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.

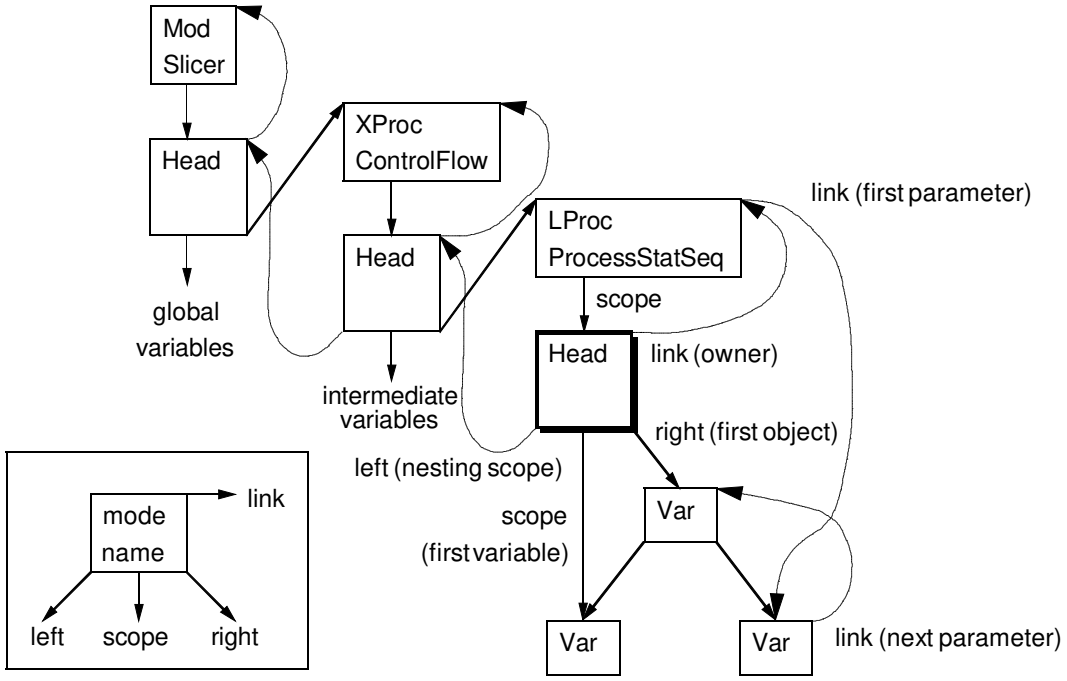

Fig. 4.1 - Scope of accessible objects of procedure *Slicer.ControlFlow.ProcessStatSeq*

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.

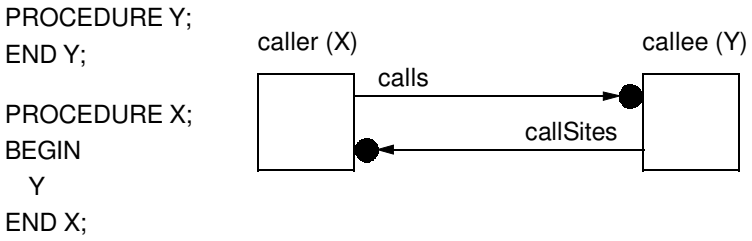

Fig. 4.2 - Bidirectional call relation between procedures

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);
```

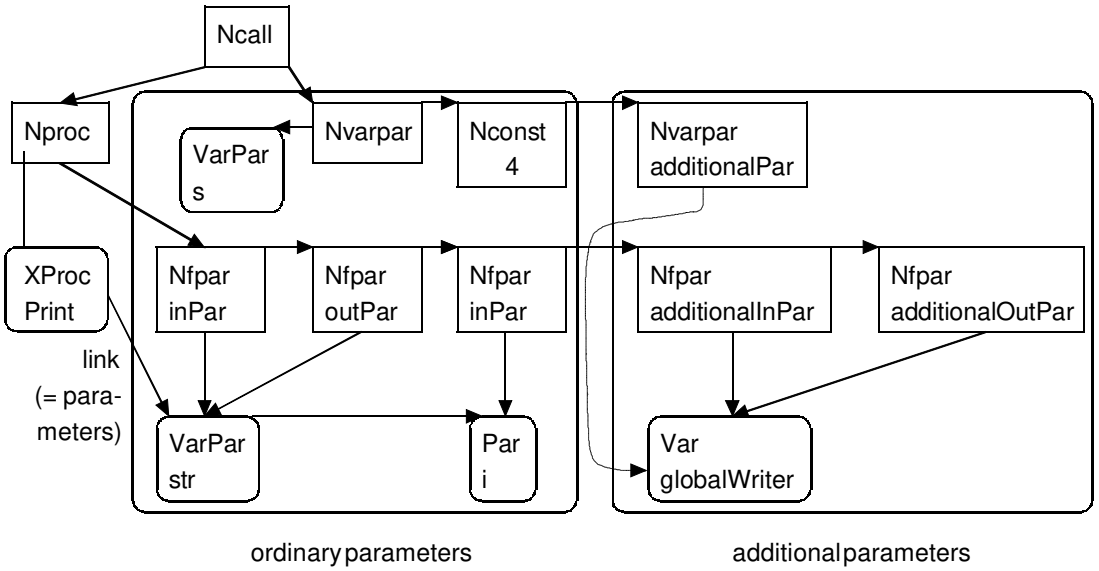

Fig. 4.3 - Procedure call with ordinary and additional parameters

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
```

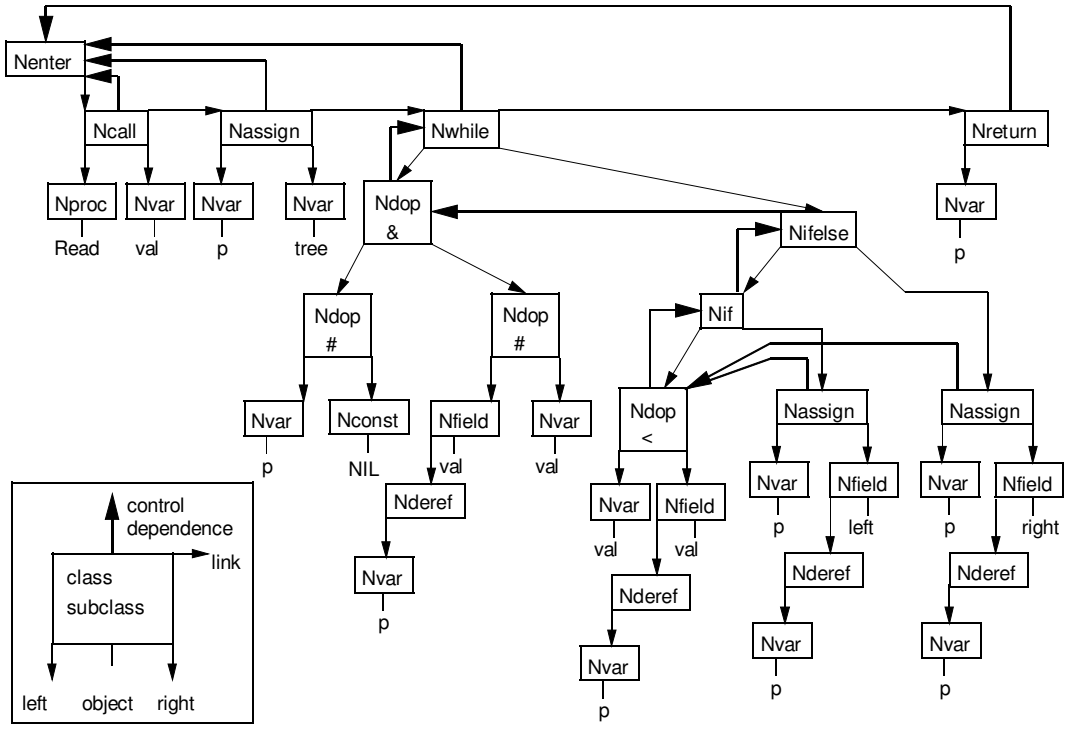

Fig. 4.4 - AST of the statement sequence in Example 4.2 with control dependences

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
```

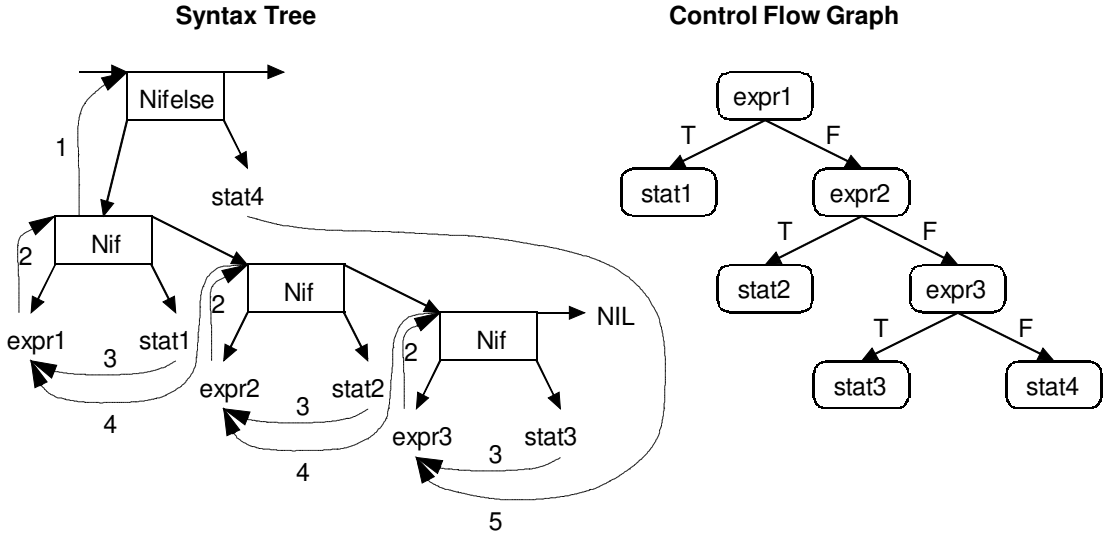

Fig. 4.5 - Syntax tree and control flow graph for an IF statement

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
```

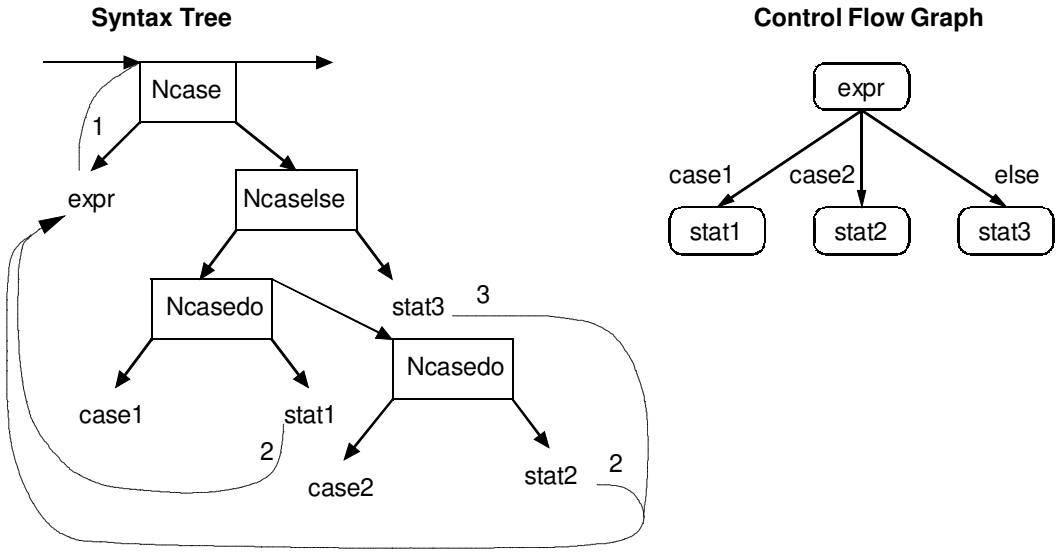

Fig. 4.6 - Syntax tree and control flow graph of a CASE statement

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
```

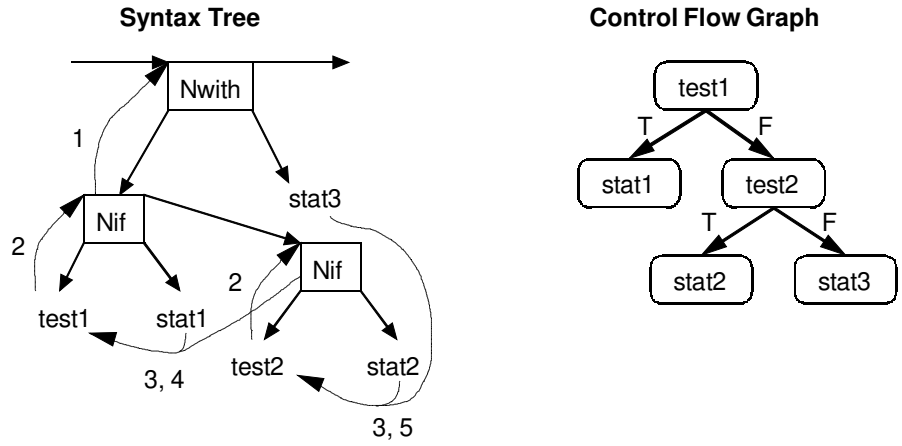

Fig. 4.7 - Syntax tree and control flow graph of a WITH statement

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
```

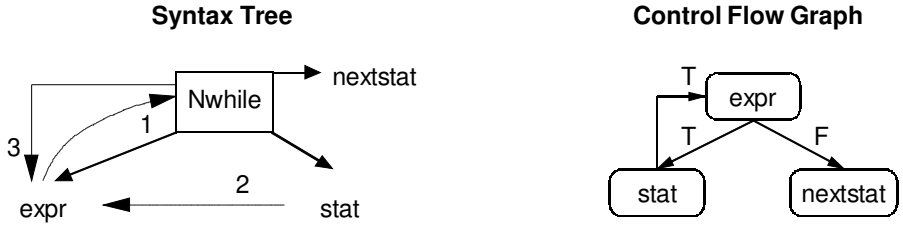

Fig. 4.8 - Syntax tree and control flow graph of a WHILE statement

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
```

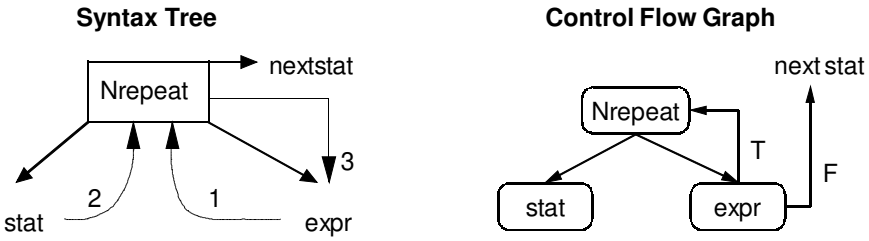

Fig. 4.9 - Syntax tree and control flow graph of a REPEAT statement

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);
```

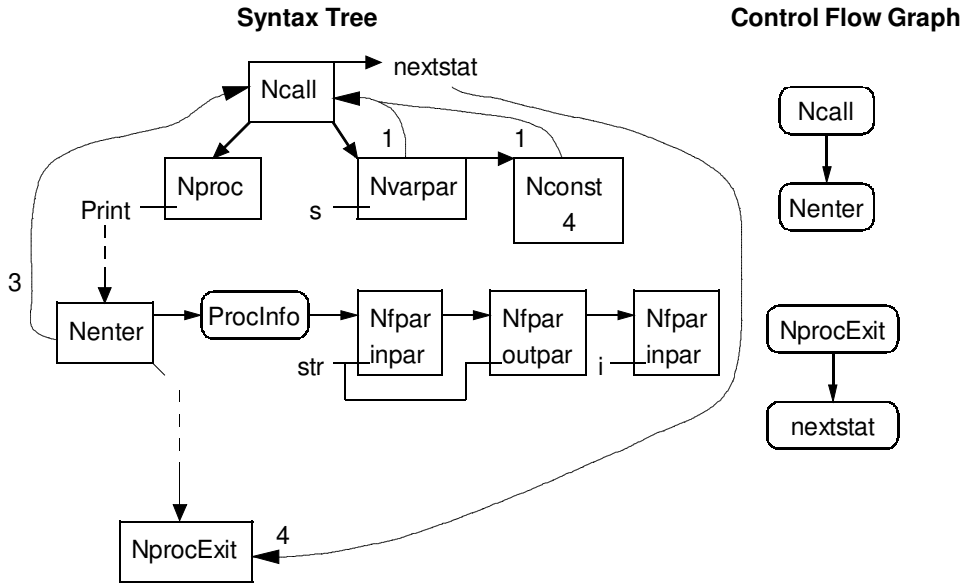

Fig. 4.10 - Syntax tree and control flow graph for a call statement

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).

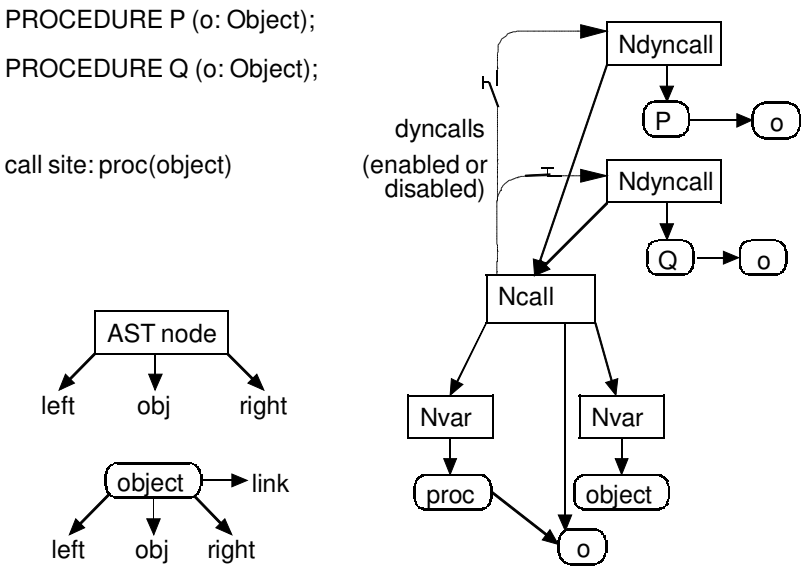

Fig. 4.11 - Dynamically bound procedure call with *Ndyncall* nodes

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."

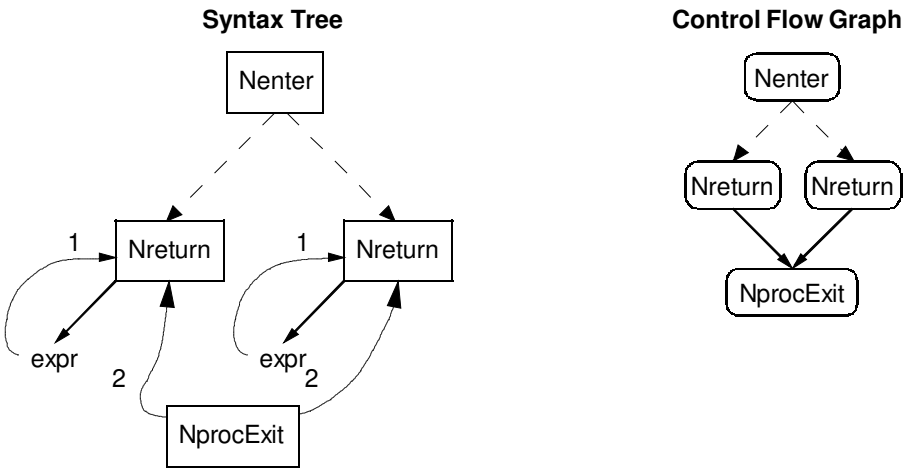

Fig. 4.12 - Syntax tree and control flow graph of a RETURN statement

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
```

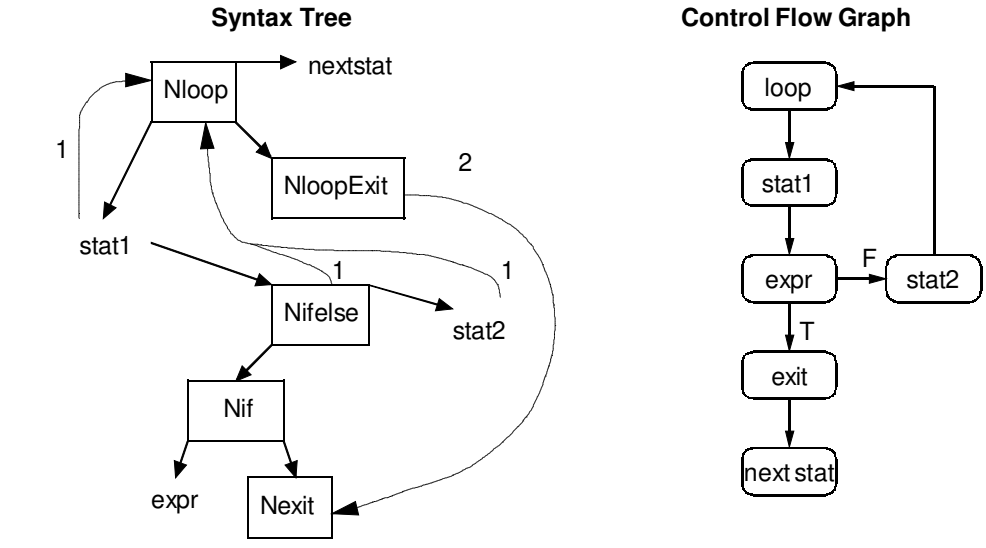

Fig. 4.13 - Syntax tree and control flow graph of a LOOP statement

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;
```

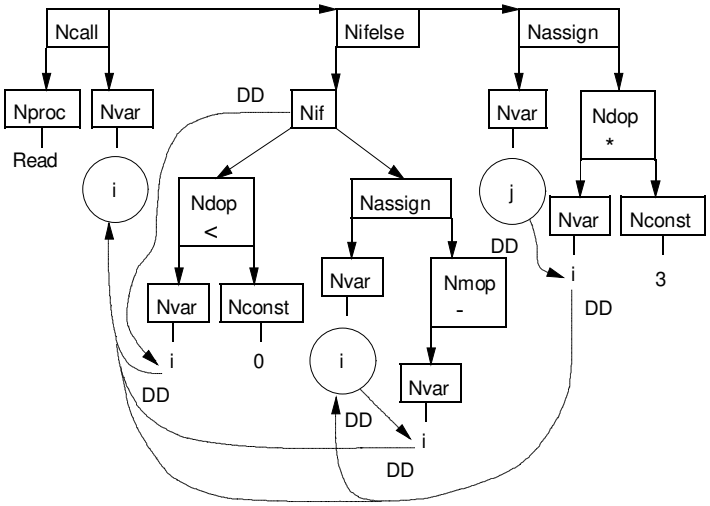

Fig. 4.14 - AST of the statement sequence in Example 4.10
with encircled definitions of variables, and data dependences (DD)

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*.

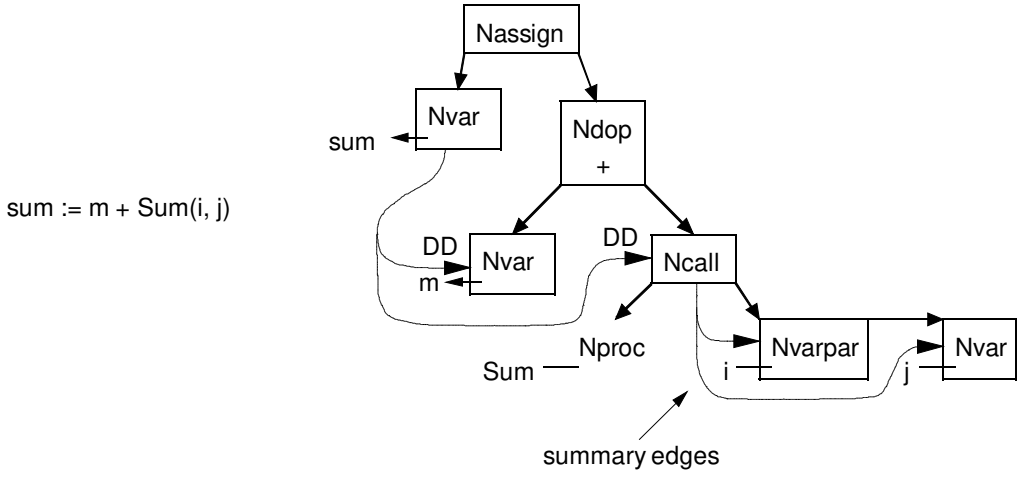

Fig. 4.15 - Data dependences for an assignment statement

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;
```

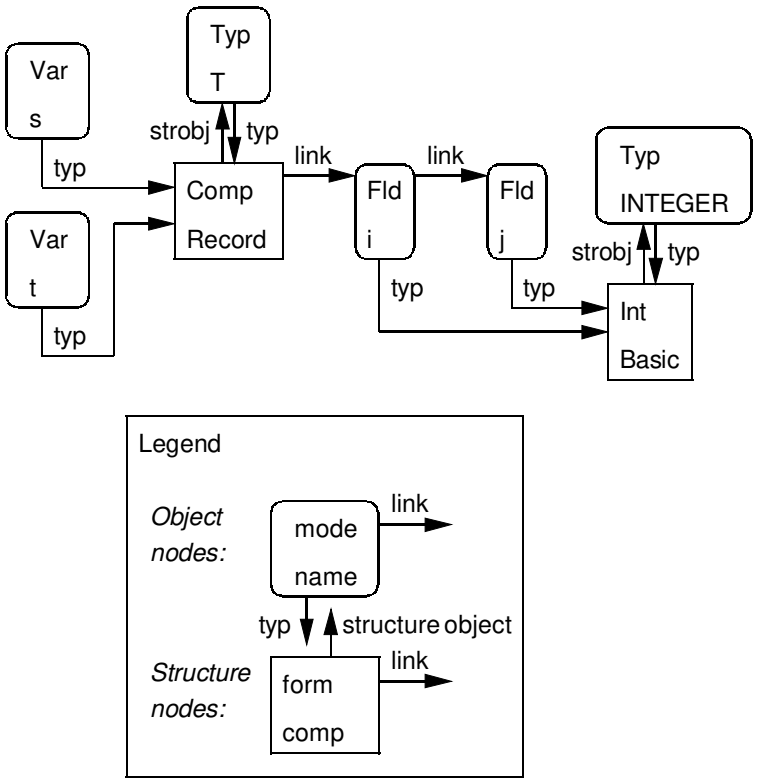

Fig. 4.16 - Symbol table entries for records and their fields

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;
```

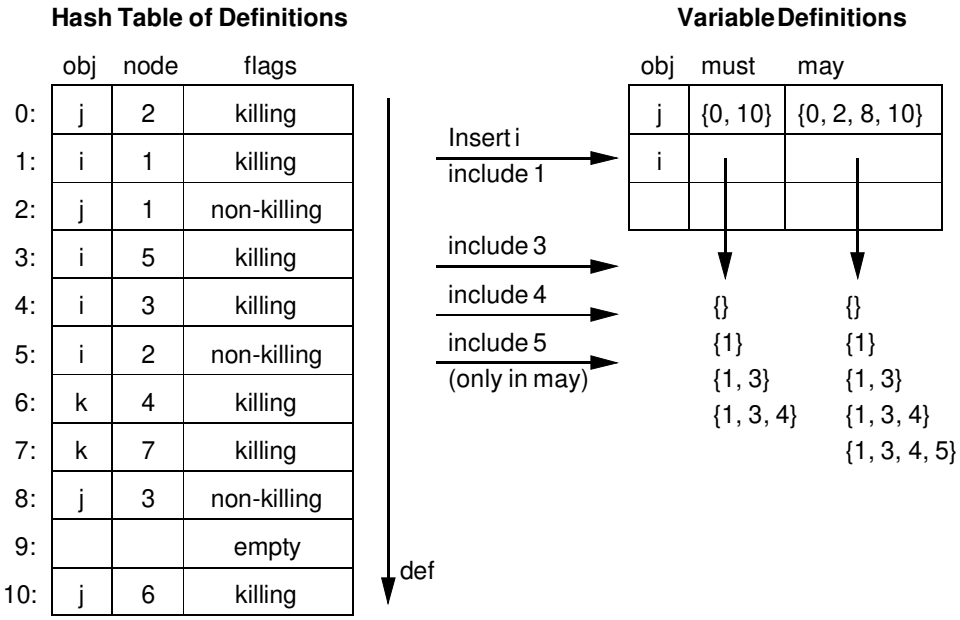

Fig. 4.17 - Computation of the definition sets of parameter *i*

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.

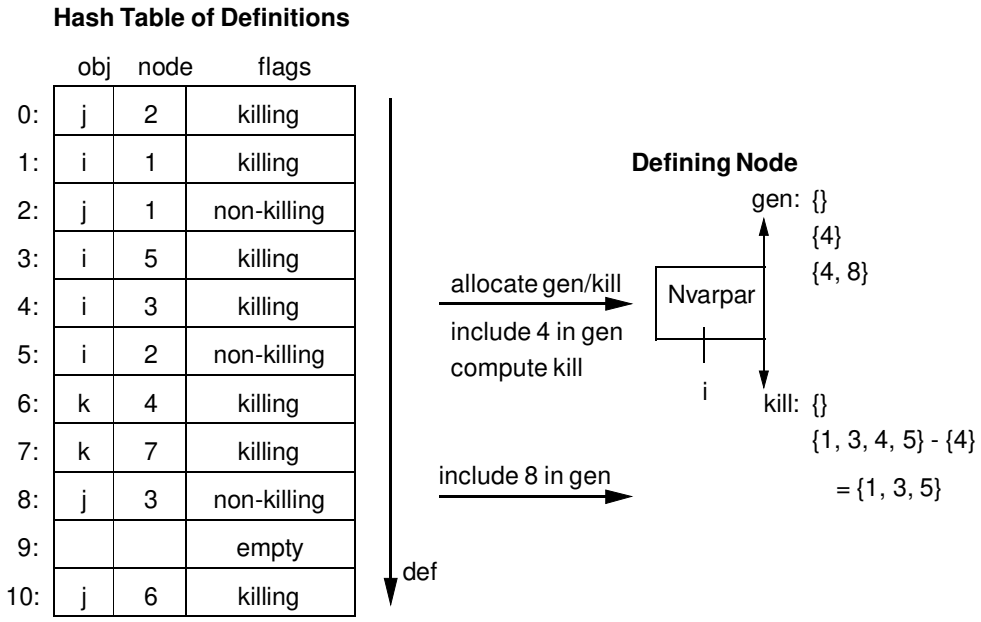

Fig. 4.18 - Computation of *gen* and *kill* for node 3

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:

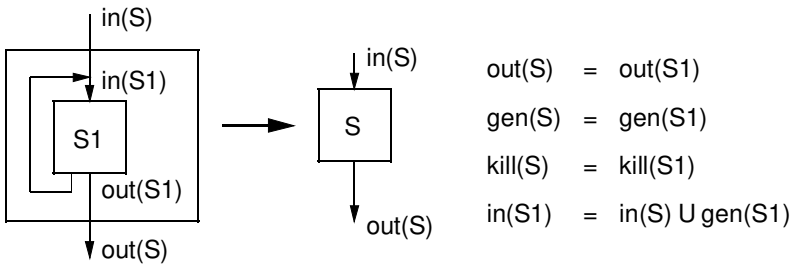

Fig. 4.19 - Data flow equations for an iterative statement

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*:

$$I_2 = J \cup O_1 = J \cup G \cup (J - K)$$
$$= J \cup G \qquad \text{since last term } J - K \text{ is a subset of } J$$

The next approximation for *O* is:

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

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:

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

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

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

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}

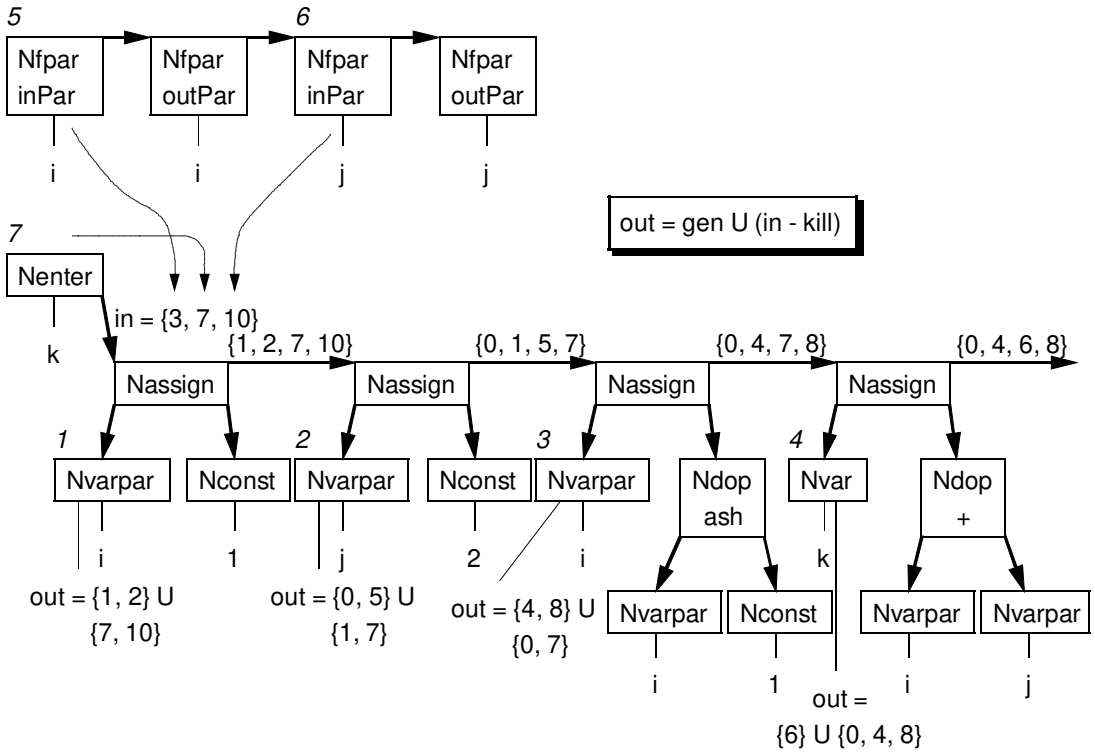

Fig. 4.20 - Computation of the *out* set of procedure *X* of Example 4.19

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.

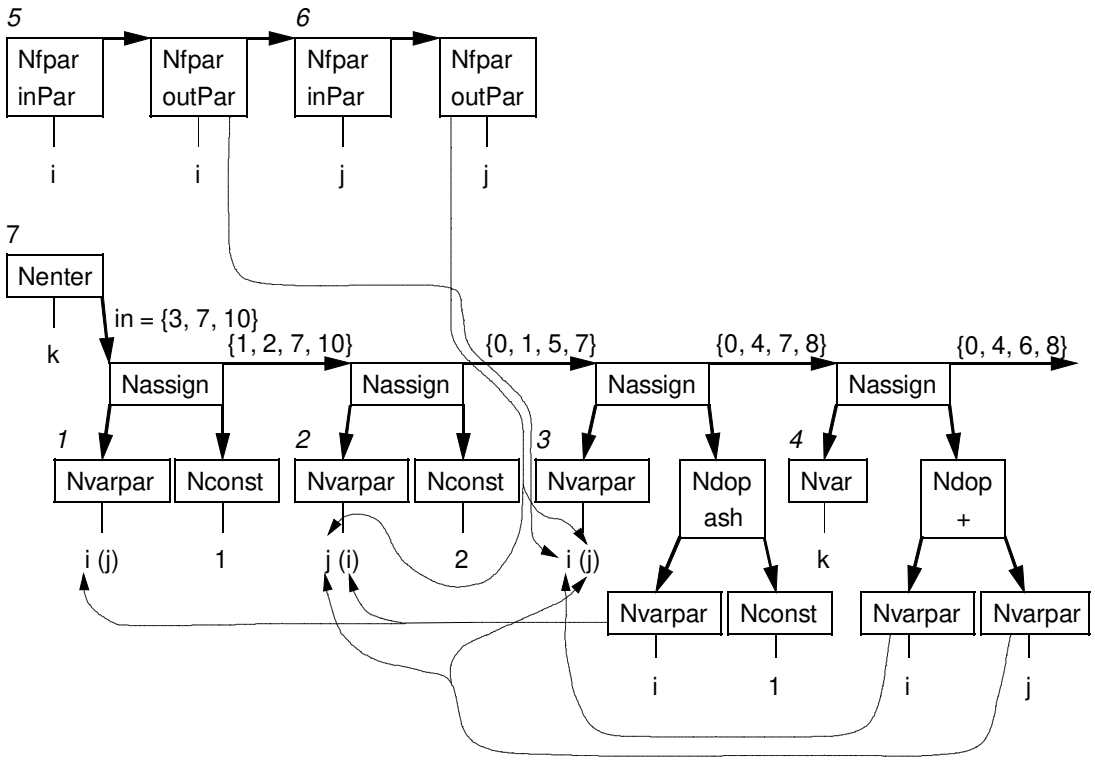

Fig. 4.21 - Reaching definitions of procedure *X* of Example 4.19

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:

```
          MODULEShortCircuitEvaluation;

          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;

          ENDShortCircuitEvaluation.
```

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.

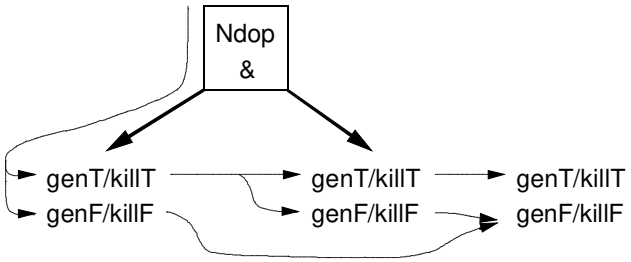

Fig. 4.22 - Computation of *gen* and *kill* for a logical AND

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

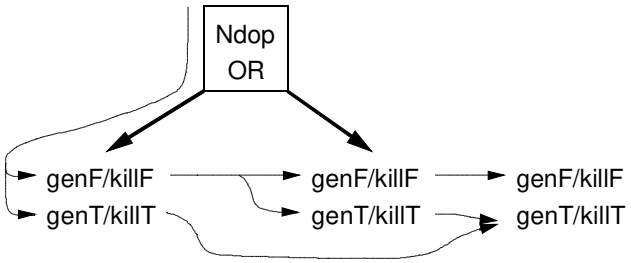

Fig. 4.23 - Computation of *gen* and *kill* for a logical OR

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

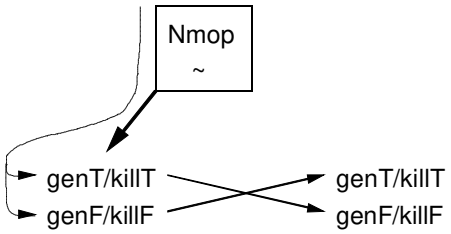

Fig. 4.24 - Computation of *gen* and *kill* for a logical NOT

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.

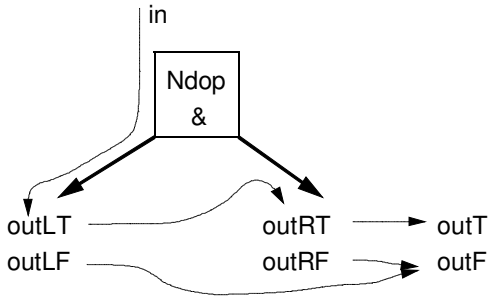

Fig. 4.25 - Computation of *out* for a logical AND

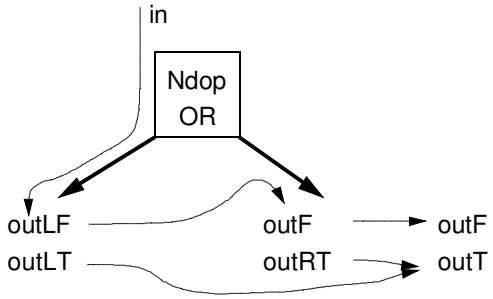

Fig. 4.26 - Computation of *out* for a logical OR

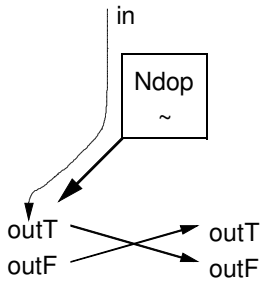

Fig. 4.27 - Computation of *out* for a logical NOT

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.

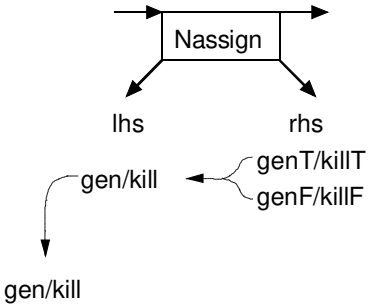

Fig. 4.28 - Computation of *gen* and *kill* for an assignment statement

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.

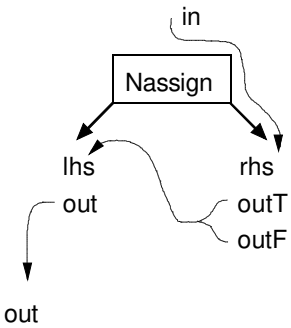

Fig. 4.29 - Computation of *out* for an assignment statement

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
```

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

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

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

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.

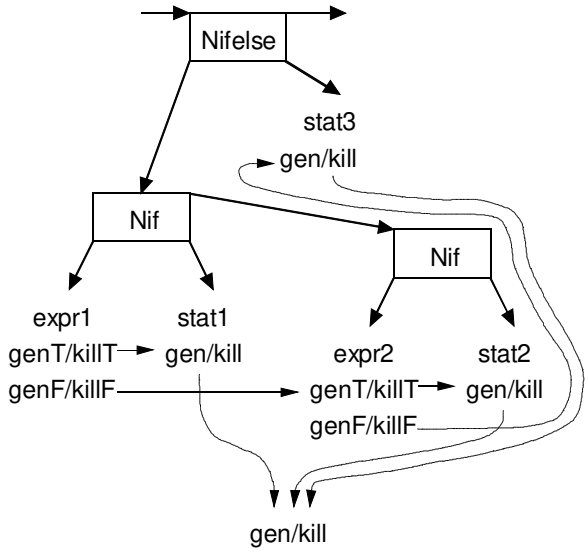

Fig. 4.30 - Computation of *gen* and *kill* for an IF statement

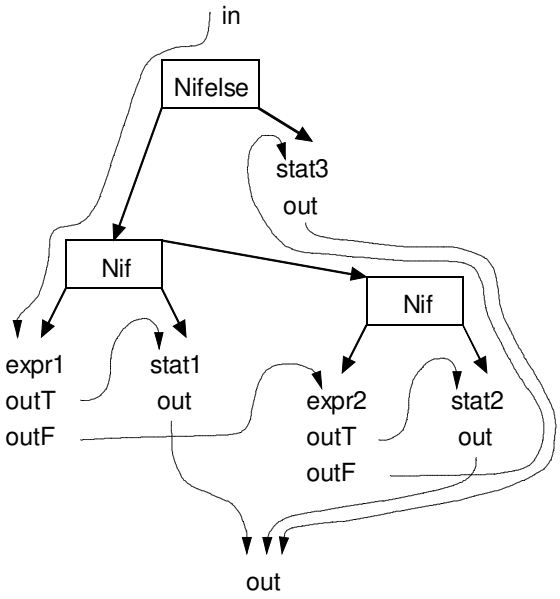

Fig. 4.31 - Computation of *out* for an IF statement

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.

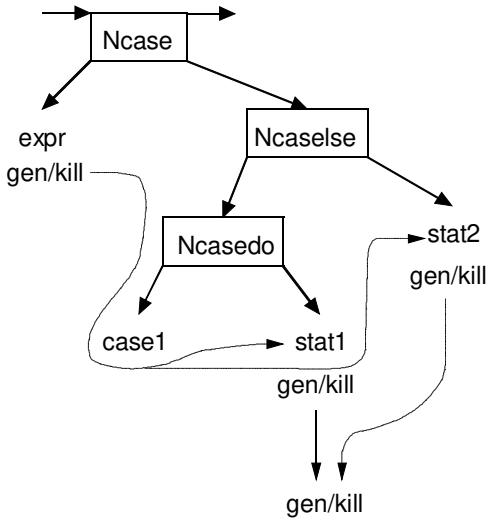

Fig. 4.32 - Computation of *gen* and *kill* for a CASE statement

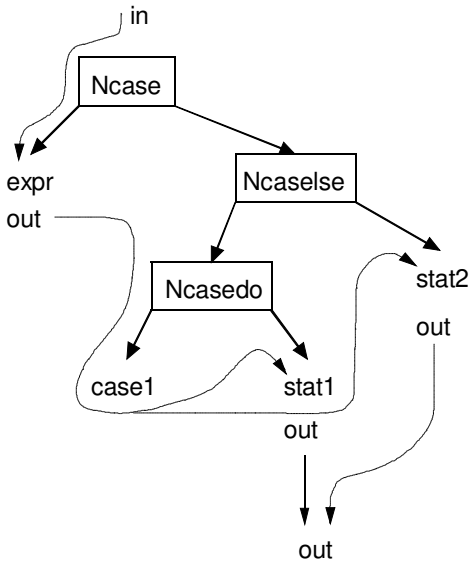

Fig. 4.33 - Computation of *out* for a CASE statement

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.

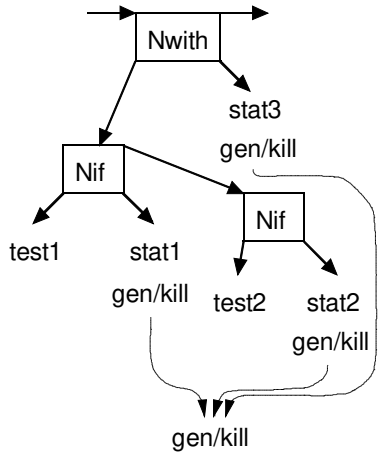

Fig. 4.34 - Computation of *gen* and *kill* for a WITH statement

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.

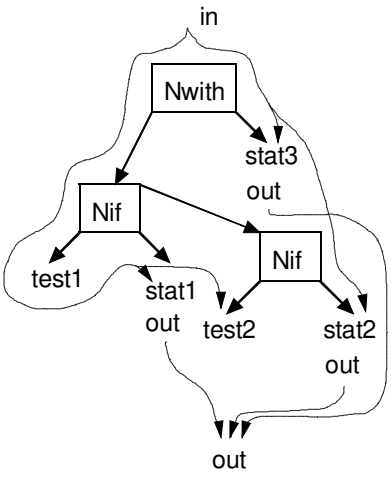

Fig. 4.35 - Computation of *out* for a WITH statement

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.

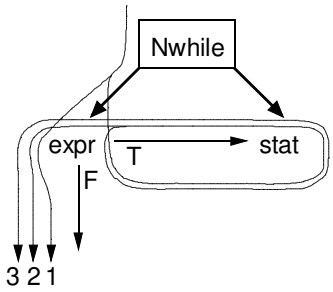

Fig. 4.36 - Iterations of a WHILE loop

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.

$$\begin{aligned} gen/kill_{WHILE} &= \{\ genT/killT_{expr}\ gen/kill_{stat}\ \}\ genF/killF_{expr} = \\ &= [\ genT/killT_{expr}\ gen/kill_{stat}\ ]\ genF/killF_{expr} = \\ &= genF/killF_{expr} \mid \\ &\quad genT/killT_{expr}\ gen/kill_{stat}\ genF/killF_{expr} \end{aligned}$$

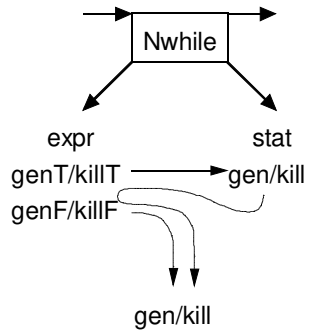

Fig. 4.37 - Computation of *gen* and *kill* for a WHILE statement

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 $in(S) \cup gen(S1)$. 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:

$$\begin{aligned} out'' &= gen(S1) \cup (out' - kill(S1)) = \\ &= gen(S1) \cup ((gen(S1) \cup (in(WHILE) - kill(S1))) - kill(S1)) = \\ &= gen(S1) \cup (gen(S1) - kill(S1)) \cup (in(WHILE) - kill(S1) - kill(S1)) = \\ &= gen(S1) \cup (in(WHILE) - kill(S1)) = out' \end{aligned}$$

   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

$$\begin{aligned} in(WHILE) \cup out' &= in(WHILE) \cup gen(S1) \cup (in(WHILE) - kill(S1)) = \\ &= in(WHILE) \cup gen(S1) \end{aligned}$$

   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)* ∪ *((in1* ∪ *in2) - kill(S))* =
= *gen(S)* ∪ *(in1 - kill(S))* ∪ *(in2 - kill(S))* =
= *gen(S)* ∪ *(in1 - kill(S))* ∪ *gen(S)* ∪ *(in2 - kill(S))* =
= *out1* ∪ *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) ∪ gen(S1) => $expr_{TRUE}$ stat => out'
in(WHILE) ∪ 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.

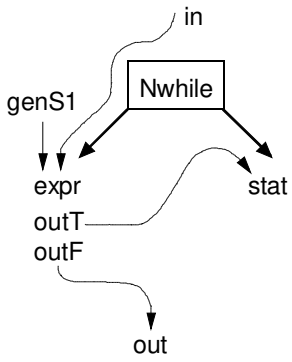

Fig. 4.38 - Computation of *out* for a WHILE statement

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.

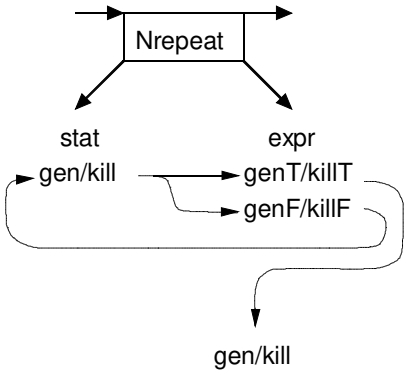

Fig. 4.39 - Computation of *gen* and *kill* for a REPEAT statement

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:

$$\begin{aligned} gen/kill_{REPEAT} &= \{\, gen/kill_{stat}\ genF/killF_{expr}\,\}\ gen/kill_{stat}\ genT/killT_{expr} \quad = \\ &= [\, gen/kill_{stat}\ genF/killF_{expr}\,]\ gen/kill_{stat}\ genT/killT_{expr} \quad = \\ &= gen/kill_{stat}\ genT/killT_{expr}\ | \\ &\quad\ gen/kill_{stat}\ genF/killF_{expr}\ gen/kill_{stat}\ genT/killT_{expr} \end{aligned}$$

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) $in(REPEAT) \Rightarrow stat\ expr_{TRUE} \Rightarrow out_1(REPEAT)$
2) $in(REPEAT) \Rightarrow stat\ expr_{FALSE} \Rightarrow out' \Rightarrow stat\ expr_{TRUE} \Rightarrow out_2(REPEAT)$

   out' could be computed as "*gen(S1)* $\cup$ *(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)* $\cup$ *gen(S1)*" into "*stat* $expr_{TRUE}$", since

$$\begin{aligned} in(REPEAT) \cup out' &= in(REPEAT) \cup gen(S1) \cup (in(REPEAT) - kill(S1)) = \\ &= in(REPEAT) \cup gen(S1) \end{aligned}$$

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

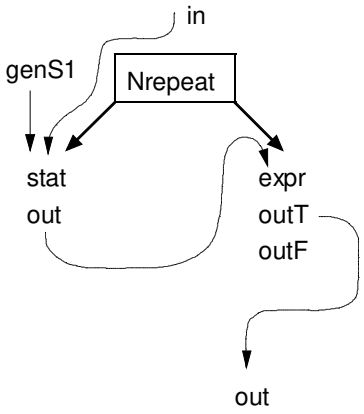

Fig. 4.40 - Computation of *out* for a REPEAT statement

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)
```

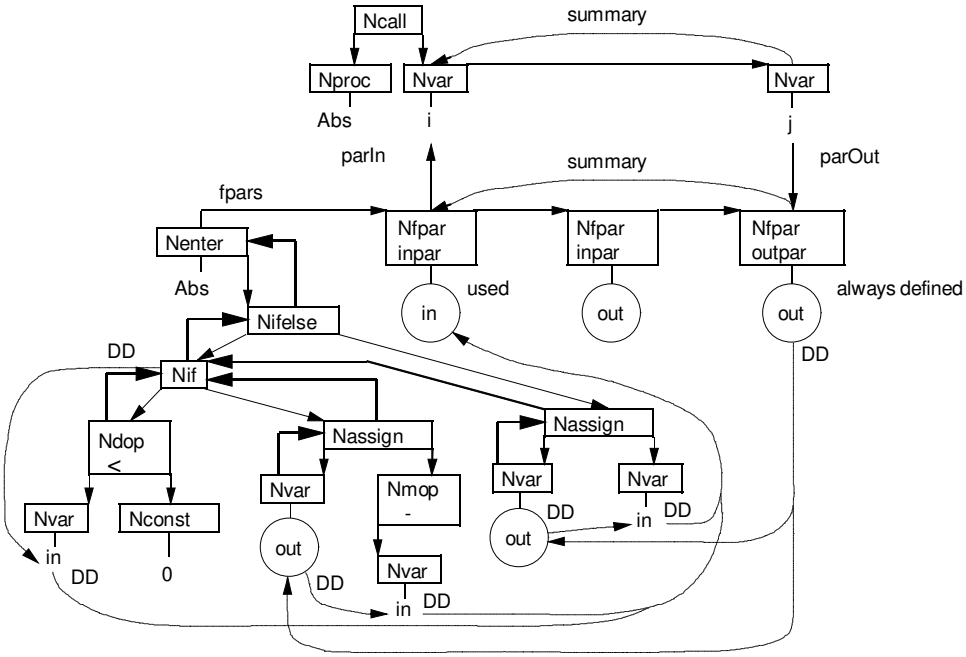

Fig. 4.41 - Computation of summary edges

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.

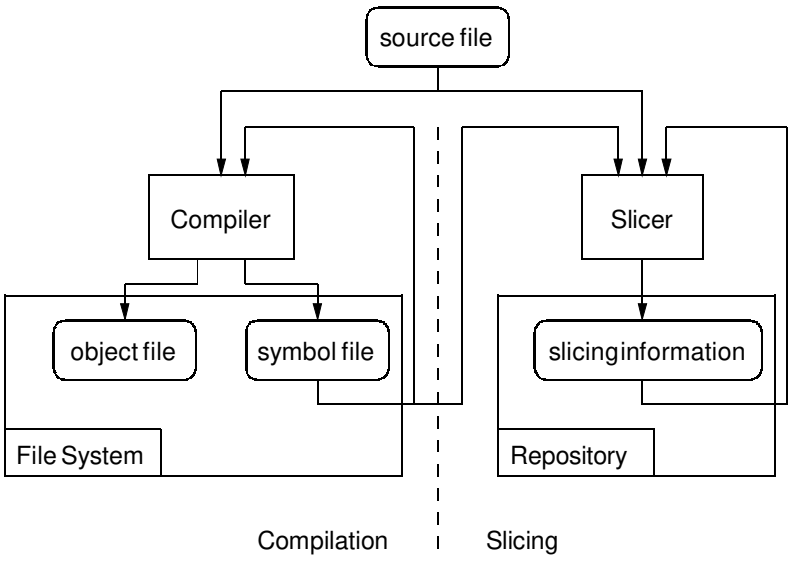

Fig. 4.43 - Compilation versus slicing

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.

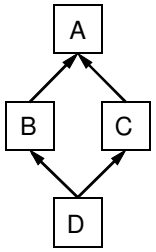

Fig. 4.44 - Import graph of four modules

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.

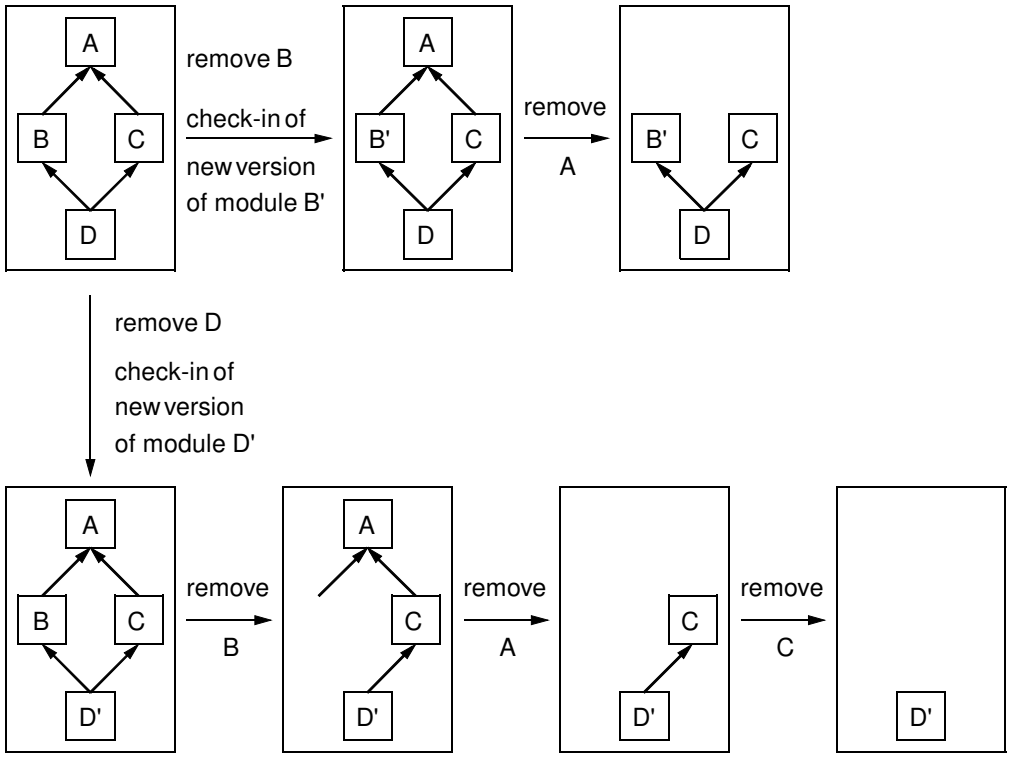

Fig. 4.45 - Recursive removal of invalidated slicing information

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.

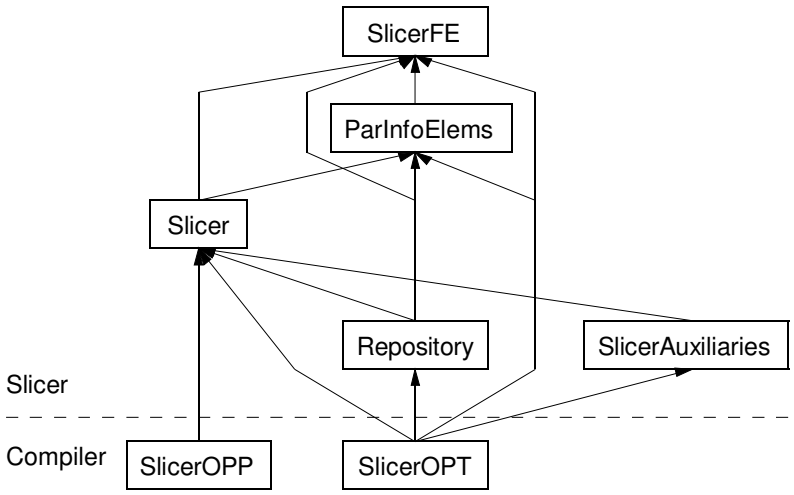

Fig. 4.46 - Import graph of the Oberon Slicing Tool

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.
- *ShowModuleInfomodName* 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*