

Fig. 4.20 - Computation of the *out* set of procedure *X* of Example 4.19

At each variable usage node *U*, links are inserted to the reaching definitions. Therefore the set *in(U)* is examined. For each bit included in *in(U)*, the hash table of definitions contains an entry *E*. If *E* defines the object that is used at *U*, a link is inserted from *U* to the defining node of *E*. Fig. 4.21 shows the links from the usage nodes to all reaching definitions.

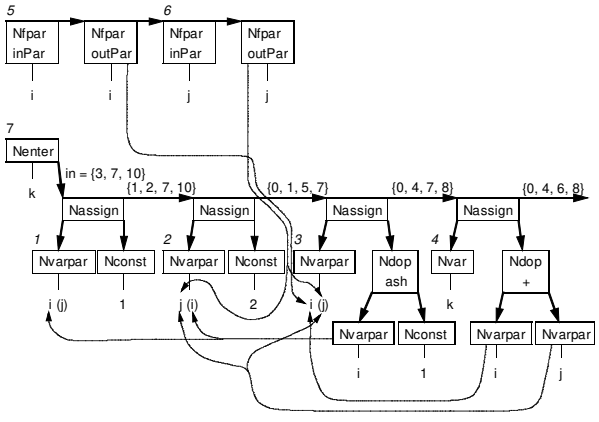

Fig. 4.21 - Reaching definitions of procedure *X* of Example 4.19

In the following we show for each language construct how the gen and kill sets can be computed by a proper traversal of the syntax tree and how the out sets can be computed by another traversal.

Short-Circuit Evaluation of Boolean Expressions

The language report of Oberon-2 defines that evaluation of Boolean expressions is stopped when the result is known (*short-circuit evaluation*, see Table 4.11).

Boolean Expression	Equivalent
p & *q*	if *p* then *q*, else FALSE
p OR *q*	if *p* then TRUE, else *q*

Table 4.11 - Evaluation of Boolean Expressions

In other words, if the first operand of the expression p & q evaluates to FALSE, the second operand is not evaluated any more, since the result of the whole expression can only be FALSE. Therefore, definitions in the left sub-tree of a Boolean expression are only killed by definitions in the right sub-tree if the right sub-tree is at all evaluated. Likewise, the second operand of the expression p OR q is only evaluated if the first evaluated to FALSE. Example 4.20 shows that exact modeling of short-circuit evaluations is necessary to compute precise reaching definitions: The expression of the IF contains two function calls, which both modify the parameter. Since the formal parameter *i* of function *ChangePar* is assigned on all

paths, the assignment of the corresponding actual parameter is a killing one. When the THEN branch of the IF is executed, both parts of the expression must have evaluated to TRUE, therefore the definitions of *i* and *j* before the IF are killed by the evaluation of the expression. When the ELSE branch of the IF is executed, either the left part of the logical AND failed (in which case the right part is not evaluated at all) or the right part failed (in which case both parts are evaluated). In the first case, the initial definition of *j* is not killed.

Example 4.20:

```
        MODULE ShortCircuitEvaluation;

        PROCEDURE ChangePar (VAR i: INTEGER): INTEGER;  (* assigns to the parameter,
                                                           but does not use it *)
        BEGIN
          i := 0; RETURN 0
        END ChangePar;

        PROCEDURE Do;
          VAR i, j: INTEGER;
        BEGIN
(* 1 *)    i := 0;
(* 2 *)    j := 2;
(* 3, 4 *)  IF (ChangePar(i) < 0) & (ChangePar(j) < 0) THEN  (* i and j defined by the function
                                                               calls                    *)
(* 5 *)       i := i;                                       (* only def 3 is reaching   *)
(* 6 *)       j := j                                        (* only def 4 is reaching   *)
           ELSE                                             (* i defined by the function call,
                                                               j may be defined by the
                                                               function call            *)
(* 7 *)       i := i;                                       (* only def 3 is reaching   *)
(* 8 *)       j := j                                        (* def 2 may be reaching    *)
           END ;
(* 9 *)    i := i;                                          (* only defs 5 and 7 are reaching*)
(* 10 *)   j := j                                           (* only defs 6 and 8 are reaching*)
        END Do;

        END ShortCircuitEvaluation.
```

Compound Boolean expressions are handled by computing a pair of *gen/kill* sets for the case that the expression evaluates to TRUE (*genT/killT*) and one for the case that it evaluates to FALSE (*genF/killF*). If the result of a Boolean expression is merely assigned to a variable, the two *gen/kill* sets are combined conservatively. They are only treated separately if they guide the flow of control (e.g. in the expression of an IF or a WHILE).

A conservative combination of the sets *gen1/kill1* and *gen2/kill2* (two merging arrows in the figures below) is implemented as:

gen = gen1 ∪ gen2 kill = kill1 ∩ kill2

A sequence of the sets *gen1/kill1* and *gen2/kill2* (e.g. "genT/killT → genF/killF" in Fig 4.22) is implemented as:

gen = gen2 ∪ (gen1 - kill2) kill = kill2 ∪ (kill1 - gen2)

For a logical AND, the *gen/kill* sets for the case that the expression evaluates to TRUE (*genT/killT*) is computed as the sequence of the *genT/killT* sets of the left-hand side and the *genT/killT* sets of the right-hand side. The *gen/kill* sets for the case that the expression evaluates to FALSE (*genF/killF*) is the conservative combination of the *genF/killF* sets of the left-hand side and the sequence of the *genT/killT* sets of the left-hand side with the *genF/killF* sets of the right-hand side as shown in Fig. 4.22.

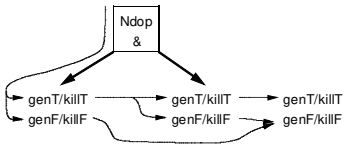

Fig. 4.22 - Computation of *gen* and *kill* for a logical AND

For a logical OR, the *gen/kill* sets are computed similarly (with reversed Boolean values) as shown in Fig. 4.23.

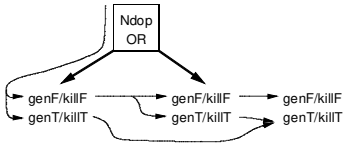

Fig. 4.23 - Computation of *gen* and *kill* for a logical OR

For a logical negation, the *gen/kill* sets for the TRUE and FALSE case are simply interchanged as shown in Fig. 4.24.

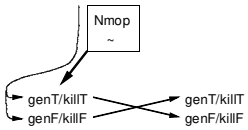

Fig. 4.24 - Computation of *gen* and *kill* for a logical NOT

The *out* set of a logical AND is again split in two parts (see Fig. 4.25): *outT* for the case that the expression evaluates to TRUE and *outF* for the case that the expression evaluates to FALSE. For the computation of *outT,* we first feed *in* into the left sub-tree and then the TRUE-output of the left sub-tree (*outLT*) into the right sub-tree. The TRUE-output of the

right sub-tree is then *outT* of the entire Boolean expression. *outF* is the union of the FALSE-outputs of both sub-trees. The *out* set of a logical OR is computed similarly (see Fig. 4.26). For a logical negation, the sets *outT* and *outF* are simply interchanged (see Fig. 4.27). When a usage node is visited during the traversal of the expression trees, links are inserted to all reaching definitions.

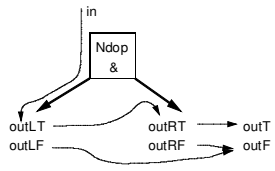

Fig. 4.25 - Computation of *out* for a logical AND

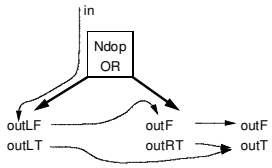

Fig. 4.26 - Computation of *out* for a logical OR

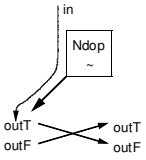

Fig. 4.27 - Computation of *out* for a logical NOT

Assignments

The order of the evaluation of the left-hand side and the right-hand side of an assignment statement is not defined in Oberon. Therefore, programs must not rely on some particular evaluation order. Depending on the implementation of the compiler, Example 4.21 may yield different results.

Example 4.21:


```
VAR arr: ARRAY 2 OF INTEGER; i: INTEGER;

PROCEDURE SideEffect (VAR i: INTEGER): INTEGER;
BEGIN i := 1; RETURN 0
END SideEffect;

BEGIN
  arr[0] := 1; arr[1] := 1; i := 0;
  arr[i] := SideEffect(i)
  (* if lhs is evaluated first: arr[0] = 0, if rhs is evaluated first: arr[1] = 0 *)
END
```

We compute data flow information under the assumption, that the right-hand side of an assignment statement is evaluated first. Although this assumption is not strictly conservative, we do not think of it as a source of major inaccuracies. Fig. 4.28 shows that the right-hand side may be a Boolean expression. Then sets *genT/killT* and *genF/killF* are combined conservatively (indicated by the merge of the two arrows into one arrow). The result is combined sequentially with the *gen/kill* sets of the left-hand side which must not be a Boolean expression.

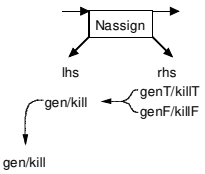

Fig. 4.28 - Computation of *gen* and *kill* for an assignment statement

Fig. 4.29 shows how the *in* set of the assignment statement is "pushed through" the right sub-tree, giving the *out* set of the right sub-tree. After combining the two possible results *outT*/*outF* conservatively (indicated by the merge of the two arrows into one arrow), the *out* set is used as the *in* set of the left sub-tree. Pushing it through the left sub-tree yields the *out* set of the entire assignment statement.

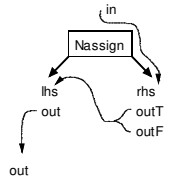

Fig. 4.29 - Computation of *out* for an assignment statement

Calls

The order of evaluation of the parameters is not defined in Oberon-2. The language report only states that "the component selectors are evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure" (Section 9.2 Procedure calls).
We assume evaluation of the parameters from left to right and give a warning if the order of the evaluation of the parameters is significant (i.e. would lead to different gen and kill sets). Evaluation from left to right yields a reasonably small out set where new definitions of a variable *x* really kill preceding definitions of *x*. If we combined the out sets of all parameters via a union, the out set of the entire call would almost always contain the entire in set, since a reaching definition must be killed in each branch in order not to leave the statement as part of the out set.

IF

The data flow equations for conditional statements have to be adapted to properly handle side-effects due to function calls within expressions and short-circuit evaluation of Boolean expressions. The following rule describes the possible paths of an IF statement:

```
IF expr1 THEN stat1
ELSIF expr2 THEN stat2
...
ELSE statElse
END
```

$$\begin{aligned}
Paths_{\text{IF with ELSE}} &= expr1_{TRUE}\ stat1 \mid \\
&\quad expr1_{FALSE}\ expr2_{TRUE}\ stat2 \mid \\
&\quad \ldots \\
&\quad expr1_{FALSE}\ expr2_{FALSE}\ \ldots\ exprN_{FALSE}\ statElse. \\
Paths_{\text{IF without ELSE}} &= expr1_{TRUE}\ stat1 \mid \\
&\quad expr1_{FALSE}\ expr2_{TRUE}\ stat2 \mid \\
&\quad \ldots \\
&\quad expr1_{FALSE}\ expr2_{FALSE}\ \ldots\ exprN_{FALSE}.
\end{aligned}$$

These two rules can be combined to one since a non-existing ELSE branch is semantically equivalent to an empty ELSE branch:

$$\begin{aligned}
Paths_{IF} &= expr1_{TRUE}\ stat1 \mid \\
&\quad expr1_{FALSE}\ expr2_{TRUE}\ stat2 \mid \\
&\quad \ldots \\
&\quad expr1_{FALSE}\ expr2_{FALSE}\ \ldots\ exprN_{FALSE}\ [statElse].
\end{aligned}$$

Fig. 4.30 shows the computation of the *gen/kill* sets for an IF statement along these paths, whereas Fig. 4.31 shows the computation of the *out* set.

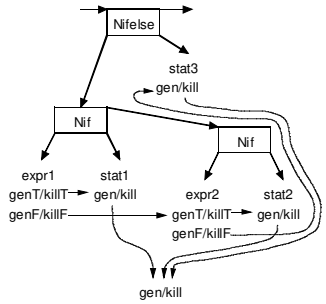

Fig. 4.30 - Computation of *gen* and *kill* for an IF statement

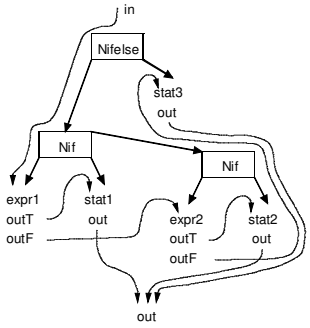

Fig. 4.31 - Computation of *out* for an IF statement

CASE

Exactly one branch of the CASE is executed depending on the result of the expression of the CASE. If none of the constant expressions guarding the respective branches matches, the ELSE branch is executed. If there is no ELSE branch, the program is aborted. Therefore, it would not be allowed to treat an empty ELSE branch in the same way as a non-existing ELSE branch. The *gen/kill* set of the CASE statement consists only of the *gen/kill* sets of the existing branches and the *gen/kill* set of the ELSE branch (only if it exists). Likewise, the *out* set consists of the *out* sets of the existing branches and the *out* set of the ELSE branch (only if it exists). Fig. 4.32 shows the computation of the *gen/kill* sets for a CASE statement, whereas Fig. 4.33 shows the computation of the *out* set.

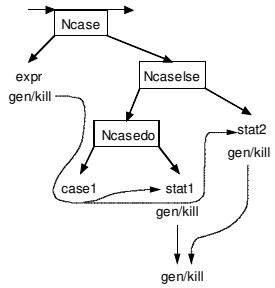

Fig. 4.32 - Computation of *gen* and *kill* for a CASE statement

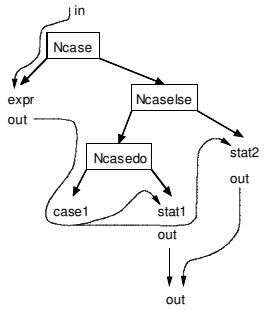

Fig. 4.33 - Computation of *out* for a CASE statement

WITH

If none of the type tests evaluates to TRUE and there is no ELSE branch in the source text of the program, the program is aborted. Therefore, it would not be allowed to treat an empty ELSE branch in the same way as a non-existing ELSE branch. The *gen/kill* set of the WITH statement consists only of the *gen/kill* sets of the existing branches and the *gen/kill* set of the ELSE branch (only if it exists). Fig. 4.34 shows the computation of the *gen/kill* sets for a WITH statement.

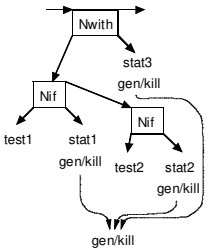

Fig. 4.34 - Computation of *gen* and *kill* for a WITH statement

Likewise, the *out* set consists of the *out* sets of the existing branches and the *out* set of the ELSE branch (only if it exists). Fig. 4.35 shows the computation of the *out* set for a WITH statement. Note that the *in* set is pushed through the tests although they cannot generate new definitions. This is necessary to insert links from the variable usage nodes to all reaching definitions.

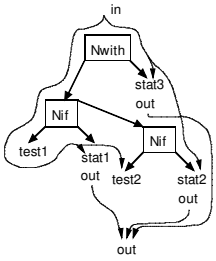

Fig. 4.35 - Computation of *out* for a WITH statement

WHILE

The *gen/kill* set of the nested statement sequence must be computed since it is needed for the computation of *out*. The *gen/kill* set of the entire WHILE loop has to combine the *gen/kill* sets of the following three cases (see Fig. 4.36):

- The loop is not entered at all because the guarding expression evaluates to FALSE.
- The loop is entered and executed once.
- The loop is entered and executed several times.

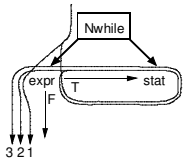

Fig. 4.36 - Iterations of a WHILE loop

One could give the following rule for possible paths of a WHILE loop:

$Paths_{WHILE} = \{expr_{TRUE}\ stat\}\ expr_{FALSE}$.

Since the *gen/kill* set for an iteration statement is the same as the *gen/kill* set of the iterated statement sequences, we can deduce that the *gen/kill* set of a WHILE loop only has to combine the first two cases (the last case does not generate new information), which are shown in Fig. 4.37.

$$gen/kill_{WHILE} = \{\ genT/killT_{expr}\ gen/kill_{stat}\ \}\ genF/killF_{expr} =$$
$$= [\ genT/killT_{expr}\ gen/kill_{stat}\ ]\ genF/killF_{expr} =$$
$$= genF/killF_{expr} \mid genT/killT_{expr}\ gen/kill_{stat}\ genF/killF_{expr}$$

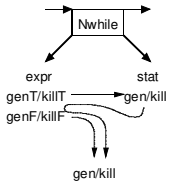

Fig. 4.37 - Computation of *gen* and *kill* for a WHILE statement

As we showed during the explanation of the data flow equations of iterative statements, the *in* set for the nested statement sequence *S1* is computed as *in(S) ∪ gen(S1)*. In order to consider side-effects of function calls within the expression of the WHILE, *S1* is the part of the loop that is executed for each iteration (i.e., the part of the rule for the *gen/kill* set of the WHILE statement that is initially enclosed in curly braces). *gen(S1)* and *kill(S1)* are the sets of definitions that are generated/killed by one iteration of the loop, i.e.

$$gen/kill(S1) = genT/killT_{expr}\ gen/kill_{stat}$$

In the following informal proofs of the computation of the *out* sets, we will use the notation

$$x => S => y$$

which means that the program fragment *S* has the *in* set *x* and produces the *out* set *y*. In this way program fragments can be concatenated where the output of the first fragment is the input of the next. Boolean expressions return two *out* sets, one for the TRUE branch and one for the FALSE branch. Which of the both is actually used is indicated by the subscript. For annotation purposes, we insert names into the chain of functions in order to give a name to the temporary result that is being passed from one function to the next.
The *out* set of the WHILE loop also has to combine the *out* sets of the three cases given above.

1) $in(WHILE) => expr_{FALSE} => out_1(WHILE)$
2) $in(WHILE) => expr_{TRUE}\ stat => out' => expr_{FALSE} => out_2(WHILE)$
   out' could be computed as "*gen(S1) ∪ (in(WHILE) - kill(S1))*", but instead we compute it by "pushing" *in(WHILE)* "through" *S1* (i.e. through the AST of the expression and the statement sequence). During this process, data dependence edges are inserted from nodes that represent variable usages to all nodes that represent reaching definitions of these variables.
3) $in(WHILE) => expr_{TRUE}\ stat => out' => \{\ expr_{TRUE}\ stat\ \} => out'' => expr_{FALSE} => out_3(WHILE)$
   out'' could be computed as follows:
   $out'' = gen(S1) \cup (out' - kill(S1)) =$
   $= gen(S1) \cup ((gen(S1) \cup (in(WHILE) - kill(S1))) - kill(S1)) =$
   $= gen(S1) \cup (gen(S1) - kill(S1)) \cup (in(WHILE) - kill(S1) - kill(S1)) =$
   $= gen(S1) \cup (in(WHILE) - kill(S1)) = out'$

   Since *out'' = out'*, we could be tempted to ignore the third case (as we did when we computed the *gen/kill* set). Although no new definitions are generated and the out set remains the same, we have to execute at least one iteration because otherwise the definitions contained in *out'* would never be used as input to "$expr_{TRUE}$ *stat*". Fortunately, one iteration is enough, so we end up with the following formula for the third case:

   $in(WHILE) => expr_{TRUE}\ stat => out' => expr_{TRUE}\ stat => expr_{FALSE} => out_3(WHILE)$

   This would require two traversals of *S1*, where the in set is once *in(WHILE)* and once *out'*. Fortunately, these two traversals can be combined by using *in(WHILE) ∪ gen(S1)* as the in set of the traversal of *S1*, since

   $in(WHILE) \cup out' = in(WHILE) \cup gen(S1) \cup (in(WHILE) - kill(S1)) =$
   $= in(WHILE) \cup gen(S1)$

   The combination of the two traversal is legal since it produces the same out set:

   Pushing a bit set *in1* through a statement sequence *S* produces the out set
   out1 = gen(S) ∪ (in1 - kill(S)).

Pushing a bit set *in2* through a statement sequence *S* produces the out set
out2 = gen(S) ∪ (in2 - kill(S)).
Pushing the bit set *in1 ∪ in2* through *S* produces the *out* set
$out = gen(S) \cup ((in1 \cup in2) - kill(S)) =$
$= gen(S) \cup (in1 - kill(S)) \cup (in2 - kill(S)) =$
$= gen(S) \cup (in1 - kill(S)) \cup gen(S) \cup (in2 - kill(S)) =$
$= out1 \cup out2$

The in set for the traversal of $expr_{FALSE}$ would have to be *in(WHILE)* for the first case and *out'* for the second and third cases. Again, these two traversals can be combined into one traversal of $expr_{FALSE}$ with *in(WHILE) ∪ gen(S1)* as the in set. So we end up with the following rule for the computation of *out(WHILE)*:

$in(WHILE) \cup gen(S1) => expr_{TRUE}\ stat => out'$
$in(WHILE) \cup gen(S1) => expr_{FALSE} => out(WHILE)$

The first rule is needed to insert all necessary data dependences within the expression and the nested statement sequence. The second rule is used to compute the actual out set of the WHILE as shown in Fig. 4.38.

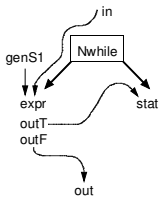

Fig. 4.38 - Computation of *out* for a WHILE statement

REPEAT

As for the WHILE loop, the *gen/kill* set of the nested statement sequence must be computed since it is needed for the computation of *out*. The *gen/kill* set of the entire REPEAT loop has to combine the *gen/kill* sets of the following two cases (see Fig. 4.39):

- The loop is entered and executed once.
- The loop is entered and executed several times.

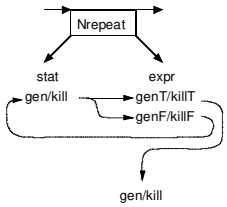

Fig. 4.39 - Computation of *gen* and *kill* for a REPEAT statement

The rule for possible paths of a REPEAT loop is:

$Paths_{REPEAT} = \{stat\ expr_{FALSE}\}\ stat\ expr_{TRUE}$.

Following the same argument as for the WHILE loop, we can deduce that the *gen/kill* set of a REPEAT loop only has to combine the following two cases:

$$gen/kill_{REPEAT} = \{\ gen/kill_{stat}\ genF/killF_{expr}\ \}\ gen/kill_{stat}\ genT/killT_{expr} =$$
$$= [\ gen/kill_{stat}\ genF/killF_{expr}\ ]\ gen/kill_{stat}\ genT/killT_{expr} =$$
$$= gen/kill_{stat}\ genT/killT_{expr} \mid gen/kill_{stat}\ genF/killF_{expr}\ gen/kill_{stat}\ genT/killT_{expr}$$

gen(S1) and *kill(S1)* are the sets of definitions that are generated/killed by one iteration of the loop, i.e.

$$gen/kill(S1) = gen/kill_{stat}\ genF/killF_{expr}$$

The *out* set of the REPEAT loop also has to combine the *out* sets of the two cases given above:

1) $in(REPEAT) => stat\ expr_{TRUE} => out_1(REPEAT)$
2) $in(REPEAT) => stat\ expr_{FALSE} => out' => stat\ expr_{TRUE} => out_2(REPEAT)$
   out' could be computed as "*gen(S1) ∪ (in(REPEAT) - kill(S1))*", but we rather compute it by "pushing" *in(REPEAT)* "through" the AST of the expression and the statement sequence. During this process, data dependence edges are inserted from nodes that represent variable usages to all nodes that represent reaching definitions of these variables.

For the computation of *out(REPEAT)*, we can simply feed "*in(REPEAT) ∪ gen(S1)*" into "*stat* $expr_{TRUE}$", since

$in(REPEAT) \cup out' = in(REPEAT) \cup gen(S1) \cup (in(REPEAT) - kill(S1)) =$
$= in(REPEAT) \cup gen(S1)$

Thereby, data dependences will also be inserted. Fig. 4.40 shows the computation of the *out* set for a REPEAT statement

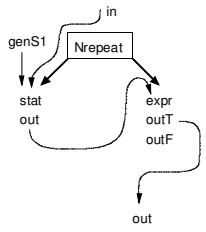

Fig. 4.40 - Computation of *out* for a REPEAT statement

LOOP

We compute two pairs of *gen/kill* sets for the LOOP: one represents the definitions generated/killed by an iteration of the loop, the other one represents the definitions generated/killed by the execution of the entire LOOP. The first is needed for the computation of the *out* set of the LOOP where it is fed as additional input due to one iteration into the nested statement sequence, the latter is a conservative combination of the *gen/kill* sets that reach the directly nested EXITs. The *out* set of the LOOP conservatively combines the *out* sets of all directly nested EXITs.

EXIT

An EXIT statement kills all definitions, i.e. no definitions reach the statements following the EXIT. But the *gen/kill* sets reaching the EXITs are combined to the *gen/kill* set of the enclosing LOOP. The *out* set of an EXIT is the same as its *in* set. It is used to compute the *out* set of the enclosing LOOP.

TRAP

Trap nodes are in principle handled in the same way as EXITs. We could collect the definitions that reach trap nodes. However, we do not think that this information can be usefully exploited by the user.

RETURN

A RETURN statement kills all definitions, i.e. no definitions reach the statements following the RETURN.

Computation of Parameter Usage Information

After computing the reaching definitions, the summary information about the usage of ordinary and additional parameters is computed for later reuse. A parameter may either be used or it may not be used. It may be not defined, it may be defined on some paths or it

may be defined on all paths.

Computation of Summary Edges

Summary edges are computed by intraprocedural slicing: for each formal output parameter (which may be an ordinary parameter or an additional parameter) that has been defined in the procedure we slice the procedure for this output parameter and insert summary edges to all input parameters that have been reached during slicing. The values of these input parameters (which may be ordinary or additional parameters) may influence the output parameter. For functions, we slice the procedure for the procedure exit node and insert summary edges from the procedure exit node to the influencing input parameters. The summary edges are reflected from the called procedure back onto the call sites: A summary edge from a formal output parameter to a formal input parameter becomes a summary edge between the corresponding actual parameters (see Fig. 4.41). A summary edge from the procedure exit node to a formal input parameter becomes a summary edge from the function call node to the corresponding actual parameter (see Fig. 4.15).
Fig. 4.41 shows the AST of Example 4.22. The formal input parameter node of *in* is marked during slicing for the formal output parameter node of *out*, therefore a summary edge is inserted from the formal output parameter node of *out* to the formal input parameter node of *in*. This summary edge is reflected onto all call sites. There is a data dependence edge to the formal input parameter node of *in* which means that the value of *in* is used. Therefore, a *parameter-in* edge is inserted from the actual parameter node to the formal parameter node. The formal input parameter node of *out* is not marked during slicing. This means that the definition of *out* at the formal input parameter node does not reach the formal output parameter node. In other words, *out* is defined on all paths in procedure *Abs*. Thus, the definition of the second parameter is a killing definition at call sites of *Abs*. There is no data dependence edge to the formal input parameter node of *out*. This means that the value of *out* is never used. It is therefore not necessary to insert a *parameter-in* edge between the second parameter at the call sites of *Abs* and the formal input parameter of *out*, but a *parameter-out* edge is necessary, since *out* is defined in *Abs*.

Example 4.22:

```
PROCEDURE Abs (in: INTEGER; VAR out: INTEGER);
BEGIN
   IF in < 0 THEN out := - in
   ELSE out := in
   END
END Abs;

...

Abs(i, j)
```


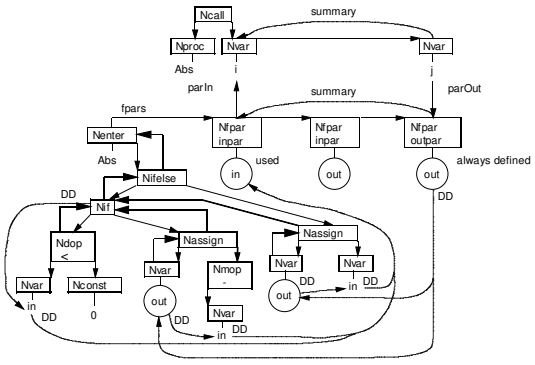

Fig. 4.41 - Computation of summary edges

4.6 Slicing

Our algorithm for static backward slicing is based on the two-pass slicing algorithm of Horwitz et al. [HoRB90] described in Section 3.2.2 where slicing is seen as a graph-reachability problem. This algorithm uses summary information at call sites to account for the calling context of procedures. We compute the summary information by a variation of the algorithm of Livadas et al. [LivC94, LivJ95].

Since we use a fine-grained program representation, the nodes that are considered for inclusion into the slice are the nodes of the abstract syntax tree. This enhances the precision of the static slices.

Since Oberon-2 is a modular programming language with separate compilation, we extended interprocedural slicing to intermodular slicing where the slicing information of modules can be computed separately. Type checking across module boundaries is implemented by reusing type information about the exported interface of a module when compiling dependent modules. In analogy, we reuse slicing information when slicing dependent modules. This slicing information can be stored in a repository.

Since Oberon-2 is an object-oriented programming language, we support inheritance, polymorphism, and dynamic binding.

4.6.1 Intraprocedural Slicing

For intraprocedural slicing we use the algorithm outlined in Fig. 3.2, where control and data dependences as well as summary edges are traversed backwards. When a node is to be included into the slice, it is marked with procedure *MarkNode* shown in Fig. 4.42.

```
PROCEDURE (s: Slice) MarkNode* (node: Node);
   VAR obj: Object;

   PROCEDURE^ MarkObject (obj: Object);
   PROCEDURE MarkStruct (typ: Struct);
   BEGIN
      IF typ is not yet marked THEN
         mark typ
         IF typ is a pointer type, a procedure type or composite type THEN
            MarkStruct(typ.BaseTyp)          (* mark the referenced record type,
                                                the result type, or element type   *)
         END ;
         MarkObject(typ.strobj)              (* mark the symbol table object for the type *)
      END
   END MarkStruct;

   PROCEDURE MarkObject (obj: Object);
   BEGIN
      IF obj is not yet marked THEN
         mark obj
         MarkStruct(obj.typ);                (* mark the type of the object        *)
         IF obj is imported THEN
            MarkObject(obj.mod)              (* mark the declaring module          *)
         END
      END
   END MarkObject;

BEGIN
   IF node is not yet marked THEN
      mark node
      MarkStruct(node.typ);                  (* mark the type of the node          *)
      MarkObject(node.obj);                  (* mark the object referenced by the node  *)
      FOR all objects obj used at node DO    (* mark all objects used at the node  *)
         MarkObject(obj)
      END ;
      FOR all objects obj defined at node DO (* mark all objects defined at the node *)
         MarkObject(obj)
      END
   END
END MarkNode;
```

Fig. 4.42 - Marking a node

Marking not only syntax tree nodes but also objects of the symbol table allows us to visualize which declarations are actually needed for the syntax tree nodes that are part of the slice.

4.6.2 Interprocedural Slicing

For interprocedural slicing we use the algorithm outlined in Fig. 3.7. We will shortly describe which nodes of our intermediate representation of the program are used to model the system dependence graph used by Horwitz et al. [HoRB90]:

- For *procedure entry nodes* we use the Nenter nodes of the abstract syntax tree. They have references to the symbol table object of the procedure and to additional information about the procedure.
- For *call-site nodes* we use the Ncall nodes of the abstract syntax tree. The left sub-tree denotes the procedure or the procedure variable that is called. For dynamically bound calls, Ndyncall nodes are used to link the call site with all possible call destinations.
- For the *actual-in* and *actual-out nodes* we use the nodes of the abstract syntax tree representing the actual parameters. For value parameters, the actual parameter may be an expression tree, for reference parameters, the actual parameter denotes an object (Nvar, Nvarpar, Nfield nodes). For additional parameters there is a Nvarpar/additionalPar node.
- For *formal-in* and *formal-out* nodes we use Nfpar nodes. For ordinary value parameters there is a formal input parameter node. For ordinary reference parameters there is a pair of two formal parameter nodes (Nfpar/inPar and Nfpar/outPar) which both reference the same object. For additional parameters there is a pair of two formal parameter nodes (Nfpar/additionalInPar and Nfpar/additionalOutPar) which reference both the same object.

We use the following kinds of edges to represent the dependences among the nodes of the abstract syntax tree:

- *Control dependences* from the depending node to the node controlling its execution.
- *Data dependences* from the usage node to the reaching definitions.
- *Parameter-in edges* from the formal-in parameter to the actual parameter node.
- *Parameter-out edges* from the actual parameter node to the formal-out parameter node.
- *Summary edges* from formal-out parameter nodes and procedure exit nodes to all formal-in parameter nodes that can be reached via intraprocedural dependences. Corresponding summary edges between the actual parameters and from the function call node to the actual parameters.
- *Call edges* are modeled by ascending from Nenter nodes of procedure *P* to all call sites that statically and/or dynamically call *P*. Some of the call destinations of dynamically bound calls can be disabled via user interaction. The disabled destinations are not visited during slicing.

4.6.3 Intermodular Slicing

Without knowledge about the procedures of imported modules, one would have to make conservative assumptions. The worst case for an external procedure *P* of module *M* would be to assume

- that each value parameter of *P* is used,
- that each reference parameter of *P* is possibly defined (leading to non-killing definitions at the call sites),
- that all global variables of any other module *N* are used since *M* might import *N* and use *N*s variables,
- that all global variables of any other module that are exported without access restriction are possibly defined,
- that all objects and arrays on the heap are used,
- that all objects and arrays on the heap are possibly defined, and
- that each reference parameter of *P* transitively depends on all other input parameters, global variables and objects on the heap.

Conservative assumptions about the parameters of imported procedures would lead to unacceptably large slices. Therefore we allow the user to store the parameter usage information of every module in the repository after the module has been sliced. When slicing modules that import previously sliced modules, the information in the repository is reused to compute more precise slices.

For separate compilation, the compiler uses *symbol files* to store the interface information of a module. These symbol files can be reused when compiling other modules that import previously compiled modules. Strong type checking can be performed across module boundaries. Fig. 4.43 compares the processes of compilation and slicing. On the left-hand side we see that the compiler generates an object (e.g. *A.Obj*) and a symbol file (e.g. *A.Sym*) from a source file (e.g. *A.Mod*). If module *A* imports module *B*, the interface of *B* with its type information is read from the symbol file *B.Sym*. This implies that modules must be compiled before they can be imported by other modules. On the right-hand side we see that the slicer computes slicing information (e.g. for module *A*) from the source file *A.Mod*. If *A* imports *B*, the interface of *B* with its type information is either extracted from the symbol file *B.Sym* or, if the slicing information has already been computed for *B* before, from the repository. Object and symbol files are usually stored in the file system, whereas the slicing information is stored in the repository.

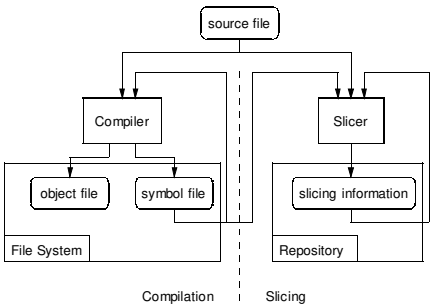

Fig. 4.43 - Compilation versus slicing

The slicing information that is stored in the repository for a module is a superset of the information that would be contained in the symbol file. The slicing information comprises:

- all exported types,
- all other exported objects (constants, variables, procedures),
- all type-bound procedures, and
- all procedures that have somewhere been assigned to a procedure variable.

For each procedure, the repository stores the parameter usage information for all parameters and the *ProcInfo* object with the list of formal parameters and their summary edges.

Version conflicts are checked by the compiler. When the interface of a module changes, the compiler generates a new symbol file and a unique number for this version. This version number is used to detect version conflicts due to changes of the interface. Changes in the implementation that do not change the interface do not lead to a new symbol file and a new version number since they do not invalidate clients of the module. Crelier [Cre94] developed finer-grained methods to extend modules without invalidating clients. However it was out of the scope of this thesis to integrate his ideas. Fig. 4.44 shows the import graph for four modules where *B* imports *D*, *C* imports *D*, and *A* imports *B* and *C*. If the interface of *D* changes, all modules depending on *D* would have to be recompiled. If *B* is recompiled, and then *A* is recompiled, the compiler reports that during the compilation of *A*, module *D* is imported once via *B* in the new version and once via *C* in the old version. Likewise, the loader would detect the version conflict, instead of loading inconsistent modules.

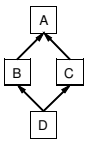

Fig. 4.44 - Import graph of four modules

In the same way, the repository handles version conflicts: If the slicing information for a module is stored into the repository, it replaces the previously existing version of the slicing information. If other modules in the repository depend on the existing version, they are removed from the repository before inserting the new version. Since removal of modules from the repository is an irrevocable action, it is only done if explicitly requested by the user. This process of removing invalidated modules continues recursively. E.g., if modules *D*, *C*, *B*, and *A* have been placed into the repository and a new version of module *B* is checked in, modules *B* and *A* are removed from the repository. If a new version of module *D* is checked in, all four modules are removed. Fig. 4.45 illustrates this recursive process of removing invalidated slicing information.

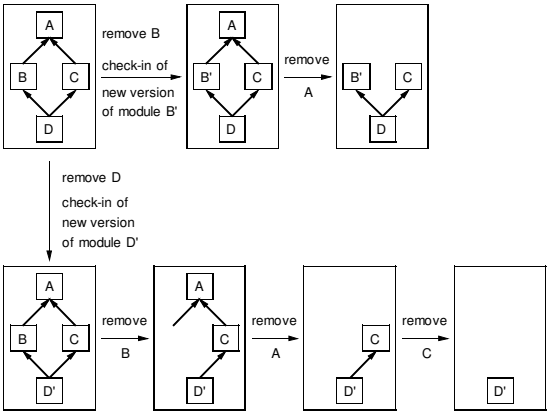

Fig. 4.45 - Recursive removal of invalidated slicing information

4.7 Support of Object-Oriented Features

The Oberon Slicing Tool supports the key concepts of object-oriented programming, such as inheritance, polymorphism and dynamic binding:

- Inheritance: The inheritance relation is modeled as described in Section 4.3. Each class contains information about the type and visibility of its new and inherited fields as well as information about its new, overridden and inherited methods.
- Polymorphism: Polymorphic variables are handled during alias analysis. The sets of possible aliases can be restricted via user feedback. A special problem of alias analysis is that when a field *x* of an object *o* of type *T* is changed via a pointer *p* of type *POINTER TO T* (e.g. "p.x := ..."), the field *x* accessed via a pointer *q* of type *POINTER TO T* (e.g. "... := q.x") may as well be changed (if *p* and *q* both point to object *o*). Since the dynamic type of *p* may be an extension of *T* (e.g. *T1* which is assumed to be derived from *T*), the field *x* accessed via a pointer *q1* of type *POINTER TO T1* may also be changed (if *p* and *q1* both point to object *o*). To the best of our knowledge, program slicinig tools make either extremely conservative assumptions when changing data via pointers (e.g. invalidate all heap-allocated data) or they do not account for the described problem at all.
- Dynamic binding: All possible call destinations are computed for dynamically bound call sites. Calls of methods and calls of procedure variables are handled uniformly. The sets of possible call destinations can be restricted via user feedback.

Information hiding and encapsulation of code and data are not really new features of object-oriented programming but can already be accomplished with modular programming languages such as Modula-2. In order to understand programs that exploit abstraction and information hiding, it is important to make visible to the user which (hidden) data is used during some calculation. This is even more important for object-oriented programs which make heavy use of information hiding. Example 4.23 illustrates the problems of information hiding on a procedural program: Module *Random* exports function *Uniform* which returns a random number and modifies the non-exported variable *state*. Module *Client* imports *Random* and calls *Random.Uniform* twice. If we slice for the last statement in *Client.Do* (the assignment to *z*), we have to include the first call of *R.Uniform* into the slice, since the last call of *R.Uniform* depends on the value of the invisible variable *R.state* which is assigned during the first call of *R.Uniform*. This is not obvious to the user unless the accessed and modified variables are listed in the parameter list of the function call.

Example 4.23:


```
MODULE Random;

VAR state: LONGINT;

PROCEDURE Uniform*(): LONGINT;
BEGIN
   state := ...
END Uniform;

END Random;
```

```
MODULE Client;
IMPORT Random;

PROCEDURE Do*;
BEGIN
   z := Random.Uniform( (*R.state*) );
   ...
   z := Random.Uniform( (*R.state*) );
END Do;

END Client.
```

4.8 Modularization

We have implemented the Oberon Slicing Tool in a set of modules. Fig. 4.46 shows the import graph of the Oberon Slicing Tool. The modules below the dashed line belong to the Oberon-2 compiler, whereas the modules above belong to the Slicer. Module *SlicerOPP* implements a syntax directed top-down recursive-descent parser. Module *SlicerOPT* is the symbol table handler which declares the data types for the abstract syntax tree and the symbol table together with the operations upon them. We have added several auxiliary data structures directly in module *SlicerOPT*, but also extracted some into module *SlicerAuxiliaries*. Module *Repository* stores the slicing information. Module *Slicer* contains the algorithms for control flow and data flow analysis. Module *ParInfoElems* implements parameter information elements of the graphical user interface. Finally, module *SlicerFE* is the front end of the Oberon Slicing Tool.

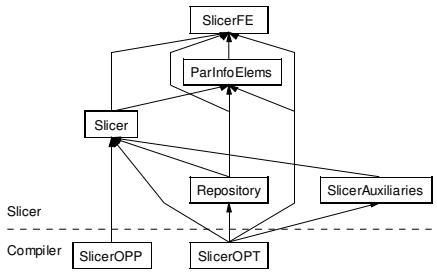

Fig. 4.46 - Import graph of the Oberon Slicing Tool

Table 4.12 shows the sizes of the particular modules in lines of code, the number of statements, and the bytes of the object code. Module *SlicerOPP* has only been marginally changed, two thirds of module *SlicerOPT* are new, the rest is reused. After subtracting the parts of the Oberon-2 compiler, the Oberon Slicing Tool consists of approximately 12500 lines of code, 9000 statements and 160000 bytes of object code (compiled for Intel x86 processors).

Module	Lines of Code	Statements	Object Code
SlicerOPP	1309	1180	18046
SlicerOPT	2585	1656	33958
SlicerAuxiliaries	208	95	1800
Repository	1622	1101	23029
Slicer	3916	2585	52002
ParInfoElems	525	362	6337
SlicerFE	3381	2721	51305

Table 4.12 - Modules of the Oberon Slicing Tool

In the following we will shortly describe the interfaces of these modules.

4.8.1 Module Repository

Module *Repository* stores the slicing information for processed modules.

```
DEFINITION Repository;

   IMPORT SlicerOPT, SlicerOPS;

   CONST
      version = "Oberon Slicing Tool V1.0 (CS)";
      defaultRepository = "Repository.Rep";          (* default file name of the repository  *)
      optionChar = "\";
      unexpectedSituation = 99;                      (* run-time error number               *)
      (* kinds of parameter usages *)
      parIn = 0; parOut = 1; parUsed = 2; parDefined = 3; parAlwaysDefined = 4; isPar = 5;
      unknown = 6;

   VAR
      modules-: SlicerOPT.ObjArr;

   PROCEDURE GetKey (modName: ARRAY OF CHAR): LONGINT;
   PROCEDURE MakePersistent (key: LONGINT; mod, topScope: SlicerOPT.Object;
      force: BOOLEAN);
   PROCEDURE ThisMod (modName: ARRAY OF CHAR; key: LONGINT): SlicerOPT.Object;
   PROCEDURE ShowModules;
   PROCEDURE ShowModuleInfo;

   PROCEDURE SetObjUsage (proc, obj: SlicerOPT.Object; expand: BOOLEAN; usage: SET);
   PROCEDURE GetObjUsage (proc, obj: SlicerOPT.Object; VAR usage: SET);
   PROCEDURE ChangeObjUsage (proc, obj: SlicerOPT.Object; usage: SHORTINT;
      incl: BOOLEAN);
```



```
   PROCEDURE RemoveObjUsageForProc (proc: SlicerOPT.Object);
   PROCEDURE DumpObjUsage;

   PROCEDURE Save;
   PROCEDURE Load;
   PROCEDURE CompleteComputation;
   PROCEDURE Reset;

END Repository.
```

Variables:

- *modules* is the array of modules for which slicing information is stored in the repository.

Operations:

- *GetKey(modName)* returns the *key* of the module *modName*. It is a version number computed by the compiler when generating the symbol file. Whenever the interface of the module changes, a new key is generated. This key is used to detect version conflicts during separate compilation. However, it cannot be used to detect inconsistencies between different versions if the interface of the module has not changed.
- *MakePersistent(key, mod, topScope, force)* makes the slicing information that is stored in the symbol table *topScope* of module *mod* with the version *key* persistent. If the repository already contains slicing information for this module and the old information is not used by other modules, it is simply replaced. If the old information is used by other modules, it is only replaced if *force* is TRUE. Then the information for all dependent modules is recursively deleted (see Fig. 4.45).
- *ThisMod(modName, key)* returns the root of the symbol table of module *modName* with the specified *key*.
- *ShowModules* lists all modules for which slicing information is stored in the repository.
- *ShowModuleInfo modName* lists the slicing information stored for module *modName*.
- *SetObjUsage(proc, obj, expand, usage)* sets the parameter usage information for parameter *obj* in procedure *proc* to the specified *usage*. *usage* is a set and may contain *parIn, parOut, parUsed, parDefined, parAlwaysDefined*, and *isPar* as its elements. If *expand* is TRUE and the parameter is a record or a pointer to a record, the same usage information is applied to all its fields.
- *GetObjUsage(proc, obj, usage)* returns in *usage* the parameter usage information stored for parameter *obj* of procedure *proc*. *usage* may be {unknown} if the parameter usage information is not available.
- *ChangeObjUsage(proc, obj, usage, incl)* includes or excludes (depending on the Boolean value *incl*) the specified usage for parameter *obj* of procedure *proc*.
- *RemoveObjUsageForProc(proc)* removes the parameter usage information for procedure *proc*.
- *DumpObjUsage* outputs all parameter usage information.

- *Save* [*fileName* [*options*]] stores the repository in the specified file. The default options are to store the pre-declared data types and built-in functions of Oberon-2, the object usage information, and the slicing information of all modules. The following parameters can modify these default options:
  - *s* pre-declared data types and built-in functions are not stored
  - *o* object usage information is not stored
  - *m* slicing information of all modules is not stored
- *Load* [*fileName* [*options*]] loads the repository from the specified file. The default options are to load the pre-declared data types and built-in function of Oberon-2, the object usage information, and the slicing information of all modules. The specified options must match the options used for storing.
- *CompleteComputation* removes unnecessary object usage information.

4.8.2 Module Slicer

Module *Slicer* declares type *Slice* which implements control flow and data flow analysis necessary to derive slices.

```
DEFINITION Slicer;

  IMPORT SlicerOPT, SlicerOPS;

  CONST
    version = "Oberon Slicing Tool V1.0 (CS)";
    unexpectedSituation = 99;
    (* Notifier op *)
    changed = 0; nodeMarked = 1; sliceComputed = 2; controlFlowComputed = 3;
    dataFlowComputed = 4; dataFlowInfoReset = 5; markChanged = 6;
    compiled = 7;

  TYPE
    Notifier = PROCEDURE (s: Slice; op: INTEGER);

    Slice = POINTER TO SliceDesc;
    SliceDesc = RECORD
      moduleName-: ARRAY 32 OF CHAR;          (* e.g. Test for Test.Obj, if the module
                                                 starts with MODULE Test          *)
      moduleFileName-: ARRAY 256 OF CHAR;     (* e.g. TestModule.Mod              *)
      notify: Notifier;                       (* called if the slice is changed   *)
      program: SlicerOPT.Node;                (* abstract syntax tree of the slice *)
      topScope: SlicerOPT.Object;             (* symbol table of the slice        *)
      moduleNames-: ARRAY 31 OF SlicerOPS.Name; (* names of imported modules       *)
      Message: PROCEDURE (str: ARRAY OF CHAR; node: SlicerOPT.Node;
        kind: SHORTINT);

      PROCEDURE (s: Slice) Compile (mod: ARRAY OF CHAR; VAR done: BOOLEAN);
      PROCEDURE (s: Slice) BuildClassHierarchy;
      PROCEDURE (s: Slice) ControlFlow;
      PROCEDURE (s: Slice) DataFlow;
      PROCEDURE (s: Slice) SliceProc (node: SlicerOPT.Node; interprocedural: BOOLEAN);
```



```
      PROCEDURE (s: Slice) SliceProcForObj (proc: SlicerOPT.Node; obj: SlicerOPT.Object;
        interprocedural: BOOLEAN);
      PROCEDURE (s: Slice) SliceStat (node: SlicerOPT.Node; interprocedural: BOOLEAN);
      PROCEDURE (s: Slice) ResetDataFlowInfo;
      PROCEDURE (s: Slice) Statistics;
      PROCEDURE (s: Slice) CountMarkedNodes (VAR marked, total: LONGINT);
      PROCEDURE (s: Slice) CompleteComputation;

      PROCEDURE (s: Slice) IsImported (obj: SlicerOPT.Object): BOOLEAN;
      PROCEDURE (s: Slice) MayAlias (o1, o2, proc: SlicerOPT.Object): BOOLEAN;
      PROCEDURE (s: Slice) MarkNode (node: SlicerOPT.Node);
      PROCEDURE (s: Slice) MarkedNode (node: SlicerOPT.Node): BOOLEAN;
      PROCEDURE (s: Slice) MarkedObject (obj: SlicerOPT.Object): BOOLEAN;
      PROCEDURE (s: Slice) MarkedStruct (typ: SlicerOPT.Struct): BOOLEAN;
      PROCEDURE (s: Slice) SetModuleFileName (str: ARRAY OF CHAR);
      PROCEDURE (s: Slice) SetModuleName (str: ARRAY OF CHAR);
      PROCEDURE (s: Slice) ThisNode (pos: LONGINT): SlicerOPT.Node;
      PROCEDURE (s: Slice) ThisProc (name: ARRAY OF CHAR): SlicerOPT.Node;
      PROCEDURE (s: Slice) ThisProcFromObj (obj: SlicerOPT.Object): SlicerOPT.Node;
    END ;

    SliceFactoryMethodType = PROCEDURE (): Slice;

  VAR
    arrayExpansionLimit-: INTEGER;
    sliceFactoryMethod-: SliceFactoryMethodType;

  PROCEDURE InitSlice (s: Slice);
  PROCEDURE SliceFactoryMethod (): Slice;
  PROCEDURE InstallDefaultSlicer;
  PROCEDURE SetSliceFactoryMethod (f: SliceFactoryMethodType);
  PROCEDURE SetArrayExpansionLimit (limit: INTEGER);

END Slicer.
```

Variables:

- *arrayExpansionLimit* sets the upper limit for the expansion of arrays as described in Section 4.5.1.
- *sliceFactoryMethod* is used to allocate new objects of type *Slice*.

Operations:

- *InitSlice(s)* initializes a newly allocated object of type *Slice*.
- *SliceFactoryMethod* allocates an object of type *Slice*, initializes it and returns it.
- *InstallDefaultSlicer* installs *SliceFactoryMethod* in the procedure variable *sliceFactoryMethod*.
- *SetSliceFactoryMethod(f)* installs the factory method *f* in the procedure variable *sliceFactoryMethod*.
- *SetArrayExpansionLimit(limit)* sets the variable *arrayExpansionLimit* to *limit*.

Types:

- *SliceFactoryMethodType* is the procedure type for factory methods that can be installed to allocate slices.
- *Notifier* is the procedure type for notifiers that can be installed in slices. They will be called when the slice is changed, when a node is marked, when the slice is computed, when control flow information or data flow information has been computed, when data flow information has been reset or when the mark of the slice has changed, or when the module has been compiled. The parameter *op* of the notifier indicates the operation.
- *Slice* is the main class of the Oberon Slicing Tool.

Methods:

- *s.Compile(mod, done)* compiles the module *mod*. *done* indicates the success of the operation.
- *s.BuildClassHierarchy* builds the class hierarchy. This method must be called after the module is compiled but before control flow and data flow information is computed.
- *s.ControlFlow* computes control flow information as described in Section 4.4. This method must be called after method *BuildClassHierarchy* but before method *DataFlow*.
- *s.DataFlow* computes data flow information as described in Section 4.5. This method must be called after method *ControlFlow*.
- *s.SliceProc(node, interprocedural)* derives the slice starting with the specified node. If *interprocedural* is TRUE, the slice is computed interprocedurally, otherwise intraprocedurally.
- *s.SliceProcForObj(proc, obj, interprocedural)* derives the interprocedural or intraprocedural slice for the output parameter *obj* of procedure *proc*.
- *s.SliceStat(node, interprocedural)* derives the interprocedural or intraprocedural slice for the statement *node*.
- *s.ResetDataFlowinfo* resets the data flow information. This method can be called after the user restricted the sets of possible aliases or the sets of possible call destinations. Afterwards more precise data flow information can be re-computed by method *DataFlow*.
- *s.Statistics* outputs statistical information.
- *s.CountMarkedNodes(marked, total)* returns the number of marked nodes (i.e. the number of nodes that are part of the slice) and the total number of nodes in the abstract syntax tree of the program.
- *s.CompleteComputation* removes information from the abstract syntax tree and the symbol table that is only necessary for computing the slices but not if the slice is to be stored in the repository. This method is called before the symbol table of the slice is checked in into the repository. After calling this method, no more slices can be computed.
- *s.IsImported(obj)* returns TRUE if the object *obj* is imported.
- *s.MayAlias(o1, o2, proc)* returns TRUE if object *o1* and *o2* may be aliases in procedure

proc. This method may be overridden to provide more precise alias analysis.

- *s.MarkNode(node)* marks the specified node as described in Section 4.6.1.
- *s.MarkedNode(node)* returns TRUE if the specified node is marked, otherwise FALSE.
- *s.MarkedObject(obj)* returns TRUE if the specified object is marked, otherwise FALSE.
- *s.MarkedStruct(typ)* returns TRUE if the specified type is marked, otherwise FALSE.
- *s.SetModuleFileName(str)* sets variable *s.moduleFileName* to *str*.
- *s.SetModuleName(str)* sets variable *s.moduleName* to *str*.
- *s.ThisNode(pos)* returns the node of the abstract syntax tree for the specified source code position.
- *s.ThisProc(name)* returns the procedure entry node for the specified procedure.
- *s.ThisProcFromObj(obj)* returns the procedure entry node for the specified procedure.

The *Factory Method* design pattern [GaHJV95] has been used to make the process of allocation of objects of type *Slice* flexible. Factory methods can be installed to allocate instances of subclasses of type *Slice*.

Module *MeasuringSlicer* uses the *Decorator Pattern* to add measuring functionality to the slices. It extends the type *Slicer.Slice* and overrides the exported methods: Each method contains a prolog and epilog to measure the time used for the operation. Method *measuringSlicer.Statistics* outputs the statistics about the minimum, maximum and average time used to perform the specific operations. Fig. 4.47 shows the pattern of methods:

```
PROCEDURE (s: Slice) Method* (parameters)
BEGIN
  start measuring
  s.Method^(parameters)      (* super call *)
  stop measuring
  remember elapsed time
END Method;
```

Fig. 4.47 - Pattern for methods of type *MeasuringSlicer.Slice*

5 User Interface

The user interface of the Oberon Slicing Tool is implemented in a separate module, module *SlicerFE*. It displays the source code of the analyzed program in a canonical form by reconstructing it from the abstract syntax tree and the symbol table. The user can slice for specific statements of the program by clicking on the line of the statement. The slice is then computed and visualized by showing all parts of the program that belong to the slice in a different color. Furthermore, control flow and data flow information is visualized by active text elements.

5.1 Visual Elements

We extended the Active Text Framework (see [Szy92] and [MöKo96]) of the Oberon System to visualize control flow and data flow information. This framework allows arbitrary objects such as pictures, tables or buttons to be inserted into the text like ordinary characters. These objects are called *text elements* and are derived from the abstract base class *Texts.Elem*. 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. Another kind of active text elements are *popup elements* that represent a menu that pops up in reaction to a click.

5.1.1 Bidirectional Links Between the Caller and the Callee

We visualize the call edges between the call site and the called procedure with hypertext links. At the call site, we insert a popup element labeled *calling*. It contains one entry for each possible call destination. The user can select a call destination from the popup element upon which the source code is scrolled to the position of the called procedure and the called procedure is highlighted. At the called procedure, a popup element labeled *called at* contains all call sites. The user can select a call site from the popup element upon which the source code is scrolled to the position of the call site and the call site is highlighted. Fig. 5.1 shows a small program with popup elements at the call sites and the called procedures. The numbers at the beginning of the lines indicate the character position within the original source code.


```
     MODULE VisualizeLinks;

     IMPORT In, Texts, Files;


 43  PROCEDURE ReadParameters (VAR name: ARRAY OF CHAR; VAR option: INTEGER); called at
     BEGIN                                                                    289
123    In.Open(); calling                                                     422
132    In.Name(name); calling
147    In.Int(option); calling
     END ReadParameters;

183  PROCEDURE Compile*;
        VAR fileName: ARRAY 32 OF CHAR; option: INTEGER; f: Files.File; len: LONGINT;
     BEGIN
289    ReadParameters(fileName, option); calling
324    len := 0
     END Compile;

348  PROCEDURE Show*;
        VAR fileName: ARRAY 32 OF CHAR; option: INTEGER;
     BEGIN
422    ReadParameters(fileName, option); calling
     END Show;

     END VisualizeLinks.
```

Fig. 5.1 - Bidirectional links between the caller and the callee

After selecting the entry 289 from the *called at* element of procedure *ReadParameters*, the call site in procedure *Compile* is highlighted as shown in Fig. 5.2.


```
     MODULE VisualizeLinks;

     IMPORT In, Texts, Files;


 43  PROCEDURE ReadParameters (VAR name: ARRAY OF CHAR; VAR option: INTEGER); called at
     BEGIN
123    In.Open(); calling
132    In.Name(name); calling
147    In.Int(option); calling
     END ReadParameters;

183  PROCEDURE Compile*;
        VAR fileName: ARRAY 32 OF CHAR; option: INTEGER; f: Files.File; len: LONGINT;
     BEGIN
289    ReadParameters(fileName, option); calling
324    len := 0
     END Compile;

348  PROCEDURE Show*;
        VAR fileName: ARRAY 32 OF CHAR; option: INTEGER;
     BEGIN
422    ReadParameters(fileName, option); calling
     END Show;

     END VisualizeLinks.
```

Fig. 5.2 - Navigation from call destination back to call site

5.1.2 Data Dependences

We visualize data dependences with links from the usage of a variable to all definitions of the same variable that might reach the usage. Popup elements labeled *DDs* contain one entry for each reaching definition. The user can select a reaching definition which is then highlighted in the source code. An exclamation mark in the label of the popup element indicates potential data flow problems (e.g., usage of a potentially uninitialized variable). Fig. 5.3 shows a small program with data dependences. The popped up element shows that all three definitions of *p* might be reaching.


```
     MODULE VisualizeDDs;

     TYPE
        BinTree = POINTER TO BinTreeDesc;
        BinTreeDesc = RECORD left, right: BinTree; val: INTEGER END ;

131     PROCEDURE Find (t: BinTree; i: INTEGER);
           VAR p: BinTree;
        BEGIN
196        p := t.DDs;
205        WHILE (p.DDs # NIL) & (p.DDs.val.DDs # i.DDs) DO
240           IF i.DDs < p.DDs.val.DDs THEN
258              p := p.DDs.left.DDs
                 ELSE
275              p := p.DDs.right.DDs
              END     DD on 258
           END        DD on 275
        END Find;     DD on 196

     END VisualizeDDs.
```

Fig. 5.3 - Visualization of data dependences

5.1.3 Parameters

Parameter Usage

Parameter usage information elements visualize the flow of the parameters between the call sites and the called procedures. Parameters may be used by the called procedure - their values flow from the caller to the callee. They may be defined by the called procedure, in which case their values may flow back from the callee to the caller (for reference parameters only). At the call sites, parameter information elements indicate the flow of information (↓ for input parameters, ↑ for output parameters, ↕ for input/output parameters). At the called procedure, similar elements give additional information about potential problems (↓! for problems with input parameters, e.g. if a reference parameter *is not assigned* a value or if an input parameter *is* assigned a value). Fig. 5.4 shows a small program with parameter information elements. At the call site in line with number 357, *t* is an input parameter, and *min* an output parameter. By clicking with the middle mouse button on the element of the output parameter *min* of procedure *FindMin*, the slice is computed for this output parameter as shown in Fig. 5.4. All parts of the program that are part of the slice are shown in blue.

```
MODULE VisualizeParElems;

TYPE
   BinTree = POINTER TO BinTreeDesc;
   BinTreeDesc = RECORD left, right: BinTree; val: INTEGER END ;

136   PROCEDURE FindMin (↓ t: BinTree; VAR ↑ min: INTEGER);
         VAR p: BinTree;
      BEGIN
209      ASSERT(t # NIL);
227      p := t;
236      WHILE p.left # NIL DO
258         p := p.left
         END ;
277      min := p.val
      END FindMin;

304   PROCEDURE Do*;
         VAR t: BinTree; min: INTEGER;
      BEGIN
357      FindMin(↓t, ↑min)
      END Do;

END VisualizeParElems.
```

Fig. 5.4 - Parameter information elements

By clicking with the middle and the right mouse button on the parameter information element of a formal reference parameter, information about possible aliases is shown in a small popup window. Fig. 5.5 shows that parameter *j* is a possible alias of parameter *i*.

```
MODULE VisualizeAliases;

                                  Possible aliases: j
26    PROCEDURE Do* (VAR ↕ i, ↕j: INTEGER);
      BEGIN
68       i := ABS(i)
      END Do;

END VisualizeAliases.
```

Fig. 5.5 - Alias information via parameter information elements

Additional information about potential problems is also shown in a small popup window. Fig. 5.6 shows the information for parameter *j* whose value is never used in procedure *Do* and which is not assigned a value in the procedure.


```
MODULE VisualizeAliases;                Possible aliases: i
                                        set but never used
                                        used as varpar, possibly not set
26    PROCEDURE Do* (VAR ↕ i, ↕j: INTEGER);
      BEGIN
68       i := ABS(i)
      END Do;

END VisualizeAliases.
```

Fig. 5.6 - Additional information via parameter information elements

For dynamically bound calls, the parameter elements combine the parameter usage information from all possible call destinations.

Additional Parameters

Additional parameters are shown in comments in the actual and formal parameter lists. Fig. 5.7 shows a small example. Procedure *Add0* defines the global variable *sum*, therefore *sum* is added as an additional parameter to its formal parameter list. In line with number 258, procedure *Add* calls procedure *Add0* with the ordinary parameter *val* and the additional parameter *sum*. This additional parameter is added as a comment to the formal parameter list of procedure *Add*. For additional parameters parameter information elements indicate their usage as previously described for ordinary parameters.

```
MODULE VisualizeAdditionalPars;

IMPORT In, Texts, Out;

VAR
   sum: INTEGER;

70    PROCEDURE Show* ((* VAR ↓↕sum: INTEGER *));
      BEGIN
94       Out.Int(↓sum, ↓0);
111      Out.Ln
      END Show;

130   PROCEDURE Add0 (↓ val: INTEGER    (* VAR ↕ sum: INTEGER *));
      BEGIN
168      INC(sum, val)
      END Add0;

193   PROCEDURE Add* ((* VAR ↕ sum: INTEGER *));
         VAR val: INTEGER;
      BEGIN
235      In.Open;
244      In.Int(↕val);
258      Add0(↓val  (* ↕sum*))
      END Add;

END VisualizeAdditionalPars.
```

Fig. 5.7 - Additional parameters

For dynamically bound calls, additional parameters are collected from all possible call destinations.

5.1.4 Aliases

For each definition of a variable with possible aliases we insert a popup element labeled *aliases*. It contains one entry for each variable that might be an alias of the defined variable. For each possible alias, non-killing definitions are generated. The user can disable and enable some of the aliases by selecting them. Initially all aliases are enabled. The popup element shows enabled aliases in black and disabled aliases in grey. After an alias has been disabled via user interaction, the user can initiate the computation of more precise data flow information. Fig. 5.8 shows a small example where variables *cnt* and *arr* may be aliases (e.g., for a call like *VisualizeAliases.CountZero(arr, arr[i])*).

```
MODULE VisualizeAliases;

26    PROCEDURE CountZero* (VAR ↓↕arr: ARRAY OF INTEGER; VAR ↕ cnt: INTEGER);
         VAR i, len: LONGINT;
      BEGIN
123      cnt[aliases] := 0;
133      i := [all on]
141      len [all off] (arr);
159      WH  arr  len DO
178         IF arr[i] = 0 THEN
197            INC(cnt[aliases])
            END      all on
         END         all off
      END CountZero  arr

END VisualizeAliases.
```

Fig. 5.8 - Possible aliases at definitions

For all enabled aliases non-killing definitions are generated which leads to conservative data flow information. In Fig. 5.9 we see that the two assignments to *cnt* in lines with numbers 123 and 197 are reaching definitions for the usage of *arr* in line with number 178.


```
MODULE VisualizeAliases;

26    PROCEDURE CountZero* (VAR ↓↕arr: ARRAY OF INTEGER; VAR ↕ cnt: INTEGER);
         VAR i, len: LONGINT;
      BEGIN
123      cnt[aliases] := 0;
133      i := 0;
141      len := LEN(arr[DDs]);
159      WHILE i[DDs] < len[DDs] DO
178         IF arr[DDs, i[DDs]] = 0 THEN
197            INC(cnt[DD on 197, aliases])
            END  DD on 123
         END     DD on 52
      END CountZero;

END VisualizeAliases.
```

Fig. 5.9 - Reaching definitions with all aliases enabled

After disabling the aliases at the assignments in lines with numbers 123 and 197, the user can reset the computed data flow information and initiate its recomputation. Fig. 5.10 shows the more precise data flow information where only the initial values (parameter *arr* at position 52) reach the usage node in line with number 178.

```
MODULE VisualizeAliases;

26    PROCEDURE CountZero* (VAR ↓↕arr: ARRAY OF INTEGER; VAR ↕ cnt: INTEGER);
         VAR i, len: LONGINT;
      BEGIN
123      cnt[aliases] := 0;
133      i := 0;
141      len := LEN(arr[DDs]);
159      WHILE i[DDs] < len[DDs] DO
178         IF arr[DDs, i[DDs]] = 0 THEN
197            INC(cnt[DD on 52, aliases])
            END
         END
      END CountZero;

END VisualizeAliases.
```

Fig. 5.10 - Reaching definitions with all aliases disabled

5.1.5 Dynamic Types

A polymorphic pointer variable may point to objects of its static type (the type specified at the declaration) or to objects of all extensions of its static type. The type of the object that the pointer actually refers to at run time is called the dynamic type of the pointer variable. We have extended the notion of the "dynamic type" of a pointer variable to procedure variables, where the dynamic types are the procedures that may have been assigned to the

procedure variable. This allows us to treat dynamic binding due to type-bound procedures and procedure variables uniformly. We insert popup elements labeled *dynamic types* at dynamically bound call sites. These elements contain one entry for each possible call destination. The user can disable and enable some of the dynamic types by selecting them. Initially all dynamic types are enabled. The popup element shows enabled dynamic types in black and disabled dynamic types in grey. In Fig. 5.11, 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 in line with number 289.


```
        MODULE VisualizeDynTypes;

        IMPORT Out;

        TYPE
           Node = POINTER TO NodeDesc;
        VAR
           head: Node; count: INTEGER;
        TYPE
           NodeDesc = RECORD i: INTEGER; next: Node END ;
           WorkProc = PROCEDURE (n: Node);

197     PROCEDURE ForAll* (↓ workProc: WorkProc    (* VAR ↕ count: INTEGER; VAR ↓head: Node *));
           VAR n: Node;
        BEGIN
258        n := head;
270        WHILE n # NIL DO
289           workProc(↓n  (* ↕count*)) dynamic types ;
304           n := n.next
           END
        END ForAll;

334     PROCEDURE Inc (↓In: Node    (* VAR ↕ count: INTEGER *));
        BEGIN
365        INC(count)
        END Inc;

386     PROCEDURE Count* ((* VAR ↕ count: INTEGER; VAR ↕ head: Node *));
        BEGIN
411        count := 0;
424        ForAll(↓Inc  (* ↕count, ↓head*));
438        Out.Int(↓count, ↓0);
457        Out.String(↓" elements.");
483        Out.Ln
        END Count;

502     PROCEDURE PrintNode (↓ n: Node);
        BEGIN
540        Out.Int(↓n.i, ↓0);
557        Out.Ln
        END PrintNode;

580     PROCEDURE Print* ((* VAR ↕ count: INTEGER; VAR ↕ head: Node *));
        BEGIN
605        ForAll(↓PrintNode  (* ↕count, ↓head*))
        END Print;

        END VisualizeDynTypes.
```

all on
all off
Inc
PrintNode

Fig. 5.11 - Dynamic types of procedure variables

The user can disable the call destination *Inc* at the call in line with number 289 and initiate the computation of more precise data flow information. Fig. 5.12 shows the same program after disabling the call destination *Inc*. Only the additional parameters relevant to the test case *Print* are shown.


```
        MODULE VisualizeDynTypes;

        IMPORT Out;

        TYPE
           Node = POINTER TO NodeDesc;
        VAR
           head: Node; count: INTEGER;
        TYPE
           NodeDesc = RECORD i: INTEGER; next: Node END ;
           WorkProc = PROCEDURE (n: Node);

197     PROCEDURE ForAll* (↓ workProc: WorkProc    (* VAR ↓head: Node *));
           VAR n: Node;
        BEGIN
258        n := head;
270        WHILE n # NIL DO
289           workProc(↓n) dynamic types ;
304           n := n.next
           END
        END ForAll;

334     PROCEDURE Inc (↓In: Node    (* VAR ↕ count: INTEGER *));
        BEGIN
365        INC(count)
        END Inc;

386     PROCEDURE Count* ((* VAR ↕ head: Node; VAR ↕ count: INTEGER *));
        BEGIN
411        count := 0;
424        ForAll(↓Inc  (* ↓head*));
438        Out.Int(↓count, ↓0);
457        Out.String(↓" elements.");
483        Out.Ln
        END Count;

502     PROCEDURE PrintNode (↓ n: Node);
        BEGIN
540        Out.Int(↓n.i, ↓0);
557        Out.Ln
        END PrintNode;

580     PROCEDURE Print* ((* VAR ↕ head: Node *));
        BEGIN
605        ForAll(↓PrintNode  (* ↓head*))
        END Print;

        END VisualizeDynTypes.
```

all on
all off
Inc
PrintNode

Fig. 5.12 - More precise data flow information after disabling the call destination *Inc*

5.2 User Feedback

Static analysis can derive information only from the source code. The information must be valid for *all* possible executions of the program. As noted earlier, conservative assumptions must be taken if the program uses conditional branches and iteration since it is not known at compile time which branches will be taken at run time and how many iterations there will be. Dynamic analysis can derive information by monitoring one particular execution of the program. It can consider the actual values of the variables during this execution. Therefore, the information is only valid for the particular execution but not in general. Static information is necessarily more general and 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.

Two main sources of imprecision of static analysis are dynamic types of polymorphic variables and alias definitions. The first lead to unnecessarily big slices because all possible call destinations are traversed at dynamically bound calls. The latter lead to unnecessarily big slices because of non-killing definitions for all possible aliases.

The goal of our thesis was to develop a fast, interactive tool for static program slicing. This ruled out the possibility to use dynamic analysis. Nevertheless we wanted to narrow the gap between static and dynamic analysis. We achieved that by integrating user feedback into our algorithms. The user can restrict the dynamic types of polymorphic variables and the sets of possible aliases since he often has some use case in mind that he wants to investigate or that has led to an error. In order to find the error faster, it may be very effective to slice the program for the erroneous statement and then to narrow the program even further by giving feedback about the intended use case to the slicing tool. The cycle of slicing and feeding back user input to the slicing tool may continue several times. The user feedback can be recorded in order to be played back later.

Zhang and Ryder showed that alias analysis in the presence of procedure variables is NP-hard in most cases [ZhR94]. This justifies the use of safe approximations in the first place since exact algorithms would be prohibitive for an interactive slicing tool where the maximum response time must be in the order of seconds. More precise control and data flow information can be computed after the user has restricted the program to the use case that he has in mind. The derived information is no longer valid for all possible executions.

When we compare the precision of the information derived by user feedback with that of dynamic information derived for the same use case and static information, we see that it lies between the two extremes, narrowing the gap between static and dynamic analysis.

5.3 Module SlicerFE

Module *SlicerFE* implements the front end of the Oberon Slicing Tool.

```
DEFINITION SlicerFE;

   IMPORT TextFrames, Display, Texts, Slicer, PopupElems, SlicerOPT, SlicerOPS;

   CONST
      version = "Oberon Slicing Tool V1.0 (CS)";
      unexpectedSituation = 99;                    (* run-time error number      *)
      withDDElems = 1;                             (* options                    *)
      withParInfoElems = 2; withActualParElems = 3;
      withCallingElems = 4; withCalledAtElems = 5;
      withDynTypeElems = 6; withAliasElems = 7;
      withParameterSummary = 8; withReachingEnd = 9;
      withPosition = 10; interprocedural = 11;
      defaultOptions = {withDDElems, withParInfoElems, withActualParElems,
         withCallingElems, withCalledAtElems, withDynTypeElems, withAliasElems,
         withPosition, interprocedural};

   TYPE
      AliasElem = POINTER TO RECORD (PopupElems.ElemDesc) END ;
      DynTypeElem = POINTER TO RECORD (PopupElems.ElemDesc) END ;
      Frame = POINTER TO FrameDesc;
      FrameDesc = RECORD (TextFrames.FrameDesc)
         slice: Slicer.Slice;
         options: SET;
      END ;
      SliceMsg = RECORD (Display.FrameMsg)
         slice: Slicer.Slice;
         op: INTEGER;
      END ;

      ...

   VAR
      forcePersistence: BOOLEAN;
      recording: BOOLEAN;

   PROCEDURE Open;
   PROCEDURE OpenCallHierarchyViewer;
   PROCEDURE ControlFlow;
   PROCEDURE DataFlow;
   PROCEDURE ResetDataFlowInfo;
   PROCEDURE ReconstructSource;
   PROCEDURE Statistics;
   PROCEDURE MakePersistent;
   PROCEDURE InspectSlice;
   PROCEDURE SetRecording;
   PROCEDURE Playback;
   PROCEDURE SetAliases;
   PROCEDURE SetDynamicTypes;
   PROCEDURE SetArrayExpansionLimit;
   PROCEDURE SetForcePersistence;
```



```
   PROCEDURE SetOption;
   PROCEDURE ShowOptions;
   PROCEDURE FindNode;
   PROCEDURE FindProc;
   ...

END SlicerFE.
```

Variables:

- *forcePersistence* is a Boolean value used as the last parameter for *Repository.MakePersistent*.
- *recording* is a Boolean value indicating whether user input shall be recorded for later play-back.

Commands:

- *Open [^] moduleName* compiles the specified module, performs control flow and data flow computation and opens a slicing viewer.
- *OpenCallHierarchyViewer* opens a viewer displaying the call hierarchy of the target slice.
- *ControlFlow* calls method *ControlFlow* for the target slice. If the command is executed from the menu, the target slice is the slice visualized by the viewer. Otherwise, the target slice is the slice visualized by the star-marked viewer (mark must be visible).
- *DataFlow* calls method *DataFlow* for the target slice.
- *ResetDataFlowInfo* calls method *ResetDataFlowInfo* for the target slice.
- *ReconstructSource* reconstructs the source code of the slice visualized by the target viewer. This can be useful after changing the options used for the visualization or after recomputing the data flow information.
- *Statistics* calls method *Statistics* for the target slice.
- *MakePersistent* makes the slicing information persistent by first calling method *CompleteComputation* for the target slice, and then calling procedures *CompleteComputation* and *MakePersistent* from the *Repository*.
- *InspectSlice* opens a viewer that shows the fields of the target slice.
- *SetRecording [^] ["Y" | "N"]* sets the Boolean variable *recording* to TRUE or FALSE.
- *Playback [^] {recorded user feedback}* plays back the previously recorded user feedback to the target slice.
- *SetAliases ("on" | "off" | "toggle") ("all" | variableName)* enables, disables or toggles all aliases or the alias of the specified variable for all alias elements in the current selection.
- *SetDynamicTypes ("on" | "off" | "toggle") ("all" | dynType)* enables, disables or toggles all dynamic types or the specified dynamic type for all dynamic type elements in the current selection.
- *SetArrayExpansionLimit [^] limit* calls *Slicer.SetArrayExpansionLimit* with the specified limit.
- *SetForcePersistence [^] ["Y" | "N"] sets the* Boolean variable *forcePersistence* to TRUE or

FALSE.

- *SetOption [^] {(name | number) ("Y" | "N")} | "defaultOptions"* sets or resets ("Y" or "N") the specified options or the default options for the target slice. The option may be specified by number (e.g., 1 for *withDDElems*) or by name (e.g., *withDDElems*).
- *ShowOptions* outputs the options used to reconstruct the source code of the target viewer.
- *FindNode [^] position* calls method *ThisNode(position)* for the target slice. The return value is stored in the variable *SlicerFE.node* which can be inspected via *InspectSlice*.
- *FindProc [^] procName* calls method *ThisProc(procName)* for the target slice. The return value is stored in the variable *SlicerFE.node* which can be inspected via *InspectSlice*.

Types:

- *AliasElem* is the type of popup elements representing the sets of enabled/disabled aliases.
- *DynTypeElem* is the type of popup elements representing the sets of enabled/disabled dynamic types.
- *FrameDesc* is the type of frame visualizing the slice with particular options.
- *SliceMsg* is the type of the message that is broadcast in order to synchronize the views after the slice has changed. *msg.slice* refers to the changed slice, *msg.op* indicates the performed operation (see explanation of *Slicer.Notifier*).

5.4 Model-View-Controller Concept

The Oberon Slicing Tool allows the user to display multiple views of the same slice by separating the model from the views. The model-view-controller concept has been introduced with Smalltalk. Burbeck [Bur92] describes it as follows:

> In the MVC concept the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task. The *view* manages the graphical and/or textual output to the display. The *controller* interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. Finally, the *model* manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

The model is represented by an object of type *Slicer.Slice*. The view and the controller are combined by an object of type *SlicerFE.Frame*. At the time of writing this thesis, two kinds of views have been implemented, the standard view and the call hierarchy view. Additional views can be implemented. They are kept consistent by broadcasting the *SliceMsg* into the "viewer space". Each viewer that displays the changed slice reacts to the indicated operation properly, e.g. by updating its view. An example shall demonstrate this: Assume

that two standard viewers (*V1* and *V2*) and one call hierarchy viewer (*V3*) display the same slice. The user clicks on a statement in *V1*, upon which the view orders the model to compute the slice for the statement. After having computed the slice, the model broadcasts the message *SlicerFE.SliceMsg* to all visible viewers. Viewers *V1*, *V2*, and *V3* react by updating their view. This is very similar to the way that multiple views of the same text are kept consistent in the Oberon System.

6 Comparison

In this chapter we describe several program slicing tools. Some of them are available freely. All of them are written for the programming language C. None of them can handle object-oriented programs. Most of them have difficulty with dynamic binding, aliases, structured data types, and side-effects of function calls. Most of them handle access to and definition of heap-allocated data extremely conservative.

In the literature, some graph representations have been proposed for slicing object-oriented programs ([Kri94] and [LaH96]). Chen et al. [ChWC97] addressed the problems of slicing object-oriented programs which make use of abstraction, encapsulation, inheritance, and polymorphism. Static slicing of Java programs has been described in [KoMG97]. Several people work on program slicing of object-oriented programs, but to the best of our knowledge, no program slicers for object-oriented programs are yet available.

6.1 Chopshop

Chopshop [Chop] is a reverse engineering tool to help programmers understand unfamiliar C code. Its data flow analysis technique is a modular generalization of static program slicing. The user can select several sources and sinks of information, and Chopshop shows how data flows from the sources to the sinks. Chopshop accepts full ANSI C and generates program slices in textual and pictorial form. The tool does alias analysis and handles all language features of ANSI C. It provides data flow abstractions for procedures and allows the user to zoom-in and zoom-out by applying different abstraction mechanisms to the pictures.

We could not evaluate the Chopshop ourselves since it is not freely available. The description is based on the technical papers and on the Web pages of Chopshop.

6.2 Ghinsu

Ghinsu is an environment [Ghin] that integrates a number of tools that aid the programmer in a number of software engineering activities, primarily maintenance. Ghinsu supports static and dynamic forward and backward slicing as well as dicing of programs written in a large subset of C (including pointers) or in Pascal. It is possible to list all uses of a definition and to list all definitions reaching a use. Dead code (i.e. code that is unreachable)

can be highlighted.

We could not evaluate the Ghinsu environment ourselves since it is not freely available. The description in this thesis is based on the technical papers and on the Web pages of Ghinsu.

6.3 Spyder

Spyder [Spyd] is a debugging tool that has been prototyped by Hiralal Agrawal [Agr91] at Purdue University. It can compute interprocedural backward slices that are either static or dynamic. It implements slicing via graph reachability over dependence graphs. For dynamic slicing, instrumented code is compiled to an executable program that is executed while recording the input and output. This information is then used by the debugger to execute the program and generate a dynamic dependence graph that is finally traversed. The tool handles almost all of ANSI C, except functions with return values in general and function pointers in particular which is a big limitation. Arrays are treated as scalars, the indices are ignored.

An alpha release of Spyder is available for SunOS 4.0 and X11 Release 5 via the Web pages of Spyder [Spyd]. However, the description in this thesis is based on the technical papers and on the Web pages of Spyder.

6.4 Unravel

Unravel [Unra] is a prototype program slicing tool that can be used to statically evaluate ANSI C source code. It has been developed at the National Institute of Standards and Technology as part of a research project [Ly+95]. There are a few limitations that decrease the precision of the computed slices:

- Unions are not handled.
- Forks are not handled.
- Pointers to functions are not handled.
- Library functions are handled as follows: If a value is passed, it is assumed to be referenced, if an address is passed, then the referenced object is assumed to be changed. Nothing is assumed about global variables.

The tool is divided into three parts: a source code analysis component, a link component that links flow information from separate source files together and an interactive slicing component.

A prototype implementation is available for SunOS 4.1 and X11 Release 5 via the Web pages of Unravel. However, the description in this thesis is based on the technical papers and on the Web pages of Unravel.

6.5 VALSOFT

VALSOFT (Validating software controlled measuring systems by Program Slicing and Constraint Solving) [VALS] analyzes the data flow of a program and calculates exact conditions under which a suspect data flow can take place. The tool handles ANSI C programs. It can derive forward and backward slices, as well as chops. The nodes of the dependence graphs can consist of parts of statements in order to account for the side-effects of function calls. There are a few limitations that decrease the precision of the computed slices:

- Only flow-insensitive alias analysis is performed for pointers.
- Only structured uses of switch statements are supported.
- Gotos are not supported.
- Unions are handled in the same way as structs.

The system dependence graph can be written to a file for further use. The program slicer can also be used as a slicing server for clients such as a graph visualizer.

We could not evaluate VALSOFT ourselves since it is not freely available. The description in this thesis is based on the technical papers and on the Web pages of VALSOFT.

6.6 Wisconsin Program-Slicing Project

The Wisconsin Program-Slicing Tool [WIPS] supports operations on C programs, including backward slicing, forward slicing, and chopping. Furthermore, it can show control and/or flow dependences between program components and the set of global variables that might be modified by a procedure. There are a few limitations that decrease the precision of the computed slices:

- Functions that are not found in the program or library functions are handled as follows: Under the optimistic assumptions that these functions do not use and modify global variables and heap-allocated data, all possible summary edges are inserted between parameter nodes. If a library function uses or modifies global data and heap-allocated data, these effects are not handled properly.
- A call via a procedure variable is considered to be a possible call of every function whose address is taken somewhere in the program (i.e., it is assigned somewhere to a procedure variable).
- Every pointer dereference is considered to be a possible access of every piece of heap-allocated storage, as well as a possible access of every variable to which the address-of operator is applied somewhere in the program.
- Fields of a structured variable are not distinguished from the variable itself.
- Arrays are treated as a whole, access to array elements with constant indices are not handled specially.

The used algorithm can be outlined as follows: In a first pass, the call graph is built gathering information about global variables and dynamically bound calls. A second pass builds a control flow graph for each procedure. The program's procedure dependence graphs are then constructed from their corresponding control flow graphs. The system dependence graph is constructed by linking together all of the program's procedure dependence graphs with call-graph information. Additional information that summarizes transitive dependences due to procedure calls is finally added to the system dependence graph which can be written to a file for later use.

The Wisconsin Program-Slicing Project can be licensed from the University of Wisconsin under a distribution fee of US$500 for non-profit research purposes. We did not license it. The description in this thesis is based on the technical papers and on the Web pages. The Wisconsin Program-Slicing Project is being integrated into the tools of GrammaTech [Gram], a company working on innovative software development tools.

7 Conclusions

To use Oberon-2 as a source and target language helped us on the one hand (because we already had a parser, a compiler and other tools for that language), but made it difficult on the other hand (e.g., due to reference parameters, dynamic binding, and modules). The Oberon System was an excellent working tool and an appropriate base for the work presented in this thesis.

We have implemented the Oberon Slicing Tool, a fully operational program slicing tool for Oberon-2, with the following features:

- It computes static 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 and procedure variables; and the program can consist of several modules.
- It uses a fine-grained program representation: the abstract syntax tree and the symbol table constructed by the front end of the Oberon-2 compiler, enriched with slicing information. The targets of the control and data dependences are the nodes of the abstract syntax tree. The entities that are considered for inclusion into the slice are nodes of the abstract syntax tree, rather than whole statements. This improves the precision of the slices significantly.
- Data flow information is computed precisely, taking into account side-effects of functions and short-circuit evaluation of Boolean expressions. Definition of array elements and record fields are handled as precisely as possible.
- It computes intraprocedural, interprocedural and intermodular slices. It uses a repository to store the computed slicing information which can be re-used later when importing already sliced modules. Thereby, precise knowledge about the parameter usage of imported functions is available.
- It handles procedural and object-oriented programs. The key concepts of object-oriented programming, such as inheritance, polymorphism and dynamic binding, as well as abstraction, information hiding, and encapsulation of data and code are handled in a natural way. Inheritance, polymorphism and dynamic binding are considered during control flow and data flow analysis, as well as during alias analysis.
- It restricts the sets of possible aliases at definitions by exploiting the fact that Oberon-2 is strongly typed and by exploiting knowledge about the place of the declaration of the possibly aliasing variables. Additionally the sets of possible aliases can be restricted by user feedback. The restricted sets of possible aliases are then used to compute more precise slicing information.
- It handles dynamic binding due to type-bound procedures and procedure variables by computing all call destinations for all call sites. The user may restrict the set of possible call destinations by user feedback. The restricted sets of possible call destinations are then used to compute more precise slicing information.
- We narrowed the gap between static and dynamic analysis by starting from conservative assumptions, having the user restrict the effects of aliases and dynamic binding and recompute more precise slicing information. The cycle of computing slicing information, slicing the program, and restricting the effects of aliases and dynamic binding may continue several times, in order to tailor the slices to the use case that the programmer has in mind. The user-feedback can be recorded and be played back later to customize the slice without user interaction.
- The computation of slicing information is very fast: Slicing information is computed within a few seconds, slices are computed without perceptible delays.
- The front end of the Oberon Slicing Tool uses active text elements to visualize the computed control flow and data flow information: bidirectional hypertext links connect the call sites and the called procedures, parameter information elements indicate how the parameters are used at call sites, the sets of possible aliases and possible call destinations are represented by popup elements that process user-feedback and forward it to the slicing algorithm. The Oberon Slicing Tool implements the Model-View-Controller concept, i.e. it is possible to have multiple views of the same slice that are kept consistent.

Among these features, the following constitute contributions to the state-of-the-art:

- Combination of state-of-the-art algorithms (slicing as a graph-reachability problem [HoRB90], computation of summary edges [LivC94, LivJ95]) for a strongly-typed programming language.
- Natural and efficient support for object-oriented features.
- Uniform handling of dynamic binding due to method calls and due to calls of procedure variables.
- Precise computation of control flow and data flow information based on the abstract syntax tree, taking into account side-effects of functions and short-circuit evaluation of Boolean expressions. Definition of array elements and record fields are handled as precisely as possible.
- Fast and efficient alias analysis taking into account type information and information about the place of the declaration.
- Intermodular slicing with a repository to store the computed slicing information for later reuse.
- Active text elements for the visualization of control and data flow information.
- User feedback via active text elements in order to restrict the sets of possible aliases at definitions and the sets of possible call destinations at polymorphic call sites, thus bridging the gap between static analysis and dynamic analysis.

The Oberon Slicing Tool is freely available under the conditions of the Oberon Slicing Tool License via the URL http://www.ssw.uni-linz.ac.at/Staff/CS/Slicing.html

8 Future Work

Since many variants of program slicing have been proposed for different applications, our work could be extended into many different directions. In the following we will only briefly discuss how the Oberon Slicing Tool could be better integrated into the programming environment. Furthermore we discuss that other variants of program slicing could be implemented based on our graph representation and how the derived information could be used for software metrics.

8.1 Integration into the Programming Environment

At the time of writing this thesis, the Oberon Slicing Tool visualizes the slices in its own window. The source code is reconstructed from the abstract syntax tree and the symbol table. There are several advantages of this approach:

- The source code is presented in a canonical form. Each line contains at most one statement.
- Additional information can be inserted right away during the reconstruction of the source code.

However, it also has some disadvantages:

- The layout of the original source code is lost.
- The front-end of the compiler skips all comments, so they are lost and cannot be displayed.
- The front-end of the compiler performs some simple optimizations such as constant folding, transformation of IF statements with constant conditions, replacement of integer multiplication by a power of two by arithmetic shifts, etc. These optimizations cannot be undone, the results are presented to the user. This may give insights, but may also confuse.
- The reconstruction of the source code is difficult, the module implementing the reconstruction and the user interface is very big (approximately 3000 lines).

Another problem is that the slicing window cannot be used to edit the program since edit operations would probably invalidate the computed information. A simple approach would be to remove all visual elements at the first insert or delete operation performed on the text. A more sophisticated approach would be to partially invalidate and remove the visual elements and to recompute the information for the invalidated parts in the background. Currently slicing information is computed for one module at a time. It might be advantageous to keep the information more or less up-to-date during editing. This could be done by performing control and data flow analysis on a per procedure basis. Additional conservative assumptions would be necessary, but while editing a procedure, intraprocedural information might be sufficient. During longer phases without edit operations, the analysis could be performed for the whole module. Therefore, the error handling and recovery capabilities of the Oberon compiler would have to be enhanced since the compiler would have to derive approximate syntax trees for incomplete or erroneous programs.

The Oberon Slicing Tool could also be integrated with the compiler which could benefit from the slicing information. Information about reaching definitions and aliases could be used to generate faster code.

The Oberon Slicing Tool could also be integrated with the Oberon interpreter [Obi98] and the debugger. Since the interpreter also uses the abstract syntax tree and the symbol table as its internal data structures, integration might be possible with reasonable effort. Interpretation of Boolean expressions might help to determine feasible paths. During debugging, information about the reaching definitions could be very useful. Additionally run-time information could be used to perform more precise data flow analysis. Gupta et al. [GuSH97] introduced *hybrid slicing*. They integrate dynamic information from a specific execution into a static analysis. They use breakpoint information and the dynamic call graph to more accurately estimate potential paths taken by the program. Their ideas could possibly be integrated into the Oberon Slicing Tool.

8.2 Other Variants of Slicing

The Oberon Slicing Tool can only compute static backward slices. This is not a big limitation for the envisaged fields of application. However, the graph representation of the program used for backward slicing has also been used by many researchers to implement other variants of slicing: Horwitz et al. [HoRB90] adapted the algorithms of backward slicing to forward slicing, Agrawal and Horgan [AgH90] presented algorithms for dynamic slicing based on dependence graphs.

8.3 Software Metrics

Since the Oberon Slicing Tool computes precise control flow and data flow information, it could be used to derive structural metrics based on

- the number of call destinations at dynamically bound calls
- the number of ordinary and additional parameters
- the number of aliases per definition
- the number of possible dynamic types per polymorphic variable
- the length of recursion chains due to static and dynamic binding
- the number of reaching definitions per usage
- the number of side-effects of procedures and functions

The resulting metrics could be tested against object-oriented and procedural programs, investigating the differences.

Appendix: Additional Module Definitions

Module MeasuringSlicer

```
DEFINITION MeasuringSlicer;

   IMPORT Slicer, SlicerOPT, SlicerOPS;

   TYPE
      Slice = POINTER TO SliceDesc;
      SliceDesc = RECORD (Slicer.SliceDesc)
         PROCEDURE (s: Slice) BuildClassHierarchy;
         PROCEDURE (s: Slice) Compile (mod: ARRAY OF CHAR; VAR done: BOOLEAN);
         PROCEDURE (s: Slice) CompleteComputation;
         PROCEDURE (s: Slice) ControlFlow;
         PROCEDURE (s: Slice) DataFlow;
         PROCEDURE (s: Slice) MayAlias (o1, o2, proc: SlicerOPT.Object): BOOLEAN;
         PROCEDURE (s: Slice) SliceProc (node: SlicerOPT.Node; interprocedural: BOOLEAN);
         PROCEDURE (s: Slice) SliceProcForObj (proc: SlicerOPT.Node; obj: SlicerOPT.Object);
         PROCEDURE (s: Slice) SliceStat (node: SlicerOPT.Node; interprocedural: BOOLEAN);
         PROCEDURE (s: Slice) Statistics;
      END ;

   PROCEDURE InitSlice (s: Slice);
   PROCEDURE InstallMeasuringSlicer;
   PROCEDURE SliceFactoryMethod (): Slicer.Slice;

END MeasuringSlicer.
```

Module SlicerFE

```
DEFINITION SlicerFE;

   IMPORT TextFrames, Display, Texts, Slicer, PopupElems, SlicerOPT, SlicerOPS;

   CONST
      unexpectedSituation = 99;
      version = "Oberon Slicing Tool V1.0 (CS)";
      (* options *)
      withDDElems = 1;
      withParInfoElems = 2; withActualParElems = 3;
      withCallingElems = 4; withCalledAtElems = 5;
      withDynTypeElems = 6; withAliasElems = 7;
      withParameterSummary = 8; withReachingEnd = 9;
```



```
      withPosition = 10; interprocedural = 11;
      defaultOptions = {withDDElems, withParInfoElems, withActualParElems,
         withCallingElems, withCalledAtElems, withDynTypeElems, withAliasElems,
         withPosition, interprocedural};
      (* id of SelectMessage *)
      on = 1; off = 2; toggle = 3;

   TYPE
      AliasElem = POINTER TO RECORD (PopupElems.ElemDesc)
      END ;
      DynTypeElem = POINTER TO RECORD (PopupElems.ElemDesc)
      END ;
      Frame = POINTER TO FrameDesc;
      FrameDesc = RECORD (TextFrames.FrameDesc)
         slice: Slicer.Slice;
         options: SET;
      END ;
      SelectMessage = RECORD (Texts.ElemMsg)
         frame: Frame;
         id: INTEGER;
         name: ARRAY 128 OF CHAR;
      END ;
      SliceMsg = RECORD (Display.FrameMsg)
         slice: Slicer.Slice;
         op: INTEGER;
      END ;

   VAR
      forcePersistence: BOOLEAN;
      recording: BOOLEAN;

   PROCEDURE AllocPopup;
   PROCEDURE ControlFlow;
   PROCEDURE DataFlow;
   PROCEDURE FindNode;
   PROCEDURE FindProc;
   PROCEDURE Flush;
   PROCEDURE Handle (f: Display.Frame; VAR msg: Display.FrameMsg);
   PROCEDURE InspectSlice;
   PROCEDURE MakePersistent;
   PROCEDURE NewFrame (slice: Slicer.Slice; T: Texts.Text; pos: LONGINT): Frame;
   PROCEDURE NotifyDisplay (s: Slicer.Slice; op: INTEGER);
   PROCEDURE Open;
   PROCEDURE OpenCallHierarchyViewer;
   PROCEDURE Playback;
   PROCEDURE ReconstructSource;
   PROCEDURE ResetDataFlowInfo;
   PROCEDURE SetAliases;
   PROCEDURE SetArrayExpansionLimit;
   PROCEDURE SetDynamicTypes;
   PROCEDURE SetForcePersistence;
   PROCEDURE SetMark;
   PROCEDURE SetOption;
   PROCEDURE SetRecording;
   PROCEDURE ShowOptions;
   PROCEDURE ShowPosition;
   PROCEDURE Statistics;

END SlicerFE.
```


Module ParInfoElems

```
DEFINITION ParInfoElems;

   IMPORT Texts, Slicer, SlicerOPT, SlicerOPS;

   CONST
      version = "Oberon Slicing Tool V1.0 (CS)";

   TYPE
      ActualParElem = POINTER TO ActualParElemDesc;
      ActualParElemDesc = RECORD (Texts.ElemDesc) END ;
      Elem = POINTER TO ElemDesc;
      ElemDesc = RECORD (Texts.ElemDesc)
      END ;

   PROCEDURE Alloc;
   PROCEDURE AllocActualPar;
   PROCEDURE Handle (e: Texts.Elem; VAR msg: Texts.ElemMsg);
   PROCEDURE HandleActualPar (e: Texts.Elem; VAR msg: Texts.ElemMsg);
   PROCEDURE NewActualParElem (varpar, used, defined: BOOLEAN): ActualParElem;
   PROCEDURE NewElem (slice: Slicer.Slice; proc: SlicerOPT.Object; parIn: SlicerOPT.Node;
      sel, abstract: BOOLEAN; col: INTEGER): Elem;
   PROCEDURE Track (e: Elem; VAR msg: Texts.ElemMsg);

END ParInfoElems.
```

Module SlicerOPT

```
DEFINITION SlicerOPT;

   IMPORT SlicerOPS;

   CONST
      (* kinds of parameter usages *)
      parIn = 0; parOut = 1; parUsed = 2; parDefined = 3; parAlwaysDefined = 4; isPar = 5;
      (* kinds of dependences *)
      (* parIn, parOut *) CD = 2; DD = 3; transDD = 4; call = 5; dynCall = 6;
      (* flags for hash table entries *)
      empty = 0; filled = 1; deleted = 2; mustAssign = 3; defOfAlias = 4;
      (* flags for ProcInfo *)
      useDefComputed = 0; solved = 1;
      (* nodes classes *)
      Nfpar = 29; NcallSite = 30; Ndyncall = 31; NloopExit = 32; NprocExit = 33;
      Nhalt = 34;
      (* nodes subclasses *)
      (* Nfpar *)
      inPar = 0; outPar = 1; additionalInPar = 2; additionalOutPar = 3;
      (* Nvarpar *)
      additionalPar = 1;
      (* Ncall *)
      normal = 0; statMeth = 1; superCall = 2;
      dynMethAllKnown = 3; dynMethNotAllKnown = 4;
      procVarAllKnown = 5; procVarNotAllKnown = 6;
```



```
      MaxConstLen = 256;
      unexpectedSituation = 99;

   TYPE
      Access = POINTER TO AccessDesc;
      AccessDesc = RECORD
         obj: Object;
         node: Node;
      END ;
      AccessArr = POINTER TO ARRAY OF Access;
      AccessIterator = RECORD
         PROCEDURE (VAR it: AccessIterator) First (): Access;
         PROCEDURE (VAR it: AccessIterator) Next (): Access;
      END ;
      AliasIterator = RECORD
         PROCEDURE (VAR it: AliasIterator) First (): Object;
         PROCEDURE (VAR it: AliasIterator) GetSelection (): BOOLEAN;
         PROCEDURE (VAR it: AliasIterator) Next (): Object;
         PROCEDURE (VAR it: AliasIterator) NextSelected (): Object;
         PROCEDURE (VAR it: AliasIterator) SetSelection (sel: BOOLEAN);
      END ;
      CallIterator = RECORD
         end-: BOOLEAN;
         PROCEDURE (VAR it: CallIterator) AdditionalAPars (): Node;
         PROCEDURE (VAR it: CallIterator) Advance;
         PROCEDURE (VAR it: CallIterator) CallNode (): Node;
         PROCEDURE (VAR it: CallIterator) ProcObj (): Object;
         PROCEDURE (VAR it: CallIterator) Reset;
      END ;
      ChoiceIterator = RECORD
         node-: Node;
         PROCEDURE (VAR it: ChoiceIterator) First (): Node;
         PROCEDURE (VAR it: ChoiceIterator) GetSelection (): BOOLEAN;
         PROCEDURE (VAR it: ChoiceIterator) Next (): Node;
         PROCEDURE (VAR it: ChoiceIterator) NextSelected (): Node;
         PROCEDURE (VAR it: ChoiceIterator) SetSelection (sel: BOOLEAN);
      END ;
      Const = POINTER TO ConstDesc;
      ConstDesc = RECORD
         ext: ConstExt;
         intval, intval2: LONGINT;
         setval: SET;
         realval: LONGREAL;
         id: LONGINT;
      END ;
      ConstExt = POINTER TO SlicerOPS.String;
      Definitions = POINTER TO ARRAY OF VarDef;
      Dependences = POINTER TO DependencesDesc;
      DependencesDesc = RECORD
         cds, dds, parIns, parOuts, transdds, dyncalls: NodeArr;
      END ;
      HashTable = RECORD
         size, count: LONGINT;
         PROCEDURE (VAR h: HashTable) Found (o: Object; n: Node;
            VAR pos: LONGINT): BOOLEAN;
         PROCEDURE (VAR h: HashTable) Free;
         PROCEDURE (VAR h: HashTable) Init (size: LONGINT);
         PROCEDURE (VAR h: HashTable) Insert (o: Object; n: Node; flags: SET);
         PROCEDURE (VAR h: HashTable) Reset;
```

```
      PROCEDURE (VAR ht: HashTable) SetIterator (VAR it: HashTableIterator);
   END ;
   HashTableIterator = RECORD
      end-: BOOLEAN;
      pos-: LONGINT;
      PROCEDURE (VAR it: HashTableIterator) CurFlags (): SET;
      PROCEDURE (VAR it: HashTableIterator) CurMustAssign (): BOOLEAN;
      PROCEDURE (VAR it: HashTableIterator) CurNode (): Node;
      PROCEDURE (VAR it: HashTableIterator) CurObj (): Object;
      PROCEDURE (VAR it: HashTableIterator) Next;
      PROCEDURE (VAR it: HashTableIterator) SetPos (pos: LONGINT);
   END ;
   Node = POINTER TO NodeDesc;
   NodeDesc = RECORD
      left, right, link: Node;
      class, subcl: SHORTINT;
      readonly: BOOLEAN;
      mark: SHORTINT;
      typ: Struct;
      obj: Object;
      conval: Const;
      procInfo: ProcInfo;
      usedObjs, definedObjs: ObjArr;
      dependences: Dependences;
      gen, kill, in, choice: SetArr;
      aliases: ObjArr;
      enabledAliases: SetArr;
      id: LONGINT;
      PROCEDURE (node: Node) SetAliasIterator (VAR it: AliasIterator);
      PROCEDURE (node: Node) SetChoiceIterator (VAR it: ChoiceIterator);
      PROCEDURE (node: Node) SetDependenceIterator (kind: SHORTINT;
         VAR it: NodeIterator);
   END ;
   NodeArr = POINTER TO ARRAY OF Node;
   NodeIterator = RECORD
      PROCEDURE (VAR it: NodeIterator) First (): Node;
      PROCEDURE (VAR it: NodeIterator) Next (): Node;
   END ;
   Nodes = POINTER TO NodesDesc;
   NodesDesc = RECORD
      using, defining: NodeArr;
      nofUsing, nofDefining: INTEGER;
      PROCEDURE (nodes: Nodes) SetNodeIterator (using: BOOLEAN; VAR it: NodeIterator);
   END ;
   ObjArr = POINTER TO ARRAY OF Object;
   Object = POINTER TO ObjDesc;
   ObjDesc = RECORD
      left, right, link, scope: Object;
      name: SlicerOPS.Name;
      leaf: BOOLEAN;
      mode, mnolev, vis: SHORTINT;
      typ: Struct;
      conval: Const;
      adr, linkadr: LONGINT;
      nodes: Nodes;
      procInfo: ProcInfo;
      assignedToProcVar: BOOLEAN;
      mark, level: SHORTINT;
      mod: Object;
```



```
      expanded: BOOLEAN;
      components: ObjArr;
      containedIn: Object;
      id: LONGINT;
   END ;
   ObjectIterator = RECORD
      PROCEDURE (VAR it: ObjectIterator) First (): Object;
      PROCEDURE (VAR it: ObjectIterator) IsIn (obj: Object): BOOLEAN;
      PROCEDURE (VAR it: ObjectIterator) Next (): Object;
   END ;
   Objects = POINTER TO ObjectsDesc;
   ObjectsDesc = RECORD
      used, defined: ObjArr;
      nofUsed, nofDefined: INTEGER;
      PROCEDURE (objs: Objects) SetObjectIterator (used: BOOLEAN; VAR it: ObjectIterator);
   END ;
   ProcInfo = POINTER TO ProcInfoDesc;
   ProcInfoDesc = RECORD
      fpars, callSites: Node;
      calls: NodeArr;
      accesses: AccessArr;
      procExit, enter: Node;
      procObj: Object;
      in, out: SetArr;
      objs: Objects;
      definitionsHT: HashTable;
      varDefs: Definitions;
      flags: SET;
      id: LONGINT;
      PROCEDURE (procInfo: ProcInfo) AddCall (call, callee: Node; VAR callSite: Node);
      PROCEDURE (procInfo: ProcInfo) AddFPar (node: Node);
      PROCEDURE (procInfo: ProcInfo) InsertObj (obj: Object; used: BOOLEAN);
      PROCEDURE (procInfo: ProcInfo) RegisterAccess (obj: Object; node: Node);
      PROCEDURE (procInfo: ProcInfo) SetAccessIterator (VAR it: AccessIterator);
      PROCEDURE (procInfo: ProcInfo) SetCallIterator (VAR it: NodeIterator);
   END ;
   SetArr = POINTER TO ARRAY OF SET;
   Struct = POINTER TO StrDesc;
   StrDesc = RECORD
      form, comp, mno, extlev: SHORTINT;
      ref, sysflag: INTEGER;
      n, size, tdadr, offset, txtpos: LONGINT;
      BaseTyp: Struct;
      link, strobj, mod: Object;
      extensions: StructArr;
      fields: ObjArr;
      id: LONGINT;
      mark: SHORTINT;
   END ;
   StructArr = POINTER TO ARRAY OF Struct;
   StructIterator = RECORD
      PROCEDURE (VAR it: StructIterator) First (): Struct;
      PROCEDURE (VAR it: StructIterator) Next (): Struct;
   END ;
   VarDef = RECORD
      mustAssign-, mayAssign-: SetArr;
   END ;
   WorkProcObject = PROCEDURE (o: Object);
```



```
VAR
   GlbMod: ARRAY 31 OF Object;
   ModFromRepository: PROCEDURE (name: ARRAY OF CHAR; key: LONGINT): Object;
   SYSimported: BOOLEAN;
   booltyp: Struct;
   bytetyp: Struct;
   chartyp: Struct;
   currentModule: Object;
   forwards: ObjArr;
   inttyp: Struct;
   linttyp: Struct;
   lrltyp: Struct;
   niltyp: Struct;
   nofForwards: LONGINT;
   nofGmod: SHORTINT;
   notyp: Struct;
   realtyp: Struct;
   settyp: Struct;
   sinttyp: Struct;
   stringtyp: Struct;
   syslink: Object;
   sysptrtyp: Struct;
   topScope: Object;
   undftyp: Struct;
   universe: Object;

PROCEDURE AppendNode (VAR head: Node; node: Node);
PROCEDURE Close;
PROCEDURE CloseScope;
PROCEDURE ExistsDependence (from, to: Node; kind: SHORTINT): BOOLEAN;
PROCEDURE Export (VAR modName: SlicerOPS.Name; VAR newSF: BOOLEAN;
   VAR key: LONGINT);
PROCEDURE Find (VAR res: Object);
PROCEDURE FindField (VAR name: SlicerOPS.Name; typ: Struct; VAR res: Object);
PROCEDURE FindImport (mod: Object; VAR res: Object);
PROCEDURE FindMethod (name: ARRAY OF CHAR; typ: Struct): Object;
PROCEDURE FindOverriddenMethod (name: ARRAY OF CHAR; typ: Struct): Object;
PROCEDURE FirstNode (n: Node; class: SHORTINT; subclasses: SET): Node;
PROCEDURE GetAliasForRealName (realName: ARRAY OF CHAR;
   VAR alias: ARRAY OF CHAR);
PROCEDURE GetRealNameForAlias (alias: ARRAY OF CHAR;
   VAR realName: ARRAY OF CHAR);
PROCEDURE Import (VAR aliasName, impName, selfName: SlicerOPS.Name);
PROCEDURE IndexOfNode (nodeArr: NodeArr; node: Node): LONGINT;
PROCEDURE IndexOfObject (objArr: ObjArr; obj: Object): LONGINT;
PROCEDURE Init;
PROCEDURE Insert (VAR name: SlicerOPS.Name; VAR obj: Object);
PROCEDURE InsertAlias (at: Node; alias: Object);
PROCEDURE InsertDependence (from, to: Node; kind: SHORTINT);
PROCEDURE InsertDyncall (from, to: Node);
PROCEDURE InsertFwdDecl (proc: Object);
PROCEDURE InsertNew (VAR varDefs: Definitions; obj: Object; size: LONGINT): LONGINT;
PROCEDURE InsertNode (VAR nodeArr: NodeArr; node: Node);
PROCEDURE InsertObject (VAR objArr: ObjArr; obj: Object);
PROCEDURE InsertStruct (VAR arr: StructArr; str: Struct);
PROCEDURE InsertUseDef (node: Node; obj: Object; used: BOOLEAN);
PROCEDURE IsExtended (typ: Struct): BOOLEAN;
PROCEDURE IsGlobal (obj: Object): BOOLEAN;
PROCEDURE IsIntermediate (obj, proc: Object): BOOLEAN;
```



```
PROCEDURE IsLocal (obj, proc: Object): BOOLEAN;
PROCEDURE IsOverridden (typ: Struct; name: ARRAY OF CHAR): BOOLEAN;
PROCEDURE Lookup (scope: Object; name: ARRAY OF CHAR; VAR obj: Object);
PROCEDURE MatchingParameterLists (par1, par2: Object; ret1, ret2: Struct): BOOLEAN;
PROCEDURE NestingLevel (obj: Object): SHORTINT;
PROCEDURE NewConst (): Const;
PROCEDURE NewExt (): ConstExt;
PROCEDURE NewNode (class: SHORTINT): Node;
PROCEDURE NewObj (): Object;
PROCEDURE NewProcInfo (): ProcInfo;
PROCEDURE NewStr (form, comp: SHORTINT): Struct;
PROCEDURE OpenScope (level: SHORTINT; owner: Object);
PROCEDURE ProcObj (call: Node): Object;
PROCEDURE SameType (t1, t2: Struct): BOOLEAN;
PROCEDURE SetCallIterator (node: Node; VAR it: CallIterator);
PROCEDURE SetNodeIterator (nodes: NodeArr; VAR it: NodeIterator);
PROCEDURE SetObjectIterator (objs: ObjArr; VAR it: ObjectIterator);
PROCEDURE SetStructIterator (structs: StructArr; VAR it: StructIterator);
PROCEDURE SetsClear (s: SetArr);
PROCEDURE SetsCopy (s1, s2: SetArr);
PROCEDURE SetsDifference (s1, s2, s3: SetArr);
PROCEDURE SetsEmpty (s: SetArr): BOOLEAN;
PROCEDURE SetsEqual (s1, s2: SetArr): BOOLEAN;
PROCEDURE SetsExcl (s: SetArr; x: LONGINT);
PROCEDURE SetsFill (s: SetArr);
PROCEDURE SetsIn (s: SetArr; x: LONGINT): BOOLEAN;
PROCEDURE SetsIncl (s: SetArr; x: LONGINT);
PROCEDURE SetsIntersection (s1, s2, s3: SetArr);
PROCEDURE SetsNew (VAR s: SetArr; size: LONGINT);
PROCEDURE SetsPrint (s: SetArr; VAR ht: HashTable);
PROCEDURE SetsUnion (s1, s2, s3: SetArr);
PROCEDURE Statistics;
PROCEDURE ThisAdditionalPar (call: Node; obj: Object): Node;
PROCEDURE ThisFPar (procInfo: ProcInfo; obj: Object): Node;
PROCEDURE ThisFPar2 (procInfo: ProcInfo; kinds: SET; obj: Object): Node;
PROCEDURE ThisVar (name: ARRAY OF CHAR; proc: Node): Object;
PROCEDURE ThisVarDef (varDefs: Definitions; obj: Object): LONGINT;
PROCEDURE TraverseSymbolTable (scope: Object; proc: WorkProcObject);

END SlicerOPT.
```

Module SlicerAuxiliaries

```
DEFINITION SlicerAuxiliaries;

   IMPORT SlicerOPT, SlicerOPS;

   TYPE
      CSGNode = POINTER TO CSGNodeDesc;
      Fixups = POINTER TO FixupsDesc;
      FixupsDesc = RECORD
         n: INTEGER;
         arr: SlicerOPT.NodeArr;
         PROCEDURE (f: Fixups) Fixup (to: SlicerOPT.Node);
         PROCEDURE (f: Fixups) Init;
```

```
        PROCEDURE (f: Fixups) Insert (n: SlicerOPT.Node);
        PROCEDURE (f: Fixups) SetIterator (VAR it: FixupsIterator);
    END ;
    FixupsIterator = RECORD
        PROCEDURE (VAR it: FixupsIterator) First (): SlicerOPT.Node;
        PROCEDURE (VAR it: FixupsIterator) Next (): SlicerOPT.Node;
    END ;

  PROCEDURE FindProc (root: CSGNode; proc: SlicerOPT.Node): CSGNode;
  PROCEDURE InsertProc (VAR root: CSGNode; proc: SlicerOPT.Node);
  PROCEDURE RemoveProc (VAR root: CSGNode; proc: SlicerOPT.Node): CSGNode;
  PROCEDURE ShowECSG (root: CSGNode);
  PROCEDURE UpdateExistingProc (VAR root: CSGNode; caller, callee: SlicerOPT.Node);

END SlicerAuxiliaries.
```

Bibliography

[AgH90] H. Agrawal, J. Horgan: *Dynamic Program Slicing*.
Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 1990.

[Agr91] H. Agrawal: *Towards Automatic Debugging of Computer Programs*.
Ph.D. dissertation, Purdue University, West Lafayette, Indiana, 1991.

[ASU86] A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques, and Tools*.
Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Bac97] D. Bacon: *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*.
PhD thesis, University of California at Berkeley.

[BaH93] S. Bates, S. Horwitz: *Incremental Program Testing Using Program Dependence Graphs*.
In Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages, ACM, 1993.

[Be86] V. Berzins: *On Merging Software Extensions*.
Acta Informatica, 23(6), 1986.

[BiO94] J. Bieman, L. Ott: *Measuring Functional Cohesion*.
IEEE Transactions on Software Engineering, 20(8), August 1994.

[Bi95] D. Binkley: *Reducing the Cost of Regression Testing by Semantics Guided Test Case Selection*.
In IEEE International Conference on Software Maintenance, 1995.

[BiO94] J. M. Bieman, L. M. Ott: Measuring Functional Cohesion.
IEEE Transactions on Software Engineering, 20(8), August 1994.

[BiG96] D. Binkley, K. B. Gallagher: *Program Slicing*.
Advances in Computers, Volume 43, 1996.

[Bra95] M. Brandis: *Optimizing Compilers for Structured Programming Languages*.
Dissertation, ETH Zürich, 1995.

[BrMö94] M. Brandis, H. Mössenböck: *Single-Pass Generation of Static Single-Assignment Form for Structured Languages*.
ACM Transactions on Programming Languages and Systems, 16(6), November 1994.

[Bur92] S. Burbeck: *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller(MVC)*.
http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[ChWC97] J.-L. Chen, F.-J. Wang, Y.-L. Chen: *Slicing Object-Oriented Programs*.
Abstract submitted to Asia-Pacific Engineering Conference and International Computer Science Conference, December 1997.
http://www.computer.org/conferen/proceed/8271abs.htm

[Chop] *Chopshop Project*
http://www.cs.cmu.edu/afs/cs.cmu.edu/project/chopshop/pub/www/home.html

[Cre90] R. Crelier: *OP2: A Portable Oberon Compiler*.
Technical report 125, ETH Zürich, February 1990.

[Cre94] R. Crelier: *Separate Compilation and Module Extension*.
Dissertation, ETH Zürich, 1994.

[DGC94] J. Dean, D. Grove, C. Chambers: *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*.
In Proceedings of the Ninth European Conference on Object-Oriented Programming - ECOOP'95 (Aarhus, Denmark), Springer-Verlag, August 1995.

[EGH94] M. Emami, R. Ghiya, L. J. Hendren: *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*.
ACM Conference on Programming Language Design and Implementation, 1994.

[Ern94] M. Ernst: *Practical Fine-Grained Static Slicing of Optimized Code*.
Technical report MSR-TR-94-14, Microsoft Research.

[FeOW87] J. Ferrante, K. Ottenstein, J. Warren: *The Program Dependence Graph and its Use in Optimization*.
ACM Transactions on Programming Languages and Systems, 9(3), July 1987.

[FoB97] I. Forgacs, A. Bertolino: *Feasible Test Path Selection by Principal Slicing*.
In Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), LNCS 1301, Springer-Verlag, September 1997.

[Gal91] K. B. Gallagher: *Using Program Slicing to Eliminate the Need for Regression Testing*.
In Eighth International Conference on Testing Computer Software, May 1991.

[GaL92] K. B. Gallagher, J. R. Lyle: *Using Program Slicing in Software Maintenance*.
IEEE Transactions on Software Engineering, 17(8), August 1991.

[GaHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*.
Addison-Wesley, 1995.

[Ghin] *The Ghinsu Environment.*
http://www.cis.ufl.edu/~pel/Ghinsu/gghinsu.html

[Gram] *GrammaTech*
http://www.grammatech.com/

[GuHS96] R. Gupta, M. J. Harrold, M. L. Soffa: *Program Slicing-Based Regression Testing Techniques*.
Journal of Software Testing, Verification and Reliability, 6(2), June 1996.

[GuSH97] R. Gupta, M. L. Soffa, J. Howard: *Hybrid Slicing: Integrating Dynamic Information with Static Analysis.*
In ACM Transactions on Software Engineering and Methodology, 6(4), October 1997.

[HaD95] M. Harman, S. Danicic: *Using Program Slicing to Simplify Testing*.
Software Testing, Verification and Reliability, 5, September 1995.

[HoPR89] S. Horwitz, J. Prins, T. Reps: *Integrating Noninterfering Versions of Programs*.
ACM Transactions on Programming Languages and Systems, 11(3), July 1989.

[HoRB90] S. Horwitz, T. Reps, D. Binkley: *Interprocedural Slicing Using Dependence Graphs.*
ACM Transactions on Programming Languages and Systems, 12(1), 1990.

[HoR91] S. Horwitz, T. Reps: *Efficient Comparison of Program Slices*.
Acta Informatica, 1991.

[JaR94] D. Jackson, E. J. Rollins: *A New Model of Program Dependences for Reverse Engineering*.
In Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (New Orleans, LA), December 1994.

[Ka80] U. Kastens: *Ordered Attribute Grammars*.
Acta Informatica 13, 3, 1980.

[KoL88] B. Korel, J. Laski: *Dynamic Program Slicing*.
Information Processing Letters, 29(3), October 1988.

[KoMG97] G. Kovács, F. Magyar, T. Gyimóthy: *Static Slicing of Java Programs*.
In Proceedings of the Fifth Symposium on Programming Languages and Software Tools, Jyväskylä, Finland, 1997.

[Kri94] Anand Krishnaswamy: *Program Slicing: An Application of Object-oriented Program Dependency Graphs*.
Technical report, Department of Computer Science, Clemson University

[LaH96] L. Larsen, M. J. Harrold: *Slicing Object-Oriented Software*.
Proceedings of the 18th International Conference on Software Engineering, 1996.

[LeTa79] T. Lengauer, R. Tarjan: *A Fast Algorithm for Finding Dominators in a Flowgraph.*
In ACM Transactions on Programming Languages and Systems, 1(1), July 1979.

[LivC94] P. E. Livadas, S. Croll: *A New Algorithm for the Calculation of Transitive Dependences*.
Technical report, Computer and Information Sciences Department, University of Florida, 1994.

[LivJ95] P. E. Livadas, T. Johnson: *An Optimal Algorithm for the Construction of the System Dependence Graph.*
Technical report, Computer and Information Sciences Department, University of Florida, 1995.

[Ly+95] J. R. Lyle, D. R. Wallace, J. R. Graham, K. B. Gallagher, J. E. Poole, D. W. Binkley: *A CASE Tool to Evaluate Functional Diversity in High Integrity Software*.
U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995.

[LyW86] J. R. Lyle, M. D. Weiser: *Automatic Program Bug Location by Program Slicing*.
In Proceeding of the Second International Conference on Computers and Applications, Peking, China, June 1987.

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

[MöWi91] H. Mössenböck, N. Wirth: *The Programming Language Oberon-2.*
Structured Programming, 12(4), 1991.

[OST] *The Oberon Slicing Tool*.
http://www.ssw.uni-linz.ac.at/Staff/CS/Slicing.html

[Obi98] G. Obiltschnig: *An Object-Oriented Interpreter Framework for the Oberon-2 Programming Language*.
Diploma thesis, Johannes Kepler University Linz, 1998.

[OtO84] K. Ottenstein, L. Ottenstein: *The Program Dependence Graph in Software DevelopmentEnvironments*.
In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, May 1984.

[PaR93] H. D. Pande, B. G. Ryder: *Static type determination for C++*.
Technical report, LCSR-TR-197, Rutgers University, February 1993.

[Spyd] *Spyder*
http://www.cs.purdue.edu/homes/spaf/spyder.html

[Ste98a] C. Steindl: *Program Slicing (1), Data Structures and Computation of Control Flow Information.*
Technical Report 11, Institut für Praktische Informatik, University Linz, 1998.

[Ste98b] C. Steindl: *Program Slicing (2), Computation of Data Flow Information*.
Technical Report 12, Institut für Praktische Informatik, University Linz, 1998.

[Szy92] C. 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.

[Unra] *The Unravel Project.*
http://hissa.ncsl.nist.gov/~jimmy/unravel.html

[VALS] *VALSOFT.*
http://www.cs.tu-bs.de/softech/valsoft/

[Wei84] M. D. Weiser: *Program Slicing*.
IEEE Transactions on Software Engineering, 10, July 1984.

[WIPS] *The Wisconsin Program-Slicing Project.*
http://www.cs.wisc.edu/wpis/html/

[Yan90] W. Yang: *A New Algorithm for Semantics-Based Program Integration.*
Ph.D. dissertation, University of Wisconsin, Madison, 1990.

[YoC79] E. Yourdon, L. Constantine: *Structured Design.*
Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[ZhR94] S. Zhang, B. G. Ryder: *Complexity of Single Level Function Pointer Aliasing Analysis*.
Technical report, LCSR-TR-233, Rutgers University, July 1994.

Curriculum Vitae

Christoph Steindl

1972	Born in Waidhofen an der Thaya
1978-82	Primary School: Volksschule in Ottenschlag
1982-90	Secondary School: BG/BRG in Zwettl Graduation with first class honors
1990-95	Computer Science at the Johannes Kepler University Linz Graduation with first class honors
1990-97	Mechatronics at the Johannes Kepler University Linz Graduation with first class honors
1995-98	Employment as university assistant at the Department of Practical Computer Science, Johannes Kepler University Linz
1998	Military service
1998-99	University assistant