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
```

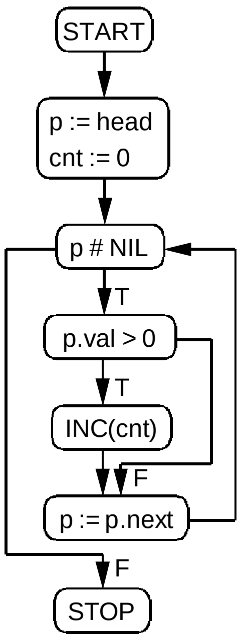


Fig. 2.1 - A piece of source code and its corresponding control flow graph

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* immediatelydominates *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.

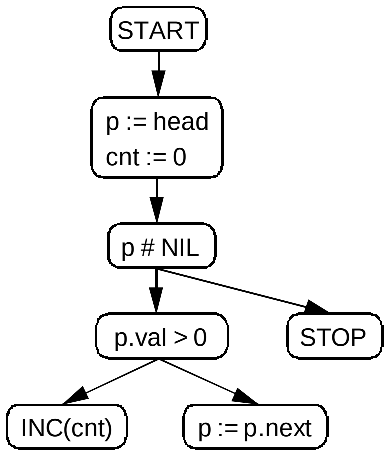


Fig. 2.2 - Dominator tree for the source code shown in Fig. 2.1

The dominator tree can be computed from the control flow graph. The algorithm due to Lengauer and Tarjan [LeTa79] runs in time O(N * α(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* immediatelypost-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.

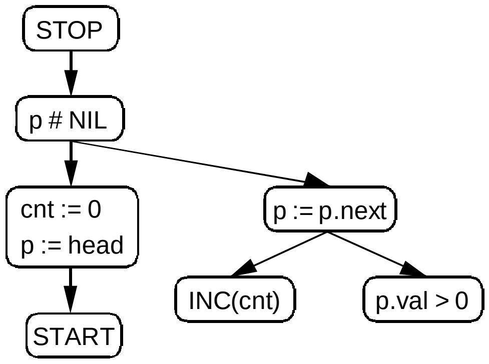


Fig. 2.3 - Post-dominator tree for the source code shown in Fig. 2.1

2.2.3 ControlDependences

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.

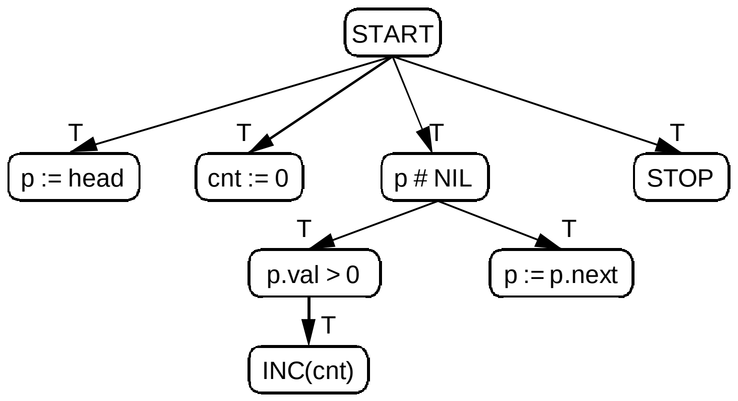


Fig. 2.4 - Control dependences for the source code shown in Fig. 2.1

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 DataDependences

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* definesvariable *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;
```

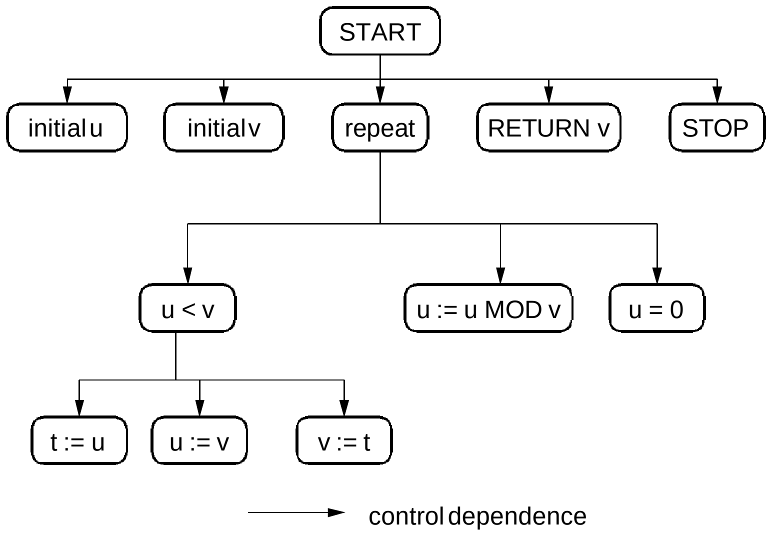


Fig. 2.5 - A program and its control dependence graph

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

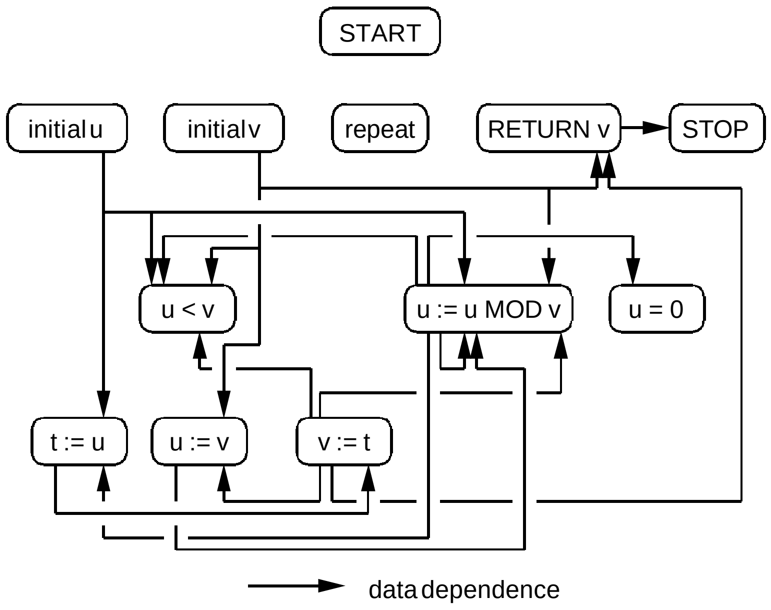


Fig. 2.6 - The data dependence graph for the program of Fig. 2.5

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

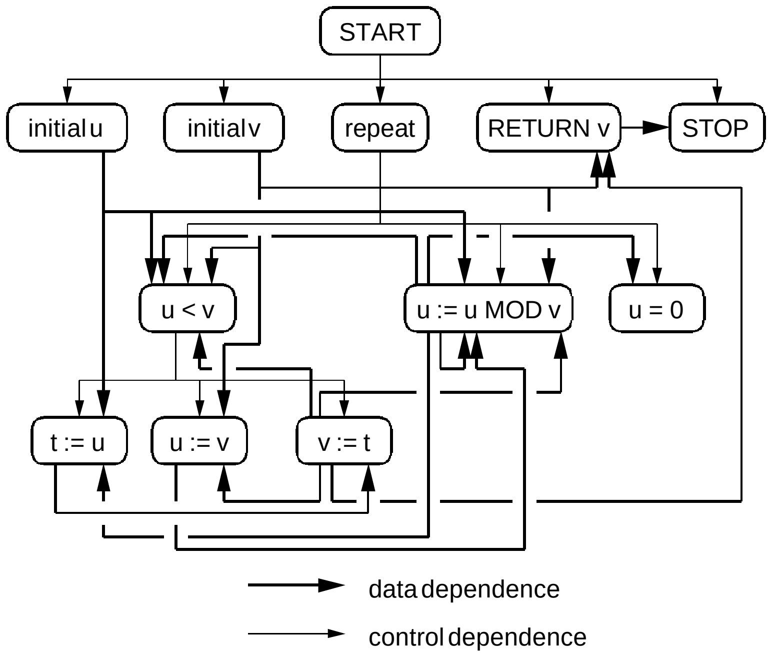


Fig. 2.7 - Program dependence graph for the program of Fig. 2.5

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:

o 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.

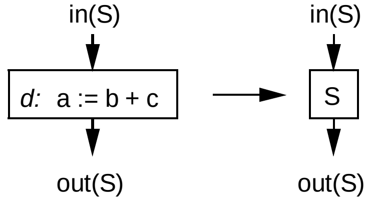
Algorithm for the computation of reaching definitions:

- 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.



gen(S) = {d}

kill(S) = DefinitionSet(a) - {d}

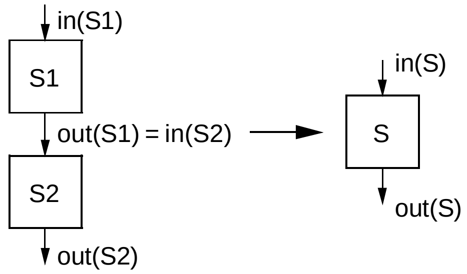
out(S) = gen(S) U (in(s) - kill(S))

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*.



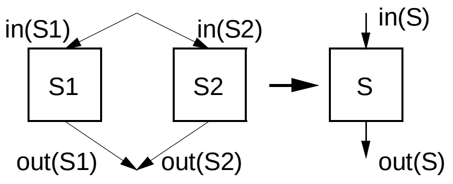
gen(S)	=	gen(S2) U (gen(S1) - kill(S2))
kill(S)	=	kill(S2) U (kill(S1) - gen(S2))
in(S1)	=	in(S)
in(S2)	=	out(S1)
out(S)	=	out(S2)

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*.



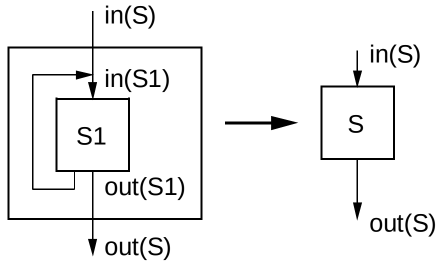
gen(S)	=	union of gen of all branches
kill(S)	=	intersection of kill of all branches
in of each branch	=	in(S)
out(S)	=	union of out of all branches

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).



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

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

  $out(S) = gen(S) \cup (in(S) - kill(S))$

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

```
            MODULE ComputeGenKill;                          DefinitionSets:
                                                              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;
(* 2: *)       t := 0;
            *)
            BEGIN
(* 3: *)       u := 10;
(* 4: *)       v := 2;
               IF u < v THEN
(* 5: *)           t := u;
(* 6: *)           u := v;
(* 7: *)           v := t
               END ;
(* 8: *)       u := u MOD v
            END ComputeGenKill.
```

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

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}	

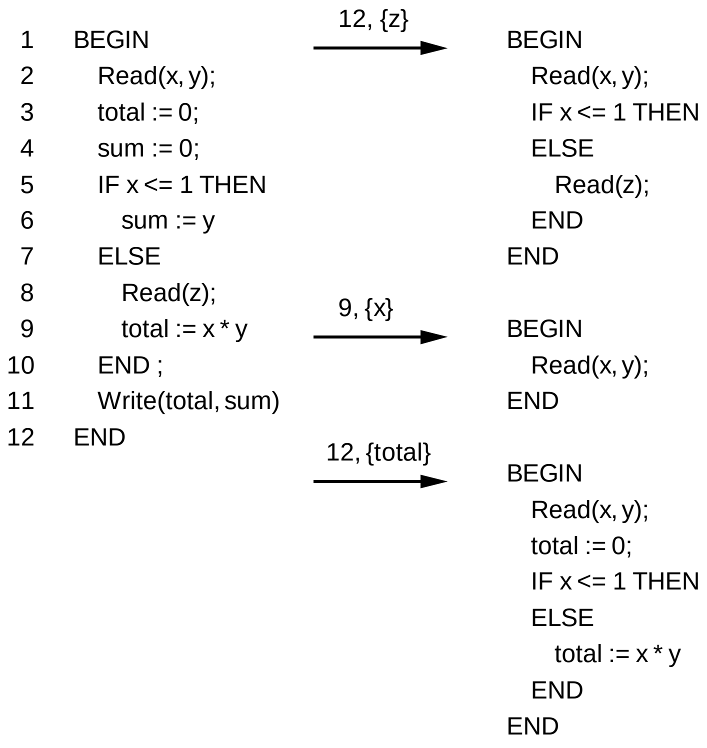
	out(4)	$= \text{gen}(4) \cup (\text{in}(4) - \text{kill}(4))$	$= \{4\} \cup (\{1..3\} - \{1, 7\})$	$= \{2..4\}$
IF:	in(IF)	$= \text{out}(4)$	$= \{2..4\}$	
			node 3 is a reaching definition for usage of *u*	
			node 4 is a reaching definition for usage of *v*	
	out(IF)	$= \text{gen}(IF) \cup (\text{in}(IF)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

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

```
BEGIN
  Read(x, y);
  IF x <= 1 THEN
  ELSE
    Read(z);
  END
END
```

```
BEGIN
  Read(x, y);
END
```

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

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*.

```
MODULEBackwardSlicing;

VAR
   x, y, z: INTEGER;

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

```
MODULEForwardSlicing;

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*.

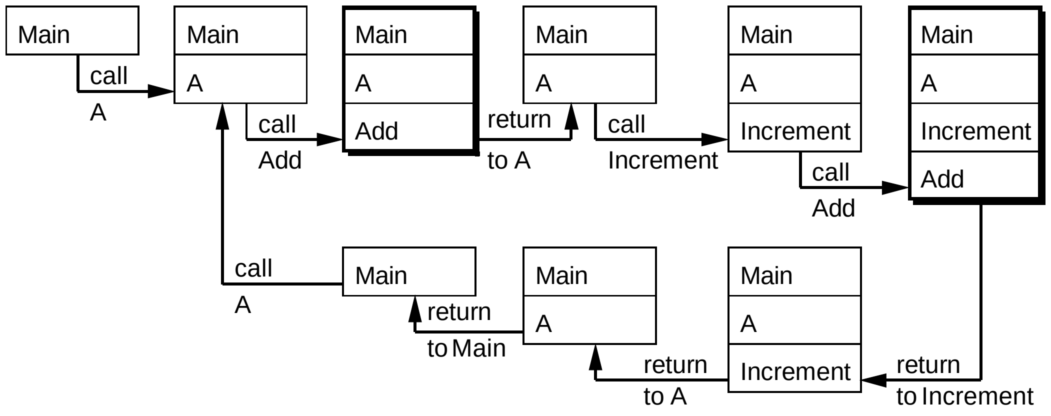


Fig. 2.17 - Trace of procedure activations (activations of *Add* are shaded)

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.

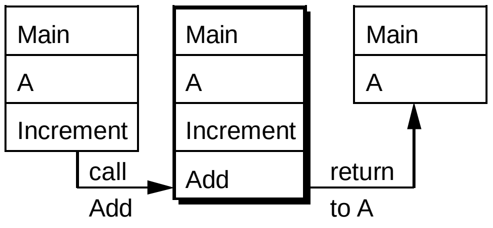


Fig. 2.18 - Trace of procedure activations equivalent to wrong handling of calling context

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*).

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.