CS 589

HANDOUT #5

Static Analysis

1. Encyclopedia of Software Engineering, Wiley.
2. P. Jalote, An Integrated Approach to Software Engineering, Springer.
3. I. Sommerville, Software Engineering, Addison-Wesley.
4. B. Korel, The program dependence graph in static program testing, IPL.

Introduction

Static Analysis. Testing evaluates the quality of software. Testing is sometimes distinguished into static and dynamic analysis. Static techniques do not exercise the software, they only examine it, but they are considered part of testing because they do evaluate software quality. Dynamic testing techniques exercise the software using sample input values.

Static analysis techniques grew out of compiler technology, and many compilers have static analysis features available (surprisingly often unused). Commercial tools are also available which can carry out extensive static analysis on software code without executing it. A significant number of programming errors are statically detectable (Hatton, 1993). Static analysis examines control flow and data flow, and can discover "dead code", infinite loops, uninitialized variables, unused data values, standards violations, etc. Various complexity metrics can also be computed.

Static analysis does not replace dynamic analysis, but is a very useful quality check normally carried out before dynamic test execution begins.

Static Analysis Tools. Many of the errors which are made in software development are "trivial", i.e., they are very easy for people to make. For example, spelling a variable name slightly differently in two places, omitting a punctuation mark, forgetting to read in the 14th variable, etc. However, such errors which are trivial to make often have severe and non-trivial effects on the software. A misspelled word in a printed output is a mere annoyance, but using the wrong variable name to compute how much oxygen to give a chemical reaction can have an explosive effect (literally).

One software tool which filters out many of these types of error is a compiler. If variables are pre-declared, for example, misspellings are found. However, there are a number of other things which a compiler may not detect: for example, whether you have forgotten to assign a value to a variable before using it, or if a declared procedure is actually never called.

Static analysis tools have developed out of common research, and can find a number of errors in software without dynamically executing that software. They can find all occurrences of some types of error, so they are 100% accurate for some things. They are not a substitute for testing, however, as there are other errors which are not found by static analysis (the software might perfectly perform the wrong function).

Static analysis tools perform several types of analysis, as detailed below. Not all tools perform all of the functions listed; a selection of types of facilities is given.

Complexity analysis computes the cyclomatic complexity, the measurement of the number of decisions in the module, and may also compute other complexity measures. Static analysis tools can also give a measure of the "structured-ness" of the code; if the code contains transfers of control (jumps) which cross other transfers, this indicates poor program structure (usually due to improper use of "go to"s).

Data flow analysis detects errors and anomalies of variable usage. For example, if a variable is used in an expression before a value has been assigned to it, or if it is assigned a value which is then never used.

Control flow analysis detects logic errors, such as program code which is never executed, or loops which cannot be exited (infinite loops).

Information flow analysis examines the relationship of input to output variables, and can tell which input values affect a given output value, or which output values are affected by a given input value. This can be used to verify (manually) that the correct dependencies between input and output values have been achieved. A partial program can be extracted, listing only the statements which use or are affected by given variables.

Static analysis tools may also provide other functions to support formal verification techniques such as correctness proving.

Static Analysis

Analysis of programs by methodically analyzing the program text is called *static analysis*. Static analysis is usually performed mechanically by the aid of software tools. During static analysis the program itself is not executed, but the program text is the input to the tools. The aim of the static analysis tools is to detect errors or potential errors or to generate information about the structure of the program that can be useful for documentation or understanding of the program. Different kinds of static analysis tools can be designed to perform different types of analyses.

Many compilers perform some limited static analysis. More often, tools explicitly for static analysis are used. Static analysis can be very useful for exposing errors that may escape other techniques. As the analysis is performed with the help of software tools, static analysis is a very cost-effective way of discovering errors. An advantage is that static analysis sometimes detects the errors themselves, not just the presence of errors, as in testing. This saves the effort of tracing the error from the data that reveals the presence of errors. Furthermore, static analysis can provide "warnings" against potential errors and can provide insight into the structure of the program. It is also useful for determining violations of local programming standards, which the standard compilers will be unable to detect. Extensive static analysis can considerably reduce the effort later needed during testing.

Data flow analysis [FO78] is one form of static analysis that concentrates on the uses of data by programs and detects some data flow anomalies. Data flow anomalies are "suspicious" use of data in a program. In general, data flow anomalies are technically not errors, and they may go undetected by the compiler. However, they are often a symptom of an error, caused due to carelessness in typing or error in coding. At the very least, presence of data flow anomalies implies poor coding. Hence, if a program has data flow anomalies, it is a cause of concern, which should be properly addressed.

[FO78] L. Fosdick, L. Osterweil, Dataflow analysis in software reliability, ACM Computing Surveys, 8(3), 1978.

[Hatton 93] L. Hatton, "Automated Static Analysis," 2nd Intern. Software Testing Analysis and Review Conf., STAR93, 1993.

```
x = a;
  :
x does not appear in any right hand side
  :
x = b;
```

FIGURE 8.2. A code segment.

An example of the data flow anomaly is the *live variable problem*, in which a variable is assigned some value but then the variable is not used in any later computation. Such a live variable and assignment to the variable are clearly redundant. Another simple example of this is having two assignments to a variable without using the value of the variable between the two assignments. In this case the first assignment is redundant. For example, consider the simple case of the code segment shown in Figure 8.2.

Clearly, the first assignment statement is useless. The question is why is that statement in the program? Perhaps the programmer meant to say `y := b` in the second statement, and mistyped `y` as `x`. In that case, detecting this anomaly and directing the programmer's attention to it can save considerable effort in testing and debugging.

In addition to revealing anomalies, data flow analysis can provide valuable information for documentation of programs. For example, data flow analysis can provide information about which variables are modified on invoking a procedure in the caller program and the value of the variables used in the called procedure (this can also be used to make sure that the interface of the procedure is minimum, resulting in lower coupling). This analysis can identify aliasing, which occurs when different variables represent the same data object. This information can be useful during maintenance to ensure that there are no undesirable side effects of some modifications to a procedure.

Other examples of data flow anomalies are *unreachable code, unused variables,* and *unreferenced labels.* Unreachable code is that part of the code to which there is not a feasible path; there is no possible execution in which it can be executed. Technically this is not an error, and a compiler will at most generate a warning. The program behavior during execution may also be consistent with its specifications. However, often the presence of unreachable code is a sign of lack of proper understanding of the program by the programmer (otherwise why would a programmer leave the unreachable code), which suggests that the presence of error's is likely. Often, unreachable code comes into existence when an existing program is modified. In that situation unreachable code may signify undesired or unexpected side effects of the modifications. Unreferenced labels and unused variables are like unreachable code in that they are technically not errors, but often are symptoms of errors; thus their presence often implies the presence of errors.

Data flow analysis is usually performed by representing a program as a graph, sometimes called the flow graph. The nodes in a flow graph represent statements of a program, while the edges represent control paths from one statement to another. Correspondence between the nodes and statements is maintained, and the graph is analyzed to determine different relationships between the statements. By use of different algorithms, different kinds of anomalies can be detected. Many of the algorithms to detect anomalies can be quite complex and require a lot of processing time. For example, the execution time of algorithms to detect unreachable code increases with the square of the number of nodes in the graph. Consequently, this analysis is often limited to modules or to a collection of some modules and is rarely performed on complete systems.

To reduce processing times of algorithms, the search of a flow graph has to be carefully organized. Another way to reduce the time for executing algorithms is to reduce the size of the flow graph. Flow graphs can get extremely large for large programs, and transformations are often performed on the flow graph to reduce their size. The most common transformation is to have each node represent a sequence of contiguous statements that have no branches in them, thus representing a block of code that will be executed together. Another transformation often done is to have each node represent a procedure or function. In that case, the resulting graph is often called the *call graph*, in which an edge from one node n to another node m represents the fact that the execution of the module represented by n directly invokes the module m.

Other Uses of Static Analysis

We have seen that data flow analysis is a technique for statically analyzing a program to reveal some types of anomalies. Other forms of static analysis to detect different errors and anomalies can also be performed. Here we list some of the other common uses of static analysis tools.

An error often made, especially when different teams are developing different parts of the software, is mismatched parameter lists, where the argument list of a module invocation is different in number or type from the parameters of the invoked module. This can be detected by a compiler if no separate compilation is allowed and the entire program text is available to the compiler (as is the case with standard Pascal). However, if the programs are separately developed and compiled, which is almost always the case with large software developments, this error will not be detected. A static analyzer with access to the different parts of the program can easily detect this error. Such errors can also be detected during code review, but it is more economical to do it mechanically. An extension of this is to detect calls to nonexistent program modules. Essentially, the interfacing of different modules, developed and compiled separately, can be checked for mutual consistency easily through static analysis. In some limited cases, static analysis can also detect infinite or potentially infinite loops, and illegal recursion (e.g., no termination condition for recursion).

There are different kinds of documents that static analyzers can produce, which can be useful for maintenance or increased understanding of the program. The first is the cross-reference of where different variables and constants are used. Often, looking at the cross-reference can help one detect subtle errors, like many constants defined (with perhaps somewhat different values) to represent the same entity. For example, the value of pi could be defined as constant in different routines with slightly different values. A report with cross-references can be useful to detect such errors. To reduce the size of such reports, it may be more useful to limit it to the use of constants and global variables.

Information about the frequency of use of different constructs of the programming language can also be obtained by static analyses. Such information is useful for statistical analysis of programs, such as what types of modules are more prone to defect. Another use is to evaluate the complexity. There are some complexity measures that are a function of the frequency of occurrence of different types of statements. To determine complexity from such measures, this information can be useful.

Static analysis can also produce the structure chart of programs. The actual structure chart of a system is a useful documentation aid. It can also be used to determine the changes made in design during the coding phase by comparing it to the structure chart produced during system design. A static nesting hierarchy of procedures can also be easily produced by static analysis.

There are some coding restrictions that the programming language imposes. However, different organizations may have further restrictions on the use of different language features for reliability, portability, or efficiency reasons. Examples of these include mixed type arithmetic, type conversion, using features that are machine-dependent, and too many gotos. Such restrictions cannot be checked by the compiler, but static analysis can be used to enforce these standards. Such violations can also be checked in code review, but it is more efficient and economical to let a program do this checking.

Static program analysers

Static program analysers are software tools which scan the source text of a program and detect possible faults and anomalies. They do not require the program to be executed. They may be used as part of the verification process to complement the error detection facilities provided by the language compiler.

The intention of automatic static analysis is to draw the verifier's attention to anomalies in the program such as variables which are used without initialization, variables which are unused, and so on. While these are not necessarily erroneous conditions, it is probable that many of these anomalies are a result of errors of commission or omission. Some of the checks which can be detected by static analysis are shown in Figure 24.7.

Unreachable code
Unconditional branches into loops
Undeclared variables
Variables used before initialization
Variables declared and never used
Variables written twice with no intervening assignment
Parameter type mismatches
Parameter number mismatches
Uncalled functions and procedures
Non-usage of function results
Possible array bound violations
Misuse of pointers

Figure 24.7
Static analysis checks.

There are a number of stages of static analysis:

(1) *Control flow analysis* This stage identifies and highlights loops with multiple exit or entry points and unreachable code. Unreachable code is code which is surrounded by unconditional goto statements and which is not referenced elsewhere in the program. If goto statements are avoided, unreachable code cannot be written.

(2) *Data use analysis* This stage is concerned with examining how variables in the program are used. It detects variables which are used without previous initialization, variables which are written twice without an intervening assignment and variables which are declared but never used. Data use analysis can also detect ineffective tests where the test condition always has the same value.

(3) *Interface analysis* This analysis checks the consistency of routine and procedure declarations and their use. It is unnecessary if a strongly typed language like Ada is used for implementation but can detect type errors in weakly typed languages like Fortran and C. Interface analysis can also detect functions and procedures which are declared and never called or function results which are never used.

(4) *Information flow analysis* This phase of the analysis identifies all input variables on which output variables depend. While it does not detect anomalies, the derivation of the values used in the program are explicitly listed. Erroneous derivations should therefore be easier to detect during a code inspection or review. Information flow analysis can also show the conditions on which a variable's value depends.

(5) *Path analysis* This phase of semantic analysis identifies all possible paths through the program and sets out the statements executed as part of that path. It essentially unravels the program's control and allows each possible predicate to be analysed individually.

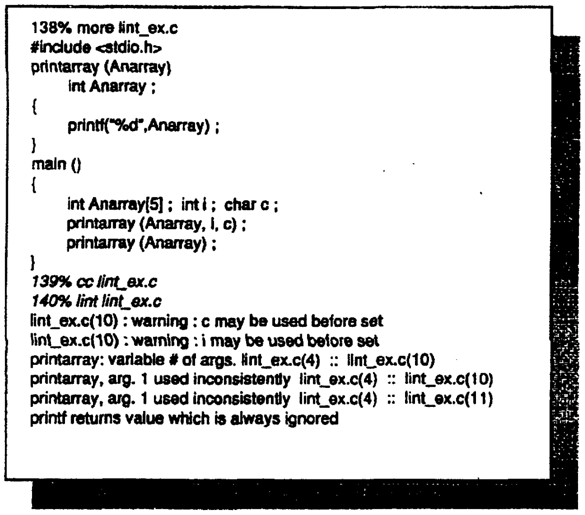
Information flow analysis and path analysis generate an immense amount of information. This information is really another way of viewing the program and does not highlight anomalous conditions. Because of the large amount of information generated, static analysis is therefore often restricted to those phases which can explicitly detect program anomalies.

Early static analysers such as DAVE (Osterweil and Fosdick, 1976), AUDIT (Culpepper, 1975) and FACES (Ramamoorthy and Ho, 1975) were developed for use with Fortran programs. They check subroutine interfaces to ensure that the number and types of subroutines parameters are consistent with the routine declaration, locate COMMON block errors and flag error-prone practices such as branching into a DO-loop. Fortran-based static analysers are particularly useful because Fortran compilers do not do much program checking and the language leaves much scope for misuse.

Osterweil, Fosdick, 1976, DAVE - a validation, error detection and documentation system for FORTRAN programs, Software Practice and Experience, 6, 473-86.

Culpepper, 1975, A system for reliable engineering software, IEEE Trans. Software Eng., 1 (2), 174-8.

Ramamoorthym, Ho, (1975), Testing large software with automated software evaluation system, IEEE Trans. on Software Engineering, 1 (1), 46-58.



```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
      int Anarray ;
{
      printf("%d",Anarray) ;
}
main ()
{
      int Anarray[5] ;  int i ;  char c ;
      printarray (Anarray, i, c) ;
      printarray (Anarray) ;
}
139% cc lint_ex.c
140% lint lint_ex.c
lint_ex.c(10) : warning : c may be used before set
lint_ex.c(10) : warning : i may be used before set
printarray: variable # of args. lint_ex.c(4)  ::  lint_ex.c(10)
printarray, arg. 1 used inconsistently  lint_ex.c(4)  ::  lint_ex.c(10)
printarray, arg. 1 used inconsistently  lint_ex.c(4)  ::  lint_ex.c(11)
printf returns value which is always ignored
```

Figure 24.8
LINT static analysis.

A static analyser for C programs, called LINT, is distributed as part of the UNIX system. LINT provides static checking equivalent to that provided by the compiler in a strongly-typed language such as Ada. The reliability advantages of a strictly typed language are combined with the efficient code generation of a systems implementation language. However, many C programmers don't always use LINT to check their programs before execution and hence spend time in unnecessary debugging.

An example of the output produced by LINT is shown in Figure 24.8. This is a transcript of a UNIX terminal session where commands are shown in italics. The first command lists the (nonsensical) program. It defines a function with one parameter called printarray then causes this function to be called with three parameters. Variables i and c are declared but never assigned values. The value returned by the function is never used.

The line numbered 139 shows the C compilation of this program with no errors reported by the C compiler. This is followed by a call of the LINT static analyser which detects and reports program errors.

The static analyser shows that the scalar variables c and i have been used but not initialized and that printarray has been called with a different number of arguments than are declared. It also identifies the inconsistent use of the first argument in printarray and the fact that the function value is never used.

Static analysers are useful tools which can take over some of the error checking functions of the program inspection process. However, they should not be considered as a substitute for inspections as there are a significant number of error types which they cannot detect. For example, they can detect uninitialized variables but they cannot detect initializations which are incorrect. They can detect (in a language like C) functions which have the wrong numbers and types of arguments but they cannot detect situations where an incorrect argument of the correct type has been correctly passed to a function.

Redundant code detection

Background

For ease of exposition, we present our approach with reference to the simple PASCAL-like programming language. The *control flow graph* CG = (V, A, en, ex) of a program is a directed graph having a unique entry node en and a unique exit node ex. V is a set of nodes (instructions). An instruction corresponds to an assignment, input, or output statement. In addition, an instruction corresponds to the expression part of an if- or while-statement (called conditional instruction). A is a set of arcs which represent a possible transfer of control from one instruction to another. A *path* in the control graph (called control path) is an ordered sequence $a_1, a_2, \ldots, a_n$ of nodes from V such that (a_i, a_{i+1}) is in A for all a_i, $1 \leqslant i < n$.

A *use* of variable y is an instruction X in which this variable is referenced. A use can be a conditional instruction, an assignment instruction, or an output instruction. A *definition* of variable y is an instruction X which assigns a value to that variable. A definition can be an assignment instruction or an input instruction.

2. The program dependence graph

It is important for a program model to capture as much data and control information as possible. One way of modeling program structure is based on the concept of dependence relations between program instructions. Some related work has been performed in the area of dependence-based program modeling. Most representations in that area treat all dependences as data dependences. For instance, data flow analysis [3] captures only data dependences. In some models, the control dependence is introduced, e.g., in [15] for contructing program slices, in [12] for optimization, and in [6] for static program testing.

Generally, we distinguish between two kinds of dependences between program instructions namely the *data* dependence and the *control* dependence. The data dependence is represented by data flow (definition-use chain), i.e., one instruction assigns a value to an item of data and the other instruction uses the value of the data item [1,4]. The control dependence is defined between conditional instruction and the instructions which can be chosen to execute or not execute by the conditional instruction. For instance, let us consider a part of a PASCAL program:

```
1 if x > 0 then
2   y := 1;
3   z := x ** 2;
```

The conditional instruction 1 has influence on the execution of instruction 2 but has no influence on the execution of instruction 3. Consequently, there is only a control dependence between instructions 1 and 2.

More formally, we define two types of dependences in the program: X is *data dependent* on Y iff there exists a variable v such that: (i) Y is a definition of v, (ii) X is a use of v, and (iii) there exists a control path from Y to X along which v is not redefined.

For instance, in the sample program of Fig. 1, instruction 12 is data dependent on instruction 4.

In the definition of data dependence, the concept of a simple variable is used. A simple variable is a variable having no components. Array variables, record variables, etc. are assumed to have simple variables as their components.

We define control dependence for two PASCAL instructions **if** and **while**:

(a) **if** Y **then** B1 **else** B2 ,

X is *control dependent* on Y iff X is in B1 or B2,

(b) **while** Y **do** B ,

X is control dependent on Y iff X is in B.

```
 1  input(x1);
 2  input(x2);
 3  input(x3);
 4  y1 := 0;
 5  y2 := x2/2;
 6  y3 := 1;
 7  y4 := 0;
 8  while y3 > x3 do
 9  begin
10      if x1 > = y1 + y2 then
11      begin
12        y1 := y1 + y2;
          {y4 := y4 + y3/2;
           correct position of instruction 14}
13      end;
14    y4 := y4 + y3/2;
15    y2 := y2/2;
16    y3 := y3/2;
17  end;
18  output(y4);
```

Fig. 1. A sample program.

In the case of arbitrary control flow, control dependences can be derived by using the concept of the nearest inverse dominator of conditional instructions [2].

The dependence relation can be represented as a directed digraph, where vertices represent instructions, and arcs represent data and control dependences. This graph will be called *program dependence graph* (PDG). More formally,

$$\mathrm{PDG} = \{V, E\},$$

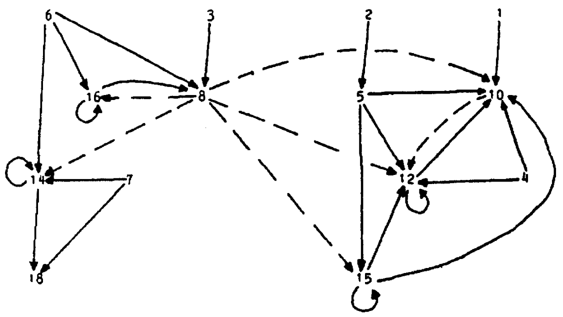


Fig. 2. The program dependence graph of the program of Fig. 1.

where V is the set of all program instructions and

$$E = \{(X, Y) \in V \times V:$$
$$Y \text{ is data or control dependent on } X\}.$$

An example of the program dependence graph is shown in Fig. 2. The solid arcs represent data dependence and dashed arcs represent control dependence.

In the next sections, two static error detection techniques based on the concept of the program dependence graph are described, namely: redundant code analysis, and input–output relation analysis.

3. Redundant code analysis

Intuitively, redundant code is one that may be executed, but its elimination would not change the function of the program computed over its domain. Clearly, this code has no 'influence' on the output of the program. Redundant code may be a symptom of an error in a program. The detection of the redundant code may help in the localization and correction of the error without, perhaps, executing the program. If, however, the program is correct, this code should be deleted. Consequently, a modified program is more reliable than the original one, because the redundant code might be a source of erroneous behavior. Most compilers [14] carry out redundant code detection analysis using control flow only, e.g., unreachable code and null transfer. In [10,11], data flow analysis is used to determine the presence or absence of dead assignments and undefined references.

In this paper, redundant code analysis is based on the observation that each instruction in the program should have influence on its output. Instructions which do not influence output instructions of the program are considered redundant. In terms of the program dependence graph, we say that an instruction X is *redundant* if there is no path in the program dependence graph from X to any of the output instructions. This idea is illustrated in Example A.

Example A. The program of Fig. 1 (cf. [5]) is supposed to divide x1 by x2 within tolerance x3, where x1, x2, and x3 are real numbers satisfying $0 \leqslant x1 < x2$ and $0 < x3$. The final value of y4 is supposed to satisfy $x1/x2 - x3 < y4 \leqslant x1/x2$. There is an error in the program: the instruction in line 14 should be in the scope of the **if**-instruction. One of the symptoms of the error is the redundant code in lines 4, 5, 10, 12, and 15. This fact may not be obvious but it can be shown through tedious argument (for details, see [6]). However, on the basis of the program dependence graph of Fig. 2 it is easy to check that there is no path from any of the instructions 4, 5, 10, 12, and 15 to the output instruction 18. Consequently, these instructions are considered redundant. On the other hand, the correct version of the program does not contain redundant code because there is a control dependence between instructions 10 and 14. Consequently, all instructions influence the output instruction 18. We have to note that this error cannot be detected through data flow analysis alone [3,10].

References

[1] A.V. Aho and J.D. Ullmann, Principles of Compiler Design (Addison-Wesley, Reading, MA, 1977).

[2] D.E. Denning and P.J. Denning, Certification of programs for secure information flow, Comm. ACM 20 (7) (1977) 504–513.

[3] L.D. Fosdick and L.J. Osterweil, Data flow analysis in software reliability, Comput. Surveys 8 (1976) 305–330.

[4] M.S. Hecht, Flow Analysis of Computer Programs (North-Holland, Amsterdam, 1977).

[5] S. Katz and Z. Manna, Logical analysis of programs, Comm. ACM 19 (4) (1976) 188–206.

[7] B. Korel and J. Laski, A tool for data flow oriented program testing, in: SoftFair II—2nd Conf. on Software Development Tools, Techniques, and Alternatives, San Francisco, CA (1985) 34–38.

[8] J. Laski and B. Korel, A data flow oriented program testing strategy, IEEE Trans. Software Engrg. SE-9 (3) (1983) 347–354.

[9] T. Lengauer and R.E. Tarjan, A fast algorithm for finding dominators in a flowgraph, ACM Trans. Programm. Languages & Systems 1 (1979) 121–141.

[10] B. Maher and D.H. Sleeman, Automatic program improvement: Variable usage transformations, ACM Trans. Programm. Languages & Systems 5 (2) (1983) 236–264.

[11] L.J. Osterweil and L.D. Fosdick, DAVE—A validation, error-detection and documentation system for FORTRAN programs, Software-Practice & Experience 6 (1976) 473–486.

[12] K.J. Ottenstein and L.M. Ottenstein, The program dependence graph in a software development environment, ACM SIGPLAN Notices 19 (5) (1984) 177–184.

[13] R.F. Sarraga, Static data flow analysis of PL/I programs with the PROBE system, IEEE Trans. Software Engrg. SE-10 (4) (1984) 451–459.

[14] B.M. Shahdad, A survey of static analysis features of compilers, Proc. Symp. on Application and Assessment of Automated Tools for Software Development, San Francisco, CA (1983) 156–165.

[15] M. Weiser, Program slicing, IEEE Trans. Software Engrg. SE-10 (4) (1984) 352–357.