PROGRAM SLICING

JENS KRINKE

Faculty of Electrical and Information Engineering,
FernUniversität in Hagen, 58084 Hagen, Germany
E-mail: Jens.Krinke@FernUni-Hagen.de

Program slicing is a technique to identify statements that may influence the computations of other statements. The original goal was to aid program understanding and debugging. Program slicing is now used as a base technique in other applications. Researchers in several areas of software engineering have suggested applications in program comprehension, software maintenance, reverse engineering and evolution, testing, and even verification. The following will give an overview of program slicing, including recent developments. This overview covers static as well as dynamic slicing. It contains a discussion of applications and basic approaches to compute slices, shows problems and their solutions, together with refinements of slicing.

Keywords: Program slicing, program dependence graph.

1. Introduction

> *Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.*
>
> Mark Weiser [115]

Program Slicing is a widely used technique for various aspects of software engineering. It basically answers the question "Which statements may affect the computation of a given statement?", something every programmer has asked. After Weiser's first publication in 1979, 25 years have passed and various approaches to compute slices have evolved. However, only recently have mature systems capable of slicing real-world systems become available. The availability of these systems makes it possible to fully explore and exploit the applications of slicing.

The original goal was to aid program understanding and debugging. Program slicing is now used as a base technique in other applications. Researchers in several areas of software engineering have suggested applications in program comprehension, software maintenance, reverse engineering and evolution, testing, and even verification. The following will give an overview of program slicing, including recent

```
 1   read(n)
 2   i := 1
 3   s := 0
 4   p := 1
 5   while (i <= n)
 6     s := s + i
 7     p := p * i
 8     i := i + 1
 9   write(s)
10   write(p)
```

(a) Original program

```
 1   read(n)
 2   i := 1
 3
 4   p := 1
 5   while (i <= n)
 6
 7     p := p * i
 8     i := i + 1
 9
10   write(p)
```

(b) Static Slice for (10, p)

```
 1
 2
 3
 4   p := 1
 5
 6
 7
 8
 9
10   write(p)
```

(c) Dynamic Slice for (10, p, n = 0)

Fig. 1. A program and two slices.

developments, thus complementing earlier works like Tip's excellent survey [107] and others [32,52].

A slice extracts those statements from a program that potentially have an influence on a specific statement of interest, which is the slicing criterion. Consider the example in Fig. 1(a). This program computes the product `p` and the sum `s` of integer numbers up to a limit `n`. Let us assume that we are only interested in the computation of the product and its output in line 10. When we eliminate all statements that have no impact on the computation, we end up with the program shown in Fig. 1(b) that still computes the product correctly. This is called a *slice* for the *slicing criterion* "variable `p` in line 10". The process of computing a slice is called *slicing*. Because this slice is independent of the program's inputs and computes `p` correctly for all possible executions, it is called a *static* slice. If we are interested in the statements that have an impact on the criterion for a *specific* execution, we can compute a *dynamic* slice. A dynamic slice eliminates all statements of a program that have no impact on the slicing criterion for a specific execution as determined by the input for this execution. In Fig. 1(c), a dynamic slice is shown for the execution where the input to variable `n` is 0. Not only the complete loop has been deleted because the body is never executed, but also the input statement itself.

Originally, (static) slicing was defined by Weiser in 1979; he presented an approach to compute slices based on iterative data flow analysis [113,115]. Another popular approach to slicing uses reachability analysis in program dependence graphs [36]. Program dependence graphs (PDGs) mainly consist of nodes representing the statements of a program, and control and data dependence edges, which is discussed in more detail in the next section. In PDGs, static slicing of programs can be computed by identifying the nodes that are reachable from the node corresponding to the criterion [36]. The underlying assumption is that all paths are *realizable*. This means that for every path a possible execution of the program exists that executes the statements in the same order. In the presence of procedures things

get complicated. Paths are now considered realizable only if they obey the calling context (i.e. called procedures always return to the correct call site). This will be discussed in Sec. 4.3.

This chapter is restricted to slicing of imperative programs. However, this is not the only slice-able thing: There are other approaches to slicing of hierarchical state machines [55], web applications [99], class hierarchies [106], knowledge-based systems [111], logical programs [110], and hardware description languages [30]. Even binary executables can be sliced [65].

The rest of this chapter is structured as follows: The next section presents how slices are computed, followed by a discussion of some applications of slicing. Section 4 will present some problems of slicing and their solutions. Section 5 discusses graphical and textual visualization of slices. Section 6 presents refinements of slicing, including dynamic slicing. The chapter concludes with a presentation of available slicing tools.

2. Computing Slices

Weiser observed that programmers mentally build abstractions of a program during debugging. He formalized that process and defined slicing. He also presented an approach to compute (static) slices based on iterative data flow analysis [113, 115], which will be presented next. The other often used approach to slicing uses reachability analysis in program dependence graphs [36], presented afterwards. In the following, we will just use "slicing" for "static slicing". Dynamic slicing will be discussed in a dedicated section.

2.1. *Weiser-style slicing*

Weiser formally defined a slice as any subset of a program, that preserves a specific behavior with respect to a criterion. The criterion, also called the *slicing criterion*, is a pair $c = (s, V)$ consisting of a statement s and a subset V of the analyzed program's variables. A slice $S(c)$ of a program P on a slicing criterion c is any executable program P', where

- P' is obtained by deleting zero or more statements from P;
- whenever P halts on a given input I, P' will halt for that input; and
- P' will compute the same values as P for the variables of V on input I.

The most trivial (but irrelevant) slice of a program P is always the program P itself. Slices of interest are as small as possible: A slice $S(c)$ of a program P with respect to a criterion c is a *statement-minimal slice*, iff no other slice of P with respect to c with fewer statements exists. In general, it is undecidable if a slice $S(c)$ is statement-minimal. Weiser presented an approximation based on identifying relevant variables and statements, implemented as an iterative data flow analysis [115].

2.2. *Slicing program dependence graphs*

Ottenstein and Ottenstein [95] were the first who suggested the use of program dependence graphs to compute Weiser's slices. Program dependence graphs mainly consist of nodes representing the statements of a program and control and data dependence edges:

- *Control dependence* between two statement nodes exists if one statement controls the execution of the other.
- *Data dependence* between two statement nodes exists if a definition of a variable at one statement might reach the usage of the same variable at another statement.

An example PDG is shown in Fig. 2, where control dependence is drawn in dashed lines and data dependence in solid ones. In that figure, a slice is computed for the statement "`write(p)`". The statements "`s := 0`" and "`s := s+i`" have no direct or transitive influence on the criterion and are not part of the slice.

Slicing without procedures is trivial: Just find reachable nodes in the PDG [36]. The underlying assumption is that all paths are *realizable*. This means that a possible execution of the program exists for any path that executes the statements in the same order.

The (*backward*) *slice* $S(n)$ of a PDG at node n consists of all nodes on which n (transitively) depends:

$$S(n) = \{m \mid m \rightarrow^{*} n\}.$$

The node n is called the *slicing criterion*.

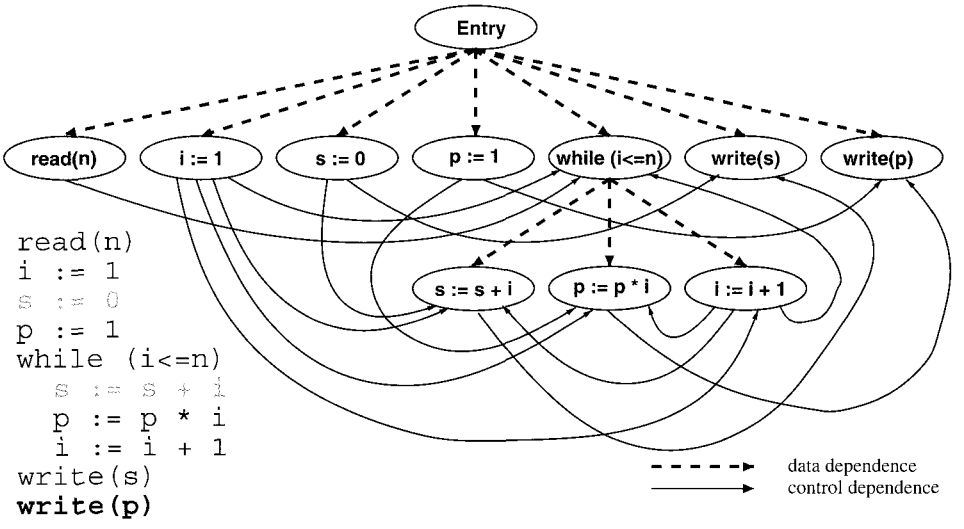

Fig. 2. A program dependence graph. The slice for the criterion "`write(p)`" is highlighted in the graph and in the source text.

This definition of a slice depends on a different slicing criterion than in Weiser-style slicing. Here, a criterion is a node in the program dependence graph, which identifies a statement together with the variable used in it. Therefore, these criteria are more restricted than Weiser-style criteria, where slices can be computed at statements for any set of variables. If a slice is to be computed at a node n for a variable that is not referenced at n, the program must be modified before analysis begins: A negligible use of that variable must be inserted at n.

While backward slicing identifies the statements that have an influence on the statement identified by the criterion, *forward slicing* identifies the statements that are influenced by the criterion:

The *forward slice* $S^F(n)$ of a PDG at node n consists of all nodes that (transitively) depend on n:

$$S^F(n) = \{m \mid n \rightarrow^\star m\}.$$

In the presence of procedures, slicing is no longer trivial in PDGs. If the calling context is ignored, the analysis is *context-insensitive*, as a called procedure may return to call sites different to the site it has been called from. However, when the calling context is obeyed, the results will be much more precise. Paths along transitive dependences are now considered realizable only if they obey the calling context. Thus, slicing is *context-sensitive* if only realizable paths are traversed. Context-sensitive slicing is solvable efficiently — one has to generate summary edges at call sites [60]: Summary edges represent the transitive dependences of called procedures at call sites. How procedural programs are analyzed will be discussed in Sec. 4.

Bergeretti and Carré [15] presented an algorithm which is neither Weiser-style nor dependence graph based. Their algorithm uses the concept of information flow relations. Venkatesh [112] developed a denotational approach to slicing.

3. Applications of Slicing

Slicing has found its way into various applications. Currently, it is probably mostly used in the area of software maintenance and reengineering. In the following, some applications are presented to show the diversity.

3.1. *Debugging*

Debugging was the first application of program slicing: Weiser [114] realized that programmers mentally ignore statements that cannot have an influence on a statement revealing a bug. Program slicing computes this abstraction and allows to focus on potentially influencing statements. Dicing [85] can be used to focus even more, when additional statements with correct behavior can be identified. Dynamic slicing (discussed in Sec. 6) can be used to focus on relevant statements for one specific execution revealing a bug [4]. More work can be found in [38].

3.2. *Testing*

Program slicing can be used to divide a program into smaller programs specific to a test case [19]. For every test case, the statements that have no impact on the tested features can be sliced away. This reduces the time needed for regression testing because of two reasons: Firstly, the smaller programs execute faster, and secondly, only a subset of the test cases has to be repeated — all test cases can be ignored where no statement of the corresponding slice has been changed since the last regression test [13, 22, 46].

3.3. *Program differencing and integration*

Program *differencing* is the problem of finding differences between two programs (or between two versions of a program). Semantic differences can be found using program dependence graphs and slicing. Program *integration* is the problem of merging two program variants into a single program. With program dependence graphs it can be assured that differences between the variants have no conflicting influence on the shared program parts [57–59].

3.4. *Software maintenance*

If a change has to be applied to a program, forward slicing can be used to identify the potential impact of the change: The forward slice reveals the part of the program that is influenced by a criterion statement and therefore is affected by a modification to that statement. Decomposition slicing [39] uses variables instead of statements as criteria.

3.5. *Function extraction and restructuring*

Slicing can also be used for *function extraction* [80]: Extractable functions are identified by slices specified by a set of input variables, a set of output variables, and a final statement. *Function restructuring* separates a single function into independent ones; such independent parts can be identified by slicing [79].

3.6. *Cohesion measurement*

Ott *et al* [18, 94] use slicing to measure functional cohesion. They define data slices which are a combination of forward and backward slices. Such slices are computed for output parameters of functions and the amount of overlapping indicates weak or strong functional cohesion.

3.7. *Clone detection*

There exist two approaches that use dependence graphs and slicing to detect cloned (or duplicated) code. One approach [72] identifies similar code in programs based

on finding similar subgraphs in program dependence graphs. It therefore considers not only the syntactic structure of programs but also the data flow within (as an abstraction of the semantics).

A very similar approach is presented in [66]. Starting from *every* pair of matching nodes, they construct isomorphic subgraphs for ideal clones that can be replaced by function calls automatically. Different from the first approach, they use heuristics to choose between alternatives and visit every node only once during subgraph construction.

3.8. *Software conformance certification*

Slicing can answer the question "*which statements have an influence on statement X?*", but slicing cannot answer the question "*why statement Y influences statement X*". Path conditions have been introduced by Snelting [104] as a way to validate software in measurement systems. Path conditions represent precise and necessary conditions for information flow between two program points. Constraint solvers can solve path conditions for the program's input variables. Solved path conditions may act as witnesses for safety violations such as an information flow between two program points of incompatible security level [77, 100].

3.9. *Other applications*

Slicing is used in model construction to slice away irrelevant code; transition system models are only built for the reduced code [54]. Slicing is also used to decompose tasks in real-time systems [40]. It has even been used for debugging and testing spreadsheets [96] or type checking programs [108].

4. Problems and Challenges in Slicing

Despite the fact that the basic principles of program slicing are fairly easy, the adaptation and implementation for real programming languages cause a series of challenges. Most implementations have been done for *C* or *C*-like languages. The following will present some general problems, which are slicing of programs with unstructured control flow, interprocedural, i.e., context-sensitive slicing, and slicing of concurrent programs.

4.1. *Unstructured control flow*

The original slicing algorithms assumed a language without jump statements. Such statements cause a small problem: Because the corresponding (correct) control flow graph (CFG) contains only a single outgoing edge, no other node can be control dependent on the jump statement, nor have these jump statements any outgoing data dependence. Therefore, a (backward) slice will never contain the jump statements. The exclusion of jump statements makes executable slices incorrect and

non-executable slices difficult to comprehend. Various approaches exist to address this problem: Ball and Horwitz [11] augment the CFG by adding *non-executable* edges between the jump statement and the immediate following statement. This makes both the statements after the jump target and the one following the jump statement control dependent on the jump. During data dependence computation, the non-executable edges are ignored. The resulting program dependence graph can be sliced with the usual slicing algorithm, except that labels (the jump targets) are included in the slice if a jump to that label is in the slice.

The algorithm of Ball and Horwitz is similar to Choi and Ferrante's first algorithm presented in [29]. Their second algorithm computes a normal slice first (which does not include the jump statements), and then adds goto statements until the slice is correct. This algorithm produces slices that are more precise than slices from the first algorithm, but the additional gotos may not be part of the original program, and the computed slices may not match the definition of a slice. Agrawal [2] presents an algorithm which also adds jump statements to a normal slice in order to make it correct. However, the added statements are always part of the original program. Harman and Danicic present another algorithm [50], which is an extension of Agrawal's.

Kumar and Horwitz [78] present an improved algorithm based on the augmented control flow graph, and a modified definition of control dependence using both the augmented and the non-augmented control flow graph. They also present a modified definition of a slice based on *semantic effects*, which basically inverts the definition of a slice: A statement x of program P has a semantic effect on a statement y, iff a program P' exists, created by modifying or removing x from P, and some input I exists such that P and P' halt on I and produce different values for some variables used at y. However, such a slice may be a superset of a Weiser slice because according to their definition, a statement like "`x := x;`" has a semantic effect.

4.2. *Interprocedural slicing*

The problem within interprocedural slicing is the calling context. If the calling context is not obeyed, the computed slices will not be accurate. Consider the example in Fig. 3, which is the interprocedural version of the example from Fig. 1. When we compute the slice for variable `p` in line 10 again, without obeying the calling context, we are not able to delete a single line. Line 8 clearly has an impact, because variable `i` is increased. Thus, lines 11–13 have an impact, too, and must be included. But lines 11–13 also have an impact on line 6, which therefore also has to be included. As `s` is used there, line 3 must be included, too. However, if we use the calling context, we are able to distinguish the calls in lines 6 and 8. The call in line 6 only impacts line 6 and the call in line 8 only line 8. Thus, it is possible to omit lines 6 and 3 from the slice.

The extension of the PDG for *interprocedural programs* introduces more nodes and edges: For every procedure a *procedure dependence graph* is constructed, which

```
 1   read(n)
 2   i := 1
 3   s := 0
 4   p := 1
 5   while (i <= n)
 6     s := add(s, i)
 7     p := mul(p, i)
 8     i := add(i, 1)
 9   write(s)
10   write(p)

11   proc add(x, y)
12     z := x + y
13     return z
14
15   proc mul(x, y)
16     z := x * y
17     return z
```

Fig. 3. An interprocedural program.

is basically a PDG with *formal-in* and *-out* nodes for every formal parameter of the procedure. A procedure call is represented by a *call* node and *actual-in* and *-out* nodes for each actual parameter. The call node is connected to the entry node by a *call* edge, the *actual-in* nodes are connected to their matching *formal-in* nodes via *parameter-in* edges and the *actual-out* nodes are connected to their matching *formal-out* nodes via *parameter-out* edges. Such a graph is called *Interprocedural Program Dependence Graph (IPDG)*. The *System Dependence Graph (SDG)* is an IPDG, where *summary edges* between actual-in and actual-out nodes have been added representing transitive dependence due to calls [60].

To slice programs with procedures, it is not enough to perform a reachability analysis on IPDGs or SDGs. The resulting slices are not accurate as the calling context is not preserved: The algorithm may traverse a parameter-in edge coming from a call site into a procedure, may traverse some edges there, and may finally traverse a parameter-out edge going to a different call site. The sequence of traversed edges (the path) is an *unrealizable path*: It is impossible for an execution that a called procedure does not return to its call site. We consider an interprocedural slice to be *precise* if all nodes included in the slice are reachable from the criterion by a *realizable* path.

The (*backward*) *slice* $S(n)$ of an IPDG $G = (N, E)$ at node $n \in N$ consists of all nodes on which n (transitively) depends via an interprocedurally realizable path:

$$S(n) = \{m \in N \mid m \rightarrow_{\mathsf{R}}^{\star} n\}.$$

Here, $m \rightarrow_{\mathsf{R}}^{\star} n$ denotes that there exists an interprocedurally realizable path from m to n.

These definitions cannot be used in an algorithm directly because it is impractical to check paths whether they are interprocedurally realizable. Accurate slices can be calculated with a modified algorithm on SDGs [60]: The benefit of SDGs is the presence of *summary* edges that represent transitive dependence due to calls. Summary edges can be used to identify actual-out nodes that are reachable from actual-in nodes by an interprocedurally realizable path through the called procedure

without analyzing it. The idea of the slicing algorithm using summary edges [60,97] is first to slice from the criterion only ascending into calling procedures, and then to slice from all visited nodes only descending into called procedures.

The algorithm to compute summary edges in [60] is based on attributed grammars. A more efficient algorithm was presented in [97]. This algorithm starts on an IPDG without summary edges and checks for all pairs of formal-in and -out nodes, if a path in the dependence graph between the nodes exists and corresponds to a same-level realizable (matched) path in the control flow graph. If such a path exists, a summary edge between the corresponding actual-in and -out node is inserted. Because the insertion of summary edges will make more paths possible, the search iterates until a minimal fixed point is found. The algorithm only follows data dependence, control dependence and summary edges. Therefore, any path consisting of these edges must correspond to a same-level realizable path. This algorithm is basically of cubic complexity [97].

System-dependence-graph based slices are not executable because different call sites may have different parameters included in the slice. Binkley [21] shows how to extend slicing in system dependence graphs to produce executable slices.

System dependence graphs assume procedures to be single-entry-single-exit, which does not hold in the presence of exceptions or interprocedural jumps. Sinha *et al* [103] present *interprocedural control dependence* which occurs if a procedure can be left abnormally by an embedded halt statement. They extend the IPDG with corresponding edges and show how to compute slices within the extended graph [102].

Meanwhile, extensive evaluations of slicing algorithms have been done. For control-flow-graph based Weiser-style algorithms some data can be found in [9,87]. A preliminary evaluation of program-dependence-based algorithms has been conducted by Agrawal and Guo [1]. That study has been shown to be flawed in [73], which also contains a large evaluation of various slicing algorithms. Another comprehensive study is presented in [20]. These evaluations have shown that context-insensitive slicing is very imprecise in comparison with context-sensitive slicing. This shows that context-sensitive slicing is highly preferable because the loss of precision is not acceptable. A surprising result is that the simple context-insensitive slicing is *slower* than the more complex context-sensitive slicing. The reason is that the context-sensitive algorithm has to visit far fewer nodes during traversal due to its higher precision.

The effect of the precision of the underlying data flow analysis and points-to analysis has been studied in a series of works [8,24,83,87,101].

4.3. *Object-oriented programs*

Slicing of object-oriented programs is more complex than slicing of procedural languages because of virtual functions and the distinction of classes and objects.

Usually, virtual functions are handled like function pointers in C programs. All approaches to slice object-oriented programs are based on dependence graphs. Larsen and Harrold [81] have introduced the class dependence graph, which introduces *class member* edges and represents the class' members as formal parameters of the methods. The main challenge in the representation of object-oriented programs as dependence graphs is the representation of objects themselves, and how they are passed as parameters to methods. Tonella *et al* [109] use flow-insensitive pointer analysis to resolve the runtime types of objects. Thus, their approach is able to distinguish data members for different objects. Liang and Harrold [82] have extended the system dependence graph by representing the passed objects as polymorphic trees of members. This C++ targeted approach turned out to be insufficient for Java programs, and Hammer and Snelting [49] presented an improved approach to slice Java programs.

4.4. *Slicing of concurrent programs*

Müller-Olm has shown that precise context-sensitive slicing of concurrent programs is undecidable in general [88]. Therefore, one has to use conservative approximations to analyze and slice concurrent programs. There are many variations of the program dependence graph for threaded programs like parallel program graphs [26, 27, 34]. Most approaches to static or dynamic slicing of threaded programs are based on such dependence graphs.

Dynamic slicing of threaded or *concurrent* programs has been approached by different authors [28,34,42,64,68] and is surveyed in [107]. Probably the first approach for *static* slicing of threaded programs was the work of Cheng [26,27,117]. He introduced some dependences, which are needed for a variant of the PDG, the *program dependence net* (PDN). His *selection* dependence is a special kind of control dependence, and his *synchronization* dependence is basically control dependence resulting from the previously presented communication dependence. Cheng's *communication dependence* is a combination of data dependence and the presented communication dependence. Cheng defines slices simply based on graph reachability. The resulting slices are not precise, as they do not take into account that dependences between concurrently executed statements are not transitive. Therefore, the integration of his technique of slicing threaded programs into slicing threaded object-oriented programs [117] has the same problem.

The first precise approach [71] to slice concurrent programs uses a `cobegin/coend` model of concurrency. That approach has been improved in [90].

Almost all approaches are context-insensitive and ignore calling context. The only context-sensitive approach [74] models the calling context explicitly. Every concurrently executing thread is represented by its own calling context; the program's calling context is then a tuple of call strings which is propagated along the edges in PDGs (one tuple element for each thread).

There is a series of works that use static slicing of concurrent programs but treat interference transitively and accept the imprecision: [53] present the semantics of a simple multi-threaded language that contains synchronization statements similar to the JVM. For this language, they introduce and define additional types of dependence: Divergence dependence, synchronization dependence and ready dependence. In [86] Cheng's approach is applied to slice Promela for model checking purposes.

5. Visualization of Slices

The computed slices and the program dependence graph itself are results that should be presented to the user if not used in following analyses. As graphical presentations are often more intuitive than textual ones, a graphical visualization is desirable. However, experience shows that the graphical presentation is less helpful than expected, and a textual presentation is superior.

This section describes approaches and experiences with both graphical and textual visualization.

5.1. *Graphical visualization*

ChopShop [62] was an early tool to visualize slices and chops, based on highlighting text (in emacs) or laying out graphs (with dot and ghostview). It is reported that even the smallest chops result in huge graphs. Therefore, only an abstraction is visualized: Normal statements (assignments) are omitted, procedure calls of the same procedure are folded into a single node and connecting edges are attributed with data dependence information.

The CANTO environment [7] has a visualization tool PROVIS based on dot which can visualize PDGs (besides other graphs). Again, problems with excessively large graphs are reported, which are omitted by only visualizing the subgraph which is reachable from a chosen node via a limited number of edges.

In [12], the same problems with visualizing dependence graphs are reported and a decomposition approach is presented: Groups of nodes are collapsed into one node. The result is a hierarchy of groups, where every group is visualized independently. Three different decompositions are presented: The first decomposition is to group the nodes belonging to the same procedure together, the second is to group the nodes belonging to the same loop and the third is a combination of both. The result of the function decomposition is identical to the visualization of the call graph.

The VALSOFT slicing system [71] visualizes the slices in program dependence graphs procedure-by-procedure. The user is able to fold sets of nodes and to navigate along dependence edges. To enable a better comprehension, slices are simultaneously highlighted in the graphical and the textual, source-based view. An example visualization can be seen in Fig. 4, where a slice is visualized in the dependence graph and in source code by highlighting.

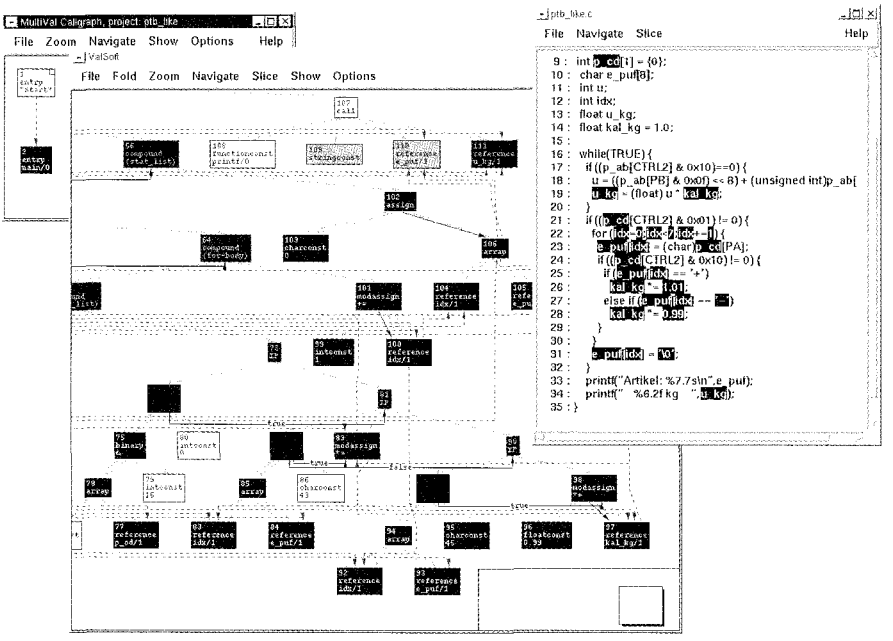

Fig. 4. The VALSOFT graphical user interface.

5.2. *Textual visualization*

Every slicing tool visualizes its results directly in the source code. However, most tools are line based, highlighting only complete lines. CodeSurfer [6] has textual visualization with highlighting parts of lines, if there is more than one statement in a line. The textual visualization includes graphical elements like pop-ups for visualization and navigation along e.g. data and control dependence or calls. Such aids are necessary as a user cannot identify relevant dependences easily from source text alone. Such problems have also been identified by Ernst [35] and he suggested similar graphical aids. However, his tool, which is not restricted to highlighting complete lines, does not have such aids and offers depth-limited slicing instead.

Steindl's slicer for Oberon [105] also highlights only parts of lines, based on the individual lexical elements of the program. The VALSOFT slicing system [71] is able to highlight unrestricted sets of PDG nodes in the corresponding source text.

SeeSlice [10] is a more advanced tool for visualizing slices. Files and procedures are not presented through source code but with an abstraction representing characters as single pixels. Files and procedures that are not part of computed slices are folded, such that only a small box is left. Slices highlight the pixels corresponding to contained elements.

CodeSurfer [6] also has a project viewer, which has a tree-like structural visualization of the SDG. This is useful for seeing "hidden" nodes, such as nodes that do not correspond to any source text.

The visualization of slices in a graphical or textual representation has not yet reached the expected usability, and more abstract visualizations are needed like the ones presented in [76]. Some of the abstractions need refinements that make the computes slices focused to the users' requirements. Such refinements will be presented in the next section.

6. Refinements

25 years have passed after the first introduction of program slicing. Usually, inventions in computer science are adopted widely after around 10 years. Why are slicing techniques not easily available yet? William Griswold gave a talk at PASTE 2001 [45] on that topic: *Making Slicing Practical: The Final Mile.* He pointed out why slicing is still not widely used today. One of the main problems is that slicing "as-it-stands" is inadequate for essential software-engineering needs. Usually, slices are hard to understand. This is partly due to bad user interfaces (which can be resolved by adequate visualization and navigation aids as presented in the previous section), but mainly related to the problem that slicing "dumps" the results onto the user without any explanation. Griswold stated the need for "slice explainers" that answer the question why a statement is included in the slice, as well as the need for "filtering". This section will present such "filtering" approaches to slicing.

6.1. *Distance-limited slicing*

One of the problems in understanding a slice for a criterion is to decide why a specific statement is included in that slice and how strong the influence of that statement is onto the criterion. A slice cannot answer these questions as it does not contain any qualitative information. Probably the most important attribute is *locality*: Users are more interested in facts that are near the current point of interest than on those far away. A simple but very useful aid is to provide the user with navigation along the dependences: For a selected statement, show all statements that are directly dependent (or vice versa). Such navigation is central to the VALSOFT system [77] or to CodeSurfer [6].

A more general approach presented in [76] accomplishes locality in slicing by limiting the length of a path between the criterion and the reached statement. Using paths in program dependence graphs has an advantage over paths in control flow graphs: A statement having a direct influence on the criterion will be reached by a path with the length one, independent of the textual or control flow distance.

6.2. *Chopping*

Slicing identifies statements in a program that may influence a given statement (the slicing criterion), but it cannot answer the question why a specific statement is part of a slice. A more focused approach can help: Jackson and Rollins [62] introduced *Chopping*, which reveals the statements involved in a transitive dependence from one

```
1
2
3
4   p := 1
5
6
7     p := p * i
8
9
10  write(p)
```

Fig. 5. A chop from (4, p) to (10, p).

specific statement (the source criterion) to another (the target criterion). Consider again the example from Fig. 1. Assume that we are interested in how the assignment to variable p in line 4 has impact on the output of p in line 10. The computed chop in Fig. 5 reveals the involved statements are only lines 4, 7 and 10.

Chopping can be formulated and computed using PDGs. A chop for a chopping criterion (s, t) is the set of nodes that are part of an influence of the (source) node s onto the (target) node t. This is basically the set of nodes that lie on a path from s to t in the PDG.

The *chop* $C(s, t)$ of an IPDG $G = (N, E)$ from the source criterion $s \in N$ to the target criterion $t \in N$ consists of all nodes on which node t (transitively) depends via an interprocedurally realizable path from node s to node t:

$$C(s, t) = \{n \in N \mid p = s \rightarrow_{\mathsf{R}}^{\star} t \wedge p = \langle n_1, \ldots, n_l \rangle \wedge \exists i\colon n = n_i\}.$$

Jackson and Rollins restricted s and t to be in the same procedure and only traversed control dependence, data dependence and summary edges but not parameter or call edges. The resulting chop is called a *truncated same-level chop*; "truncated" because nodes of called procedures are not included. In [98], Reps and Rosay presented more variants of precise chopping. A *non-truncated* same-level chop is like the truncated chop but includes the nodes of called procedures. They also present truncated and non-truncated *non-same-level* chops (which they call *interprocedural*), where the nodes of the chopping criterion are allowed to be in different procedures.

Some additional chopping algorithms are presented in [73], which also contains the first evaluation (Reps and Rosay [98] only report limited experience).

6.3. *Barrier slicing and chopping*

The presented slicing and chopping techniques compute very fixed results where the user has no influence. However, during slicing and chopping a user might want

to give additional restrictions or additional knowledge to the computation:

(1) A user might know that a certain data dependence cannot happen. Because the underlying data flow analysis is a conservative approximation, and the pointer analysis is imprecise, it might be clear to the user that a dependence found by the analysis cannot happen in reality. For example, the analysis assumes a dependence between a definition `a[i]=...` and a usage `...=a[j]` of an array, but the user discovers that `i` and `j` never have the same value. If such a dependence is removed from the dependence graph, the computed slice might be smaller.
(2) A user might want to exclude specific parts of the program that are of no interest for his purposes. For example, he might know that certain statement blocks are not executed during runs of interest; or he might want to ignore error handling or recovery code, when he is only interested in normal execution.
(3) During debugging, a slice might contain parts of the analyzed program that are known (or assumed) to be bug-free. These parts should be removed from the slice to make it more focused.

Both points have been tackled independently: For instance, the removal of dependences from the dependence graph by the user has been applied in Steindl's slicer [105]. The removal of parts from a slice has been presented by Lyle and Weiser [85] and is called *dicing*.

Another approach presented in [75] integrates both strategies into a new kind of slicing, called *barrier slicing*, where nodes (or edges) in the dependence graph are declared to be a *barrier* that transitive dependence is not allowed to pass. A barrier slice or chop is computed by not allowing the paths to contain nodes or edges of the barrier, because the presented approaches to compute slices and chops in dependence graphs all rely on realizable paths "$m \rightarrow_R^* n$" between nodes m and n.

6.4. *Dynamic slicing*

During debugging not all possible executions of a program are of interest. Usually, only one test case where a problem arises is in focus. Therefore, Korel and Laski have introduced *dynamic slicing* [69]: Dynamic slicing focuses on a single execution of a program instead of all possible executions like in Weiser's (static) slicing. The slicing criterion for a dynamic slice additionally specifies the (complete) input for the program. The criterion is now a triple $c = (I, s, V)$ consisting of the input I, a statement s and a subset V of the variables of the analyzed program. A *dynamic slice* $S(c)$ of a program P on a slicing criterion c is any executable program P', where

(1) P' is obtained by deleting zero or more statements from P,
(2) whenever P halts for the given input I, P' will halt for the input, and
(3) P' will compute the same values as P for the variables of V on input I.

Korel and Laski also presented an algorithm to compute dynamic slices based on the computation of sets of dynamic def-use relations (a variant of data dependences). However, the algorithm has been shown to be imprecise in [5]. The algorithm of Gopal [41], based on dynamic versions of information-flow relations [15], may compute non-terminating slices under certain loop conditions.

Agrawal and Horgan [5] have developed dynamic slicing on top of dependence graphs. They present four algorithms differing in precision and complexity:

(1) During execution of the program the nodes of the executed statements are marked. The approximate dynamic slice is then computed by doing a static slice on the node induced subgraph of the program's PDG.
(2) During execution, the dependence edges relating to the data and control dependences of the executed statements are marked. The approximate dynamic slice is done on the edge induced subgraph.
(3) For each execution of a statement a new node is generated. The data and control dependence of the executed statement to a certain execution of another statement generates an edge between the related node instances. The resulting graph is called *Dynamic Dependence Graph*. This algorithm is much more precise than the first two, however, its space requirement is much larger: In the worst case, it is equivalent to the amount of executed statements.
(4) To reduce the space requirement of the third algorithm, a reduction can be used that merges nodes with the same transitive dependences, resulting in a *Reduced Dynamic Dependence Graph*.

This work has been extended to *interprocedural* dynamic slicing in [3]. Kamkar's algorithms [63] are similar and focus on interprocedural dynamic slicing. An approach to interprocedural flowback analysis is presented in [28].

If the computed dynamic slices have to be executable, the use of unstructured control flow requires special treatment, like in static slicing. Algorithms for such circumstances have been developed by Korel [67] and Huynh [61].

To overcome the space requirements of dynamic dependence graphs, Goswami and Mall [43] have suggested to compute the graphs based on the paths between loop entries and exits. If the same path is traversed in a later iteration of the loop, the earlier execution of the path is ignored. Obviously, this produces incorrect dynamic slices: Consider a loop that contains two alternating paths A and B through the loop. Assume an execution where the loop is iterated at least four times, and the B path is executed at last; such an execution will be similar to $\ldots ABAB$. The suggested algorithm will now only consider AB as an execution and will omit dependences originating from statements in B going to statements in A.

A correct and efficient technique has been presented by Mund *et al* [89]: Their dynamic slicing algorithm uses the standard PDG where unstable edges are handled separately. Unstable edges are data dependence edges reaching the same target node, and additionally, their source nodes define the same variable. During execution the unstable edges are marked corresponding to which some of them has been

executed lately and the dynamic slice for each node is updated accordingly. Its space complexity is $O(n^2)$ where n is the number of statements in the program. The authors claim that the time complexity is also $O(n^2)$, which is clearly wrong as it must be dependent on the number of executed statements N. Thus, the time complexity must be at least $O(nN)$.

Beszédes *et al* [17] have developed a technique that computes dynamic slices for each executed statement during execution. They employ a special representation, which combines control and data dependence. During execution, the presented algorithm keeps track of the points of the last definition for any used variable. An implementation of the algorithm is able to slice ANSI C including unstructured control flow and procedure calls.

Most dynamic slicing approaches require static analysis as a preprocessing phase where data and control dependences are computed and stored in an intermediate representation. In [116], this preprocessing phase is limited or even eliminated.

Dynamic slicing has been formalized based on natural semantics in [44]. This leads to a generic, language-independent dynamic slicing analysis, which can be instantiated for imperative, logic or functional languages.

Extensions for object-oriented programs are straightforward. Ohata *et al* [92] includes lightweight tracing information like tracing procedure calls [91]. Such an approach, called *hybrid slicing*, has been presented earlier by Gupta and Soffa [47].

Extensions to dynamic slicing for concurrent programs have been presented in [27, 34, 42, 64, 68].

Hall [48] has presented another form of dynamic slicing: *Simultaneous dynamic slicing*, where a dynamic slice for a set of inputs is computed. This is not just the union of the dynamic slices for each of the inputs, as the union may not result in a correct dynamic slice for an input [33]. Beszédes [16] uses unions of dynamic slices to approximate the minimal static slice.

Between static and dynamic slicing lies *quasi static slicing*, presented by Venkatesh [112], where only some of the input variables have a fixed value (as specified in the slicing criterion) and the others may vary like in static slicing. A generalization is *conditioned slicing* [25,31]: The input variables are constrained by first order logic formula. A similar approach is *parametric program slicing*, presented by Field *et al* [37].

Surveys on dynamic slicing approaches can be found in [70, 107].

6.5. *Other refinements*

The SeeSlice slicing tool [10] already included some of the presented focusing and visualization techniques, e.g. the distance-limited slicing, visualizing distances, etc. The slicer of the CANTO environment [7] can be used in a stepping mode which is similar to distance-limited slicing: At each step the slice grows by considering one step of data or control dependence.

A *decomposition slice* [39] is basically a slice for a variable at all statements writing that variable. The decomposition slice is used to form a graph using the

partial ordering induced by proper subset inclusion of the decomposition slices for all variables. Beck and Eichmann [14] use slicing to isolate statements of a module that influence an exported behavior. Their work uses *interface dependence graphs* and *interface slicing*.

Set operations on slices produce various variants: Chopping uses intersection of a backward and a forward slice. The intersection of two forward or two backward slices is called a *backbone slice*. Dicing [85] is the subtraction of two slices. However, set operations on slices need special attention because the union of two slices may not produce a valid slice [33].

Orso *et al* [93] present a slicing algorithm which augments edges with types, and restricts reachability onto a set of types, creating slices restricted to these types. Their algorithm needs to compute the summary edges specific to each slice. However, it only works for programs without recursion.

In *amorphous program slicing* [51], a slice is generated by any simplifying transformation, not only by statement deletion. Binkley [23] has shown how to compute amorphous slices with dependence graphs.

7. Available Systems

Most of the slicing systems had been research prototypes and were not available or vanished in the last years. Some comparative studies have been done to subsets of formerly available slicers [56]. None of these slicers is still available and to the author's knowledge, only the three following tools are available to the public.

7.1. *CodeSurfer*

The commercially available CodeSurfer [6] is the successor of the Wisconsin Program-Integration System. It is the most advanced, complete and stable slicing tool. Its primary target is program understanding of ANSI C programs.

It can visualize call graphs graphically; procedures are textually visualized. For better usability, other elements like variables or files can be browsed hierarchically. Programs can be sliced and chopped in various way. The main data structure is the system dependence graph of a program. CodeSurfer can be programmed using its scripting language (Scheme). The scripting has access to the complete dependence graphs through an API. Thus, CodeSurfer can be used as an infrastructure for other program analyses.

7.2. *Sprite*

Sprite is a slicing tool built on top of Icaria and Ponder [9,87]. Ponder is a language independent infrastructure to build tools for performing syntactic and semantic analyses of large software systems. Icaria is the language dependent component for ANSI C and contains the Sprite tool, which is able to slice ANSI C. It contains a textual visualization of slices and is able to do typical set operations on slices. The implemented slicing tool uses a Weiser-style algorithm via data flow analysis iterating over a control flow graph. All three tools are freely available.

7.3. *Unravel*

Unravel [84] is a prototype tool that can be used to evaluate ANSI C source code statically by using program slicing. In its target to evaluate high integrity software it is similar to the VALSOFT system [77]. However, it is limited to the computation of forward and backward slices, which can be combined using set operations. The implemented Weiser-style algorithm is based on data flow equations and control flow graphs.

References

1. G. Agrawal and L. Guo, "Evaluating explicitly context-sensitive program slicing", *Workshop on Program Analysis for Software Tools and Engineering* (2001) 6–12.
2. H. Agrawal, "On slicing programs with jump statements", *SIGPLAN Conference on Programming Language Design and Implementation* (1994) 302–312.
3. H. Agrawal, R. A. DeMillo and E. H. Spafford, "Dynamic slicing in the presence of unconstrained pointers", *Symposium on Testing, Analysis, and Verification* (1991) 60–73.
4. H. Agrawal, R. A. DeMillo and E. H. Spafford, "Debugging with dynamic slicing and backtracking", *Software, Practice and Experience* **23**, no. 6 (June 1993) 589–616.
5. H. Agrawal and J. R. Horgan, "Dynamic program slicing", *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (1990) 246–256.
6. P. Anderson and T. Teitelbaum, "Software inspection using codesurfer", *Workshop on Inspection in Software Engineering (CAV 2001)*, 2001.
7. G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei and E. Merlo, "Program understanding and maintenance with the CANTO environment", *International Conference on Software Maintenance* (1997) 72–81.
8. D. C. Atkinson and W. G. Griswold, "Effective whole-program analysis in the presence of pointers", *Foundations of Software Engineering* (1998) 46–55.
9. D. C. Atkinson and W. G. Griswold, "Implementation techniques for efficient data-flow analysis of large programs", *Proceedings of the International Conference on Software Maintenance* (2001) 52–61.
10. T. Ball and S. G. Eick, "Visualizing program slices", *IEEE Symposium on Visual Languages* (1994) 288–295.
11. T. Ball and S. Horwitz, "Slicing programs with arbitrary control-flow", *Automated and Algorithmic Debugging* (1993) 206–222.
12. F. Balmas, "Displaying dependence graphs: A hierarchical approach", *Proceedings of the Eighth Working Conference on Reverse Engineering* (2001) 261–270.
13. S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs", *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1993) 384–396.
14. J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering", *IEEE/ACM 15th Conference on Software Engineering (ICSE'93)* (1993) 509–518.
15. J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs", *ACM Transactions on Programming Languages and Systems* **7**, no. 1 (January 1985) 37–61.
16. Á. Beszédes, C. Faragó, Z. M. Szabó, J. Csirik and T. Gyimóthy, "Union slices for program maintenance", *International Conference on Software Maintenance (ICSM'02)* (2002) 12–21.

17. Á. Beszédes, T. Gergely, Z. M. Szabó, J. Csirik and T. Gyimothy, "Dynamic slicing method for maintenance of large C programs", *Proceedings of the Fifth Conference on Software Maintenance and Reengineering, CSMR 2001* (2001) 105–113.
18. J. M. Bieman and L. M. Ott, "Measuring functional cohesion", *IEEE Transactions on Software Engineering* **20**, no. 8 (August 1994) 644–657.
19. D. Binkley, "Using semantic differencing to reduce the cost of regression testing", *Proceedings of the International Conference on Software Maintenance* (1992) 41–50.
20. D. Binkley and M. Harman, "A large-scale empirical study of forward and backward static slice size and context sensitivity", *International Conference on Software Maintenance* (2003) 44–53.
21. D. Binkley, "Precise executable interprocedural slices", *ACM Letters on Programming Languages and Systems* **2**, no. 1–4 (1993) 31–45.
22. D. Binkley, "The application of program slicing to regression testing", *Information and Software Technology* **40**, no. 11–12 (1998) 583–594.
23. D. Binkley, "Computing amorphous program slices using dependence graphs and a data-flow model", *ACM Symposium on Applied Computing* (1999) 519–525.
24. D. W. Binkley and J. R. Lyle, "Application of the pointer state subgraph to static program slicing", *The Journal of Systems and Software* (1998) 17–27.
25. G. Canfora, A. Cimitile and A. De Lucia. "Conditioned program slicing", *Information and Software Technology* **40**, no. 11–12 (1998) 595–607.
26. J. Cheng. "Dependence analysis of parallel and distributed programs and its applications", *International Conference on Advances in Parallel and Distributed Computing*, 1997.
27. J. Cheng, "Slicing concurrent programs", *Automated and Algorithmic Debugging, 1st International Workshop, AADEBUG'93, Lecture Notes in Computer Science 749* (Springer, 1993) 223–240.
28. J.-D. Choi, B. Miller and R. Netzer, "Techniques for debugging parallel programs with flowback analysis", *ACM Transactions on Programming Languages and Systems* **13**, no. 4 (October 1991) 491–530.
29. J.-D. Choi and J. Ferrante, "Static slicing in the presence of goto statements", *ACM Transactions on Programming Languages and Systems* **16**, no. 4 (1994) 1097–1113.
30. E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar and T. Teitelbaum, "Program slicing of hardware description languages", *Conference on Correct Hardware Design and Verification Methods* (1999) 298–312.
31. S. Danicic, C. Fox, M. Harman and R. Hierons, "Consit: A conditioned program slicer", *International Conference on Software Maintenance* (2000) 216–226.
32. A. De Lucia, "Program slicing: Methods and applications", *IEEE Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001, Invited paper.
33. A. De Lucia, M. Harman, R. Hierons and J. Krinke, "Unions of slices are not slices", *7th European Conference on Software Maintenance and Reengineering*, 2003.
34. E. Duesterwald, R. Gupta and M. L. Soffa, "Distributed slicing and partial reexecution for distributed programs", *5th Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science 757* (Springer, 1992) 497–511.
35. M. D. Ernst, "Practical fine-grained static slicing of optimized code", Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 1994.
36. J. Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems* **9**, no. 3 (July 1987) 319–349.

37. J. Field, G. Ramalingam and F. Tip, "Parametric program slicing", *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages* (1995) 379–392.
38. M. A. Francel and S. Rugaber, "The value of slicing while debugging", *Proceedings of the 7th International Workshop on Program Comprehension* (2001) 151–169.
39. K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance", *IEEE Transactions on Software Engineering* **17**, no. 8 (1991) 751–761.
40. R. Gerber and S. Hong, "Slicing real-time programs for enhanced schedulability", *ACM Transactions on Programming Languages and Systems* **13**, no. 3 (1997) 525–555.
41. R. Gopal, "Dynamic program slicing based on dependence relations", *Conference on Software Maintenance* (1991) 191–200.
42. D. Goswami and R. Mall, "Dynamic slicing of concurrent programs," *High Performance Computing — HiPC 2000, 7th International Conference, Lecture Notes in Computer Science 1970* (2000) 15–26.
43. D. Goswami and R. Mall, "An efficient method for computing dynamic program slices", *Information Processing Letters* (2002) 111–117.
44. V. Gouranton and D. Le Metayer, "Dynamic slicing: A generic analysis based on a natural semantics format", *Journal of Logic and Computation* **9**, no. 6 (1999) 835–871.
45. W. G. Griswold, "Making slicing practical: The final mile", 2001. Invited Talk, PASTE'01.
46. R. Gupta, M. J. Harrold and M. L. Soffa, "An approach to regression testing using slicing", *Proceedings of the IEEE Conference on Software Maintenance* (1992) 299–308.
47. R. Gupta and M. L. Soffa, "Hybrid slicing: An approach for refining static slices using dynamic information", *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering* (1995) 29–40.
48. R. J. Hall, "Automatic extraction of executable program subsets by simultaneous dynamic program slicing", *Automated Software Engineering* **2**, no. 1 (March 1995) 33–53.
49. C. Hammer and G. Snelting, "An improved slicer for Java", *Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, 2004.
50. M. Harman and S. Danicic, "A new algorithm for slicing unstructured programs", *Journal of Software Maintenance* **10**, no. 6 (1998) 415–441.
51. M. Harman and S. Danicic, "Amorphous program slicing", *5th IEEE International Workshop on Program Comprehension (IWPC'97)* (1997) 70–79.
52. M. Harman and K. B. Gallagher, "Program slicing", *Information and Software Technology* **40**, no. 11–12 (1998) 577–581.
53. J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski and H. Zheng, "A formal study of slicing for multi-threaded programs with JVM concurrency primitives", *Static Analysis Symposium, Lecture Notes in Computer Science 1694* (Springer, 1999) 1–18.
54. J. Hatcliff, M. B. Dwyer and H. Zheng, "Slicing software for model construction", *Higher-Order and Symbolic Computation* **13**, no. 4 (2000) 315–353.
55. M. P. E. Heimdahl and M. W. Whalen, "Reduction and slicing of hierarchical state machines", *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)* (1997) 450–467.
56. T. Hoffner, M. Kamkar and P. Fritzson, "Evaluation of program slicing tools", *2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)* (1995) 51–69.

57. S. Horwitz and T. Reps, "Efficient comparison of program slices", *Acta Informatica* **28** (1991) 713–732.
58. S. Horwitz, "Identifying the semantic and textual differences between two versions of a program", *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (1990) 234–245.
59. S. Horwitz, J. Prins and T. Reps, "Integrating noninterfering versions of programs", *ACM Transactions on Programming Languages and Systems* **11**, no. 4 (July 1989) 345–387.
60. S. B. Horwitz, T. W. Reps and D. Binkley, "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems* **12**, no. 1 (January 1990) 26–60.
61. D. Huynh and Y. Song, "Forward computation of dynamic slicing in the presence of structured jump statements", *Proceedings of ISACC'97* (1997) 73–81.
62. D. Jackson and E. J. Rollins, "A new model of program dependences for reverse engineering", *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering* (1994) 2–10.
63. M. Kamkar, P. Fritzson and N. Shahmerhi, "Three approaches to interprocedural dynamic slicing", *Microprocessing and Microprogramming* **38** (1993) 625–636.
64. M. Kamkar and P. Krajina, "Dynamic slicing of distributed programs", *International Conference on Software Maintenance* (1995) 222–231.
65. A. Kiss, J. Jasz, G. Lehotai and T. Gymothy, "Interprocedural slicing of binary executables", *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation* (2003) 118–127.
66. R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code", *Eighth International Static Analysis Symposium (SAS), Lecture Notes in Computer Science 2126* (2001).
67. B. Korel, "Computation of dynamic slices for unstructured programs", *IEEE Transactions on Software Engineering* **23**, no. 1 (1997) 17–34.
68. B. Korel and R. Ferguson, "Dynamic slicing of distributed programs", *Applied Mathematics and Computer Science Journal* **2**, no. 2 (1992) 199–215.
69. B. Korel and J. Laski, "Dynamic program slicing", *Information Processing Letters* **29**, no. 3 (October 1988) 155–163.
70. B. Korel and J. Rilling, "Dynamic program slicing methods", *Information and Software Technology* **40**, no. 11–12 (1998) 647–659.
71. J. Krinke, "Static slicing of threaded programs", *Proceedings of the ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)* (ACM Press, 1998) 35–42, ACM SIGPLAN Notices 33(7).
72. J. Krinke, "Identifying similar code with program dependence graphs", *Proceedings of the Eighth Working Conference on Reverse Engineering* (2001) 301–309.
73. J. Krinke, "Evaluating context-sensitive slicing and chopping", *Proceedings of the International Conference on Software Maintenance* (2002) 22–31.
74. J. Krinke, "Context-sensitive slicing of concurrent programs", *Proceedings of the ESEC/FSE* (2003) 178–187.
75. J. Krinke, "Slicing, chopping, and path conditions with barriers", *Software Quality Journal* **12**, no. 4 (December 2004) 339–360.
76. J. Krinke, "Visualization of program dependence and slices", *Proceedings of the International Conference on Software Maintenance* (2004) 168–177.
77. J. Krinke and G. Snelting, "Validation of measurement software as an application of slicing and constraint solving", *Information and Software Technology* **40**, no. 11–12 (December 1998) 661–675.

78. S. Kumar and S. Horwitz, "Better slicing of programs with jumps and switches", *Proceedings of FASE 2002: Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science 2306* (Springer, 2002) 96–112.
79. A. Lakhotia and J.-C. Deprez, "Restructuring programs by tucking statements into functions", *Information and Software Technology* **40**, no. 11–12 (1998) 677–690.
80. F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph-based program slicing", *IEEE Transactions on Software Engineering* **23**, no. 4 (April 1997) 246–259.
81. L. Larsen and M. J. Harrold, "Slicing object-oriented software", *18th International Conference on Software Engineering* (1996) 495–505.
82. D. Liang and M. J. Harrold, "Slicing objects using system dependence graphs", *Proceedings of the International Conference on Software Maintanence* (1998) 358–367.
83. D. Liang and M. J. Harrold, "Efficient points-to analysis for whole-program analysis", *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Foundations of Software Engineering* (1999) 199–215.
84. J. Lyle and D. Wallace, "Using the unravel program slicing tool to evaluate high integrity software", *Proceedings of Software Quality Week*, 1997.
85. J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing", *2nd International Conference on Computers and Applications* (1987) 877–882.
86. L. I. Millett and T. Teitelbaum, "Issues in slicing promela and its applications to model checking, protocol understanding, and simulation", *International Journal on Software Tools for Technology Transfer* **2**, no. 4 (2000) 343–349.
87. M. Mock, D. C. Atkinson, C. Chambers and S. J. Eggers, "Improving program slicing with dynamic points-to data", *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, 2002.
88. M. Müller-Olm and H. Seidl, "On optimal slicing of parallel programs", *STOC 2001 (33th ACM Symposium on Theory of Computing)* (2001) 647–656.
89. G. B. Mund, R. Mall and S. Sarkar, "An efficient dynamic program slicing technique", *Information and Software Technology* **44**, no. 2 (2002) 123–132.
90. M. G. Nanda and S. Ramesh, "Slicing concurrent programs", *International Conference on Software Testing and Analysis (ISSTA 2000)* (2000) 180–190.
91. A. Nishimatsu, M. Jihira, S. Kusumoto and K. Inoue, "Call-mark slicing: An efficient and economical way of reducing slice", *International Conference of Software Engineering* (1999) 422–431.
92. F. Ohata, K. Hirose, M. Fujii and K. Inoue, "A slicing method for object-oriented programs using lightweight dynamic information", *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, 2001.
93. A. Orso, S. Sinha and M. J. Harrold, "Incremental slicing based on data-dependences types", *International Conference on Software Maintenance*, 2001.
94. L. M. Ott and J. M. Bieman, "Program slices as an abstraction for cohesion measurement", *Information and Software Technology* **40**, no. 11–12 (1998) 691–700.
95. K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices* **19**, no. 5 (1984) 177–184.
96. J. Reichwein, G. Rothermel and M. M. Burnett, "Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging", *Conference on Domain Specific Languages* (1999) 25–38.

97. T. Reps, S. Horwitz, M. Sagiv and G. Rosay, "Speeding up slicing", *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering* (1994) 11–20.
98. T. Reps and G. Rosay, "Precise interprocedural chopping", *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering* (1995) 41–52.
99. F. Ricca and P. Tonella, "Web application slicing", *International Conference on Software Maintenance* (2001) 148–157.
100. T. Robschink and G. Snelting, "Efficient path conditions in dependence graphs", *Proceedings of the 24th International Conference of Software Engineering (ICSE)* (2002) 478–488.
101. M. Shapiro and S. Horwitz, "The effects of the precision of pointer analysis", *Proceedings from the 4th International Static Analysis Symposium, Lecture Notes in Computer Science 1302* (1997) 16–34.
102. S. Sinha, M. J. Harrold and G. Rothermel, "System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow", *International Conference on Software Engineering* (1999) 432–441.
103. S. Sinha, M. J. Harrold and G. Rothermel, "Interprocedural control dependence", *ACM Transactions on Software Engineering and Methodology* **10**, no. 2 (2001) 209–254.
104. G. Snelting, "Combining slicing and constraint solving for validation of measurement software", *Static Analysis Symposium, Lecture Notes in Computer Science 1145* (Springer, 1996) 332–348.
105. C. Steindl, "Benefits of a data flow-aware programming environment", *Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, 1999.
106. F. Tip, J-D Choi, J. Field and G. Ramalingam, "Slicing class hierarchies in C++", *Conference on Object-oriented Programming Systems, Languages and Applications* (1996) 179–197.
107. F. Tip, "A survey of program slicing techniques', *Journal of Programming Languages* **3**, no. 3 (September 1995).
108. F. Tip and T. B. Dinesh, "A slicing-based approach for locating type errors", *ACM Transactions on Software Engineering and Methodology* **10**, no. 1 (January 2001) 5–55.
109. P. Tonella, G. Antoniol, R. Fiutem and E. Merlo, "Flow insensitive C++ pointers and polymorphism analysis and its application to slicing", *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)* (1997) 433–444.
110. W. W. Vasconcelos, "A flexible framework for dynamic and static slicing of logic programs", *Proceedings of PADL'99, Lecture Notes in Computers Science 1551* (1999) 259–274.
111. W. W. Vasconcelos and M. A. T. Aragao, "Slicing knowledge-based systems: Techniques and applications", *Knowledge Based Systems* **13**, no. 4 (2000).
112. G. A. Venkatesh, "The semantic approach to program slicing", *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (1991) 26–28.
113. M. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method", PhD Thesis, University of Michigan, Ann Arbor (1979).
114. M. Weiser, "Programmers use slices when debugging", *Communications of the ACM* **25**, no. 7 (1982) 446–452.
115. M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering* **10**, no. 4 (July 1984) 352–357.

116. X. Zhang, R. Gupta and Y. Zhang, "Precise dynamic slicing algorithms", *IEEE/ACM International Conference on Software Engineering (ICSE)* (2003) 319–329.
117. J. Zhao, J. Cheng and K. Ushijima, "Static slicing of concurrent object-oriented programs", *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference* (1996) 312–320.