5 User Interface

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

5.1 Visual Elements

We extended the Active Text Framework (see [Szy92] and [MöKo96]) of the Oberon System to visualize control flow and data flow information. This framework allows arbitrary objects such as pictures, tables or buttons to be inserted into the text like ordinary characters. These objects are called *text elements* and are derived from the abstract base class *Texts.Elem*. Text elements are *active*, because they react on mouse clicks. For example, a text element representing a hypertext link will cause the editor to scroll to another text position when the user clicks on the link. Another kind of active text elements are *popup elements* that represent a menu that pops up in reaction to a click.

5.1.1 Bidirectional Links Between the Caller and the Callee

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

```
MODULE VisualizeLinks;

IMPORT In, Texts, Files;

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

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

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

END VisualizeLinks.
```

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

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

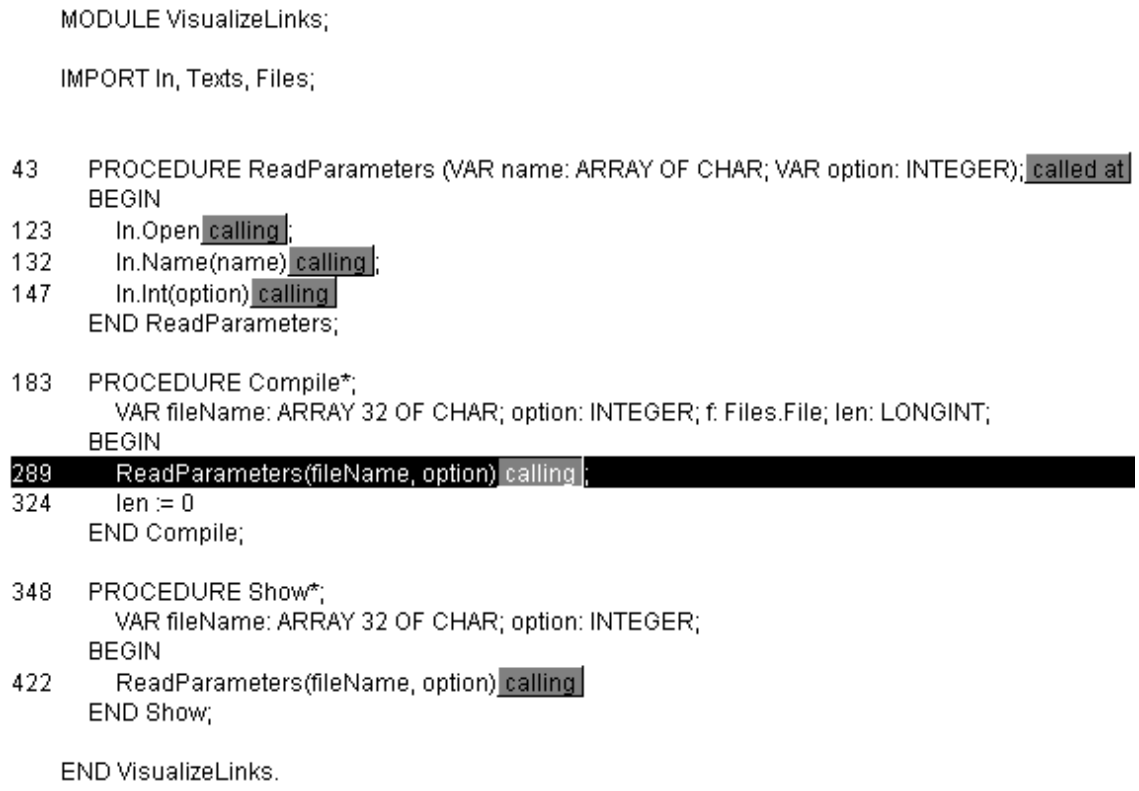

Fig. 5.2 - Navigation from call destination back to call site

5.1.2 DataDependences

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

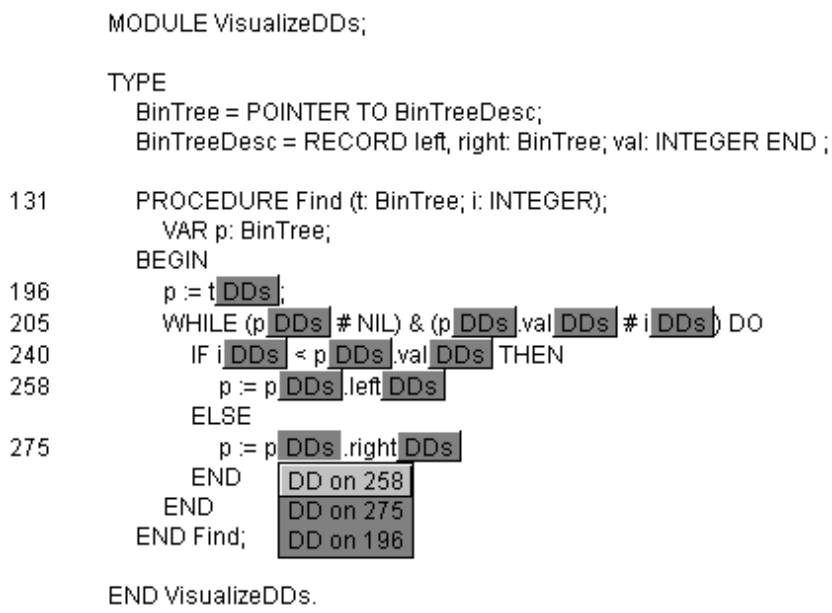

Fig. 5.3 - Visualization of data dependences

5.1.3 Parameters

Parameter Usage

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

```
        MODULE VisualizeParElems;

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

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

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

        END VisualizeParElems.
```

Fig. 5.4 - Parameter information elements

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

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

        END VisualizeAliases.
```

Fig. 5.5 - Alias information via parameter information elements

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

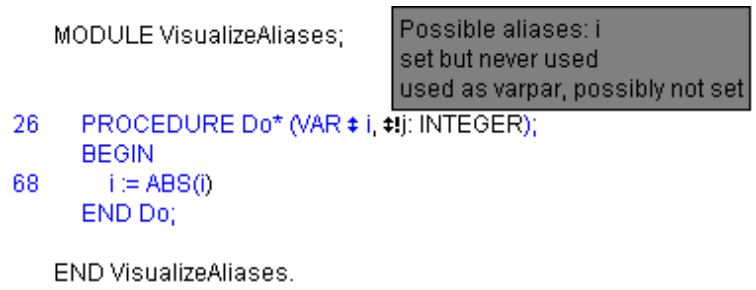

Fig. 5.6 - Additional information via parameter information elements

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

Additional Parameters

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

```
MODULE VisualizeAdditionalPars;

IMPORT In, Texts, Out;

VAR
   sum: INTEGER;

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

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

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

END VisualizeAdditionalPars.
```

Fig. 5.7 - Additional parameters

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

5.1.4 Aliases

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

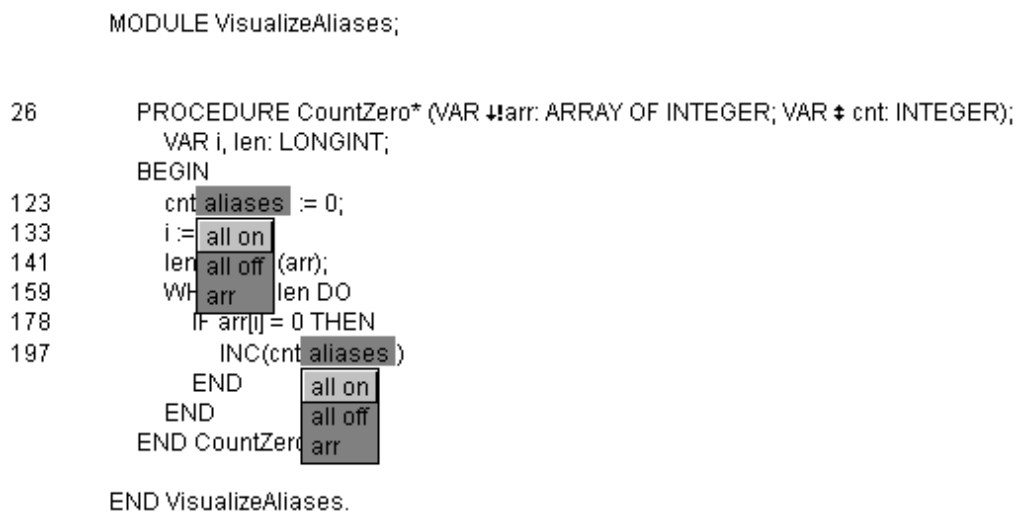

Fig. 5.8 - Possible aliases at definitions

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

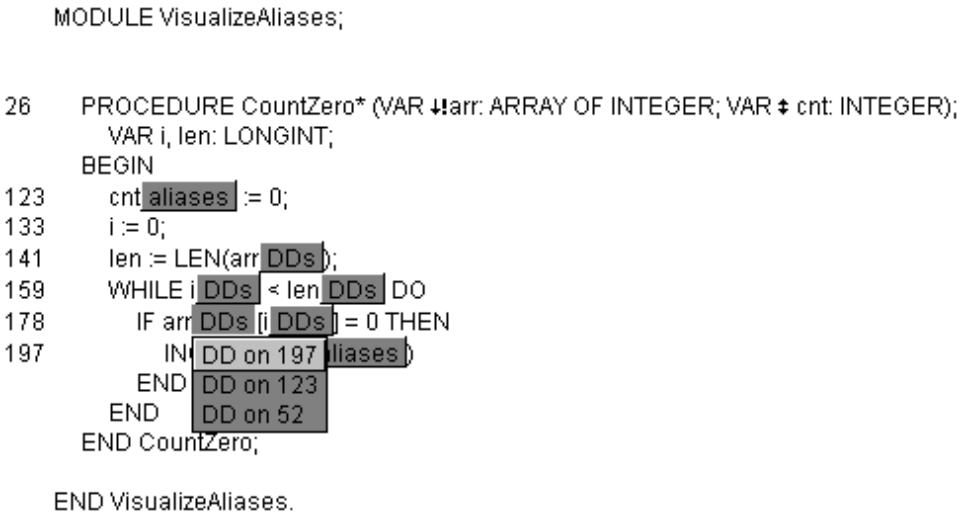

Fig. 5.9 - Reaching definitions with all aliases enabled

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

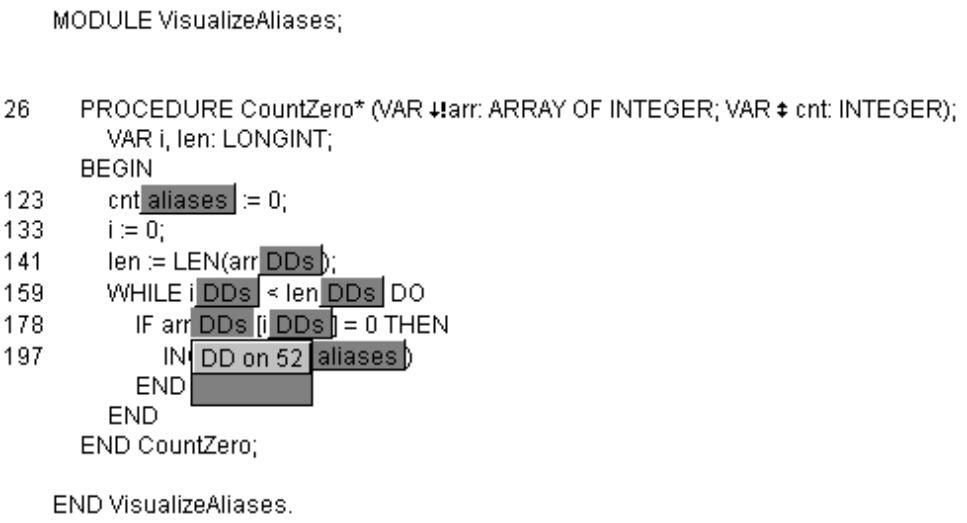

Fig. 5.10 - Reaching definitions with all aliases disabled

5.1.5 DynamicTypes

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

procedure variable. This allows us to treat dynamic binding due to type-bound procedures and procedure variables uniformly. We insert popup elements labeled *dynamic types* at dynamically bound call sites. These elements contain one entry for each possible call destination. The user can disable and enable some of the dynamic types by selecting them. Initially all dynamic types are enabled. The popup element shows enabled dynamic types in black and disabled dynamic types in grey. In Fig. 5.11, procedure *ForAll* contains a dynamically bound call with two possible call destinations *Inc* and *PrintNode*. They are both initially enabled, leading to their additional parameters (namely parameter *count* of procedure *Inc*) being shown at the call site in line with number 289.

```
      MODULE VisualizeDynTypes;

      IMPORT Out;

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

197   PROCEDURE ForAll* (↓ workProc: WorkProc    (* VAR ↕ count: INTEGER; VAR ↓!head: Node *));
        VAR n: Node;
      BEGIN
258     n := head;
270     WHILE n # NIL DO
289       workProc(↓n  (* ↕count*)) dynamic types ;
304       n := n.next                  all on
        END                            all off
      END ForAll;                      Inc
                                       PrintNode
334   PROCEDURE Inc (↓!n: Node    (* VAR ↕ count: INTEGER *));
      BEGIN
365     INC(count)
      END Inc;

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

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

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

      END VisualizeDynTypes.
```

Fig. 5.11 - Dynamic types of procedure variables

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

```
      MODULE VisualizeDynTypes;

      IMPORT Out;

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

197   PROCEDURE ForAll* (↓ workProc: WorkProc    (* VAR ↓!head: Node *));
        VAR n: Node;
      BEGIN
258     n := head;
270     WHILE n # NIL DO
289       workProc(↓n) dynamic types ;
304       n := n.next
        END
      END ForAll;
                                  all on
                                  all off
                                  Inc
                                  PrintNode
334   PROCEDURE Inc (↓!n: Node    (* VAR ↕ count: INTEGER *));
      BEGIN
365     INC(count)
      END Inc;

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

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

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

      END VisualizeDynTypes.
```

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

5.2 User Feedback

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

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

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

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

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

5.3 Module SlicerFE

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

```
DEFINITION SlicerFE;

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

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

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

      ...

   VAR
      forcePersistence: BOOLEAN;
      recording: BOOLEAN;

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

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

END SlicerFE.
```

Variables:

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

Commands:

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

specified options or the default options for the target slice. The option may be specified by number (e.g., 1 for *withDDElems*) or by name (e.g., *withDDElems*).

- *ShowOptions* outputs the options used to reconstruct the source code of the target viewer.
- *FindNode [^] position* calls method *ThisNode(position)* for the target slice. The return value is stored in the variable *SlicerFE.node* which can be inspected via *InspectSlice.*
- *FindProc [^] procName* calls method *ThisProc(procName)* for the target slice. The return value is stored in the variable *SlicerFE.node* which can be inspected via *InspectSlice.*

Types:

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

5.4 Model-View-Controller Concept

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

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

The model is represented by an object of type *Slicer.Slice*. The view and the controller are combined by an object of type *SlicerFE.Frame*. At the time of writing this thesis, two kinds of views have been implemented, the standard view and the call hierarchy view. Additional views can be implemented. They are kept consistent by broadcasting the *SliceMsg* into the "viewer space". Each viewer that displays the changed slice reacts to the indicated operation properly, e.g. by updating its view. An example shall demonstrate this: Assume that two standard viewers (*V1* and *V2*) and one call hierarchy viewer (*V3*) display the same slice. The user clicks on a statement in *V1,* upon which the view orders the model to compute the slice

for the statement. After having computed the slice, the model broadcasts the message *SlicerFE.SliceMsg* to all visible viewers. Viewers *V1*, *V2*, and *V3* react by updating their view. This is very similar to the way that multiple views of the same text are kept consistent in the Oberon System.