Universidade do Minho

A survey on Code Analysis and Slicing

Program Analysis and Transformation
2007/2008

Braga, July 2008

Contents

Glossary

A

API Application Programming Interface, p. 26.

ASP Active Server Pages, p. 20.

AST Abstract Syntax Tree, p. 15.

C

CFG Control Flow Graph, p. 8.

CLR Common Language Runtime, p. 28.

D

DDG Dynamic Dependence Graph, p. 58.

DLL Dynamic Link library, p. 23.

F

FSA Finite State Automata, p. 16.

H

HTML HyperText Markup Language, p. 20.

J

JVM Java Virtual Machine, p. 19.

L

LR Left-to-right parse, Rightmost derivation, p. 14.

M

MDG Module Dependence Graph, p. 12.

P

PDG Program Dependence Graph, p. 9.

S

SDG System Dependence Graph, p. 10.

SSA Single Static Assignment, p. 13.

T

TFG Trace Flow Graph, p. 13.

V

VDG Value Dependence Graph, p. 12.

Chapter 1

State-of-the-Art: Code Analysis

The increasing amount of software developed in the last few years have produced a growing demand for programmers and programmer productivity to maintain it working along the years. During maintenance, the most reliable and accurate description of the behavior of a software system is its source code. However, given the complexity of modern software, the manual analysis of source code is costly and ineffective. A more viable solution is to resort to tool support. Such tools provide information to programmers that can be used to coordinate their efforts and improve their overall productivity.
In [Bin07], David Binkley presents a definition of source code analysis:

> *Source code analysis* is the process of extracting information about a program from its source code or artifacts (e.g. from Java byte code or execution traces) generated from the source code using automatic tools. *Source code* is any static, textual, human readable, fully executable description of a computer program that can be compiled automatically into an executable form. To support dynamic analysis the description can include documents needed to execute or compile programs, such as program inputs.

The rest of this chapter will have as basis this definition of source code analysis.
At the earlier stage of compilers (when they were introduced), programmers compile their code and then make minor adjustments (tweak) to the output assembly code to improve its performance. Once adjusted, future updates (that might be better made at high-level source) require one of the three choices:

- re-adjusting the assembly code;
- performing the changes at the lower-level assembly code; or
- changing the high-level source code, recompiling and forget the adjustments.

The final option was adopted after the emergency of the improved compiler technology and faster hardware.
Nowadays, the modern software projects often start with the construction of models (e.g. using the UML). These models can be "compiled" to a lower-level representation: source code. But this code is incomplete and thus requires that the programmers analyze the generated

code and complete it. Until such models are fully executable, the source code is considered "the truth" and "the system".

So, in both cases, code analysis is a relevant task in the life cycle of programs.

There are two kind of code analysis: static and dynamic code analysis. In both of them, the extracted information must be coherent with the language semantics and should be disproved from lexical concerns, focusing on abstract semantic information. This extracted information should help a programmer gain insight of the source code's meaning.

The remainder of this chapter is organized as follow. In section 1.1 are presented some basic concepts around code analysis area. In section 1.2 are presented the stages of a typical code analysis. In section 1.3 some current code analysis techniques are discussed. In section 1.4 the applications of code analysis are reviewed. In section 1.5 some tools for code analysis are presented.

1.1 Basic Concepts

In this section are introduced some basic concepts related not only with code analysis but also with the other areas covered by this pre-thesis. These concepts are relevant to a better understanding of the remainder of this and the following chapters.

Definition 1 *A* Control Flow Graph *(CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. A CFG contains a node for each statement and control predicate in the program; an edge from node i to node j indicates the possible flow of control from the former to the latter. CFGs contains special nodes labeled Start and Stop indicating the beginning and the end of the program, respectively.*

There are several types of **data dependencies**: *flow dependence*; *output dependence*; and *anti-dependence*. In the context of slicing, only flow dependence is relevant.

Definition 2 *A node j is* flow dependent *on node i if there exists a variable x such that:*

- $x \in DEF(i)$;
- $x \in REF(j)$; *and*
- *there exists a path from i to j without intervening definitions of x.*

where DEF(i) denotes the set of variables defined at node i, and REF(i) denotes the set of variables referenced at node i.

In other words, we can say that the definition of a variable x at a node i is a *reaching definition* for node j.

Control dependence is usually defined in terms of *post-dominance*.

Definition 3 *A node i in the CFG is* post-dominated *by a node j if all paths from i to Stop pass through j.*

Definition 4 *A node j is* control dependent *on a node i if and only if:*

- *There exists a path from i to j such for that any $u \neq i$, in that path u is post-dominated by j; and*
- *i is not post-dominated by j.*

Notice that if j is control dependent on i, then i has two outgoing edges (i.e., corresponds to a predicate). Following one of the edges always results in j being executed, while taking the other edge may result in j not being executed. If the edge which always causes the execution of j is labeled with *true* ($false$, respectively), then j is control dependent on the *true* ($false$) branch of i.

Definition 5 *A program path from the entry node Start to the exit Stop is a* feasible path *if there exists some input values which cause the path to be traversed during program execution (assuming program termination).*

Definition 6 *A* state trajectory *of length k of a program P for input I is a finite list of ordered pairs $T =< (p_1, \sigma_1), (p_2, \sigma_2), ..., (p_k, \sigma_k) >$, where $p_i \in P$, $1 \leq i \leq k$, and σ_i is a function mapping the variables in V to the values they assume immediately before the execution of p_i.*

A feasible path that has actually been executed for some input can be mapped onto the values the variables in V (V is the set of variables in a program P) assume before the execution of each statement. Such a mapping is the referred **state trajectory**. An input to the program univocally determines a state trajectory.
Program slices can be computed using the *Program Dependence Graph* [FOW87, HRB88] both at intraprocedural [OO84] and interprocedure level [KFS93b], and also in the presence of goto statements [CF94]. A program dependence graph (PDG) is a program representation containing the same nodes as the CFG and two types of edges: *control dependence edges* and *data dependence edges*.

Definition 7 *A program dependence graph (PDG) is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies.*

The concepts hereby defined of CFG and PDG are illustrated in Figures 1.1 and 1.2 w.r.t. program in Listing 1.1, which asks for a number n and computes the sum and the product of the first n positive numbers.
In Figure 1.1 node 7 is flow dependent on node 4 because:

a) Node 4 defines variable `product`;

b) Node 7 references variable `product`; and

c) There exists a path $4 \rightarrow 5 \rightarrow 6 \rightarrow 7$ without intervening definitions of product.

Notice that, for the same reason, node 7 is also flow dependent on node 2 and 8.
Also in the CFG of Figure 1.1, node 7 is control dependent on node 5 because there exists a path $5 \rightarrow 6 \rightarrow 7$ such that:

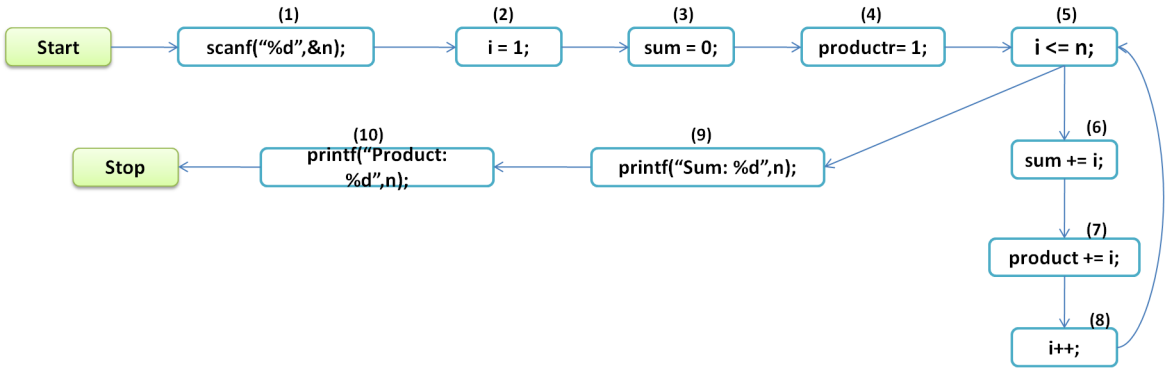

Figure 1.1: CFG corresponding to the program listed in 1.1

a) Node 6 is post-dominated by node 7; and

b) Node 5 is not post-dominated by node 7.

Figure 1.2 shows a PDG constructed according to [HRB88] variant, where solid edges represent control dependencies and dashed edges represent flow dependencies.

Listing 1.1: Program example 1: iterate sum and product

```
1  main() {
2      int n, i, sum, product;
3      scanf("%d",&n);
4      i = 1;
5      sum = 0;
6      product = 1;
7      while (i <= n) {
8          sum += i;
9          product *= i;
10         i++;
11     }
12     printf("Sum: %d\n", sum);
13     printf("Product: %d\n", product);
14 }
```

Definition 8 *A* System Dependence Graph *(SDG) is a collection of procedure-dependence graphs (PDG) - one for each procedure - in which vertices are statements or predicate expressions.*

The term "system" is used to emphasize a program with multiple procedures. Parameter passing by value-result is modeled as follows:

a) *the calling procedure copies its actual parameters to temporary variables before the call;*

b) *the formal parameters of the called procedure are initialized using the corresponding temporary variables;*

c) *before returning, the called procedure copies the final values of the formal parameters to the temporary variables; and*

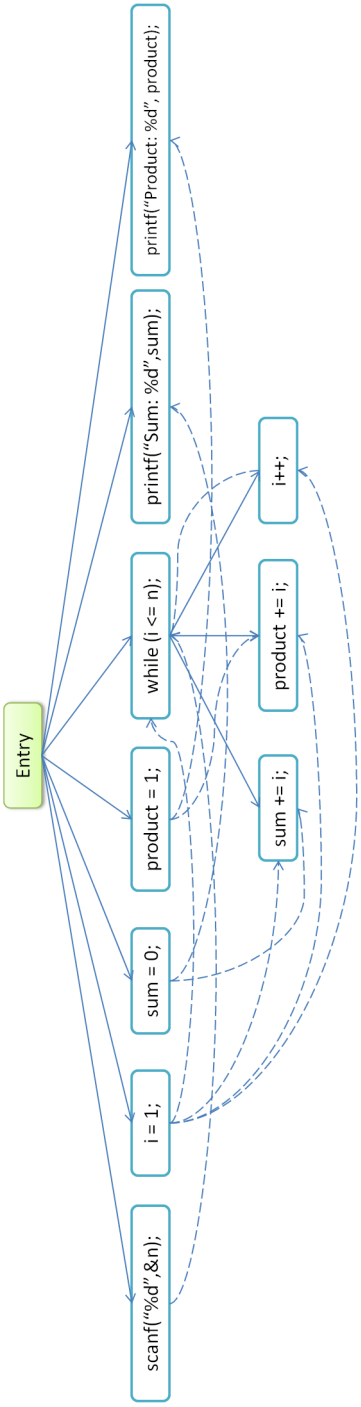

Figure 1.2: PDG corresponding to the program listed in 1.1

d) *after returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables.*

A SDG contains a program dependence graph for the main program, and a procedure dependence graph for each procedure. Procedure dependence graphs are very similar to program dependence graphs except that they include vertices and edges representing call statements, parameter passing, and transitive flow dependencies due to calls. A call statement is represented using a call *vertex; in the called procedure, parameter passing is represented using four kinds of* parameter *vertices: on the calling side, parameter passing is represented by* actual-in *and* actual-out *vertices, which are control dependent on the call vertex and model copying of actual parameters to/from temporary variables; in the called procedure, parameter passing is represented by* formal-in *and* formal-out *vertices, which are control dependent on the procedure's entry vertex and model copying of formal parameters to/from temporary variables. Actual-in and formal-in vertices are included for every global variable that may be used or modified as a result of the call and for every parameter; actual-out and formal-out are included only for global variables and parameters that may be modified as a result of the call.*

Transitive dependence edges, called summary *edges, are added from actual-in vertices to actual-out vertices to represent transitive flow dependencies due to called procedures. According to the Reps et al algorithm [RHSR94], a summary edge is added if a path of control, flow and summary edges exists in the called procedure from the corresponding formal-in vertex to the corresponding formal-out vertex. The addition of a summary edge in procedure Q may complete a path from a formal-in vertex to a formal-out vertex in Q's PDG, which in turn may enable the addition of further summary edges in procedures that call Q.*

Procedure dependence graphs are connected to form a SDG using three new kinds of edges:

- *a* call *edge is added from each call-site vertex to the corresponding procedure-entry vertex;*
- *a* parameter-in *edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; and*
- *a* parameter-out *edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.*

Definition 9 *A* Call Graph *is a directed graph that represents calling relationships between subroutines in a program. Each node represents a procedure and each edge* (f, g) *indicates that procedure* f *calls procedure* g*. Thus, a cycle in the graph indicates recursive procedure calls.*

Definition 10 *A* Value Dependence Graph *(VDG) is a directed graph whose vertices are nodes representing computations and operand values (ports) representing values. Arcs connect nodes to their operand values and ports to the computation (nodes) producing them as results. Each port is either produced by exactly one node or is a free value*[1] *not produced by any node.*

Definition 11 *A* Module Dependence Graph *(MDG) is a graph* $MDG = (M, R)$ *where* M *is the set of named modules of a software system, and* $R \subseteq M \times M$ *is the set of ordered pairs*

[1] The free values of a VDG can be viewed as analogous to the free variables in a lambda term.

$< u, v >$ that represent the source-level dependencies (e.g., procedural invocation, variable access) between modules u and v of the same system.

Definition 12 *A* XTA[2] Graph *is a graph $G = \{V, E, TypeFilters, ReachableTypes\}$:*

- *$V \subseteq M \bigcup F\{\alpha\}$, where M is a set of methods, F is a set of fields, and α is an abstract name representing array elements;*
- *$E \subseteq V \times V$, is the set of directed edges;*
- *$TypeFilters \subseteq E \rightarrow S$, is a map from an edge to a set of types; and*
- *$ReachableTypes \subseteq V \rightarrow T$, is a map from a node to a set of types T.*

The XTA graph combines call graphs and field/array accesses. A call from a method A to a method B is modeled by an edge from node A to node B. The filter set includes parameter types of method B. If B's return type is a reference type, it is added in the filter set of the edge from B to A. Field reads and writes are modeled by edges between method and fields, with the fields' declaring classes in the filter. Each node as a set of reachable types.

Definition 13 *A* Trace Flow Graph *(TFG) is derived from a collection of annotated CFGs. The TFG is a reduced "inlined" representation of the CFGs. In the TFG, all method invocations are replaced by expansions of the methods that they call, and the resulting graph is then reduced by the removal of all nodes that neither bear event annotations nor affect control flow.*

Definition 14 *The* Static Single Assignment *(SSA) is a representation that exposes very explicitly the flow of data within the program. Every time a variable X is assigned a new value, the compiler creates a new version of X and the next time that variable X is used, the compiler looks up the latest version of X and uses that.*

The central idea of the SSA is versioning. This representation is completely internal to the compiler, it is not something that shows up in the generated code nor could be observed by the debugger.
For example, for the program below (see Listing 1.2) the internal representation using the SSA form is shown in Listing 1.3.

Listing 1.2: Program example

```
1 int getValue() {
2     int a = 3;
3     int b = 9;
4     int c = a + b;
5     int d = a + c;
6     return d;
7 }
```

[2]XTA is a mechanism for implementing a dynamic reachability-based interprocedural analysis.

Listing 1.3: Static Single Assignment Form of Listing 1.2

```
1 int getValue() {
2     int a_1 = 3;
3     int b_2 = 9;
4     int c_3 = a_1 + b_2;
5     int d_4 = a_1 + c_3;
6     return d_4;
7 }
```

Notice that every assignment generates a new version number for the variable being modified. And every time a variable is used inside an expression, it always uses the latest version. So, the use of variable a in line 4 is modified to use a_1.

Definition 15 *An* Abstract Syntax Tree *(AST) is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators.*

1.2 Anatomy of code analysis

Under the umbrella of code analysis, there are many techniques used to handle relevant static and dynamic information from a program: slicing, parsing, software visualization, software metrics, and so on. In this section, the three components needed for code analysis are described.

The three components, illustrated in Figure 1.3, are:

- Data extraction;
- Information representation; and
- Knowledge exploration.

1.2.1 Data extraction

The process of retrieving data out of data sources for further data processing or data storage is named data extraction. The import of that data into an intermediate representation is a common strategy to make easier the data analysis/transformation and possibly the addition of metadata prior to export to another stage in the data workflow.

In the context of the code analysis this process is usually done by a syntactic analyzer, or parser. It parses the code into one or more internal representations. A parser is the part of a compiler that goes through a program and cuts it into identifiable chunks before translation, each chunk more understandable than the whole.

Basically, the parser searches for patterns of operators and operands to group the source string into smaller but meaningful parts (which are commonly called *chunks*).

Parsing is the necessary evil of most code analysis. While not theoretically difficult, the complexities of modern programming languages, in particular those that are not LR(1) [AU72, FRJL88] and those incorporating some kind of preprocessing significantly make harder code analysis, as will be seen in section 1.3.1.

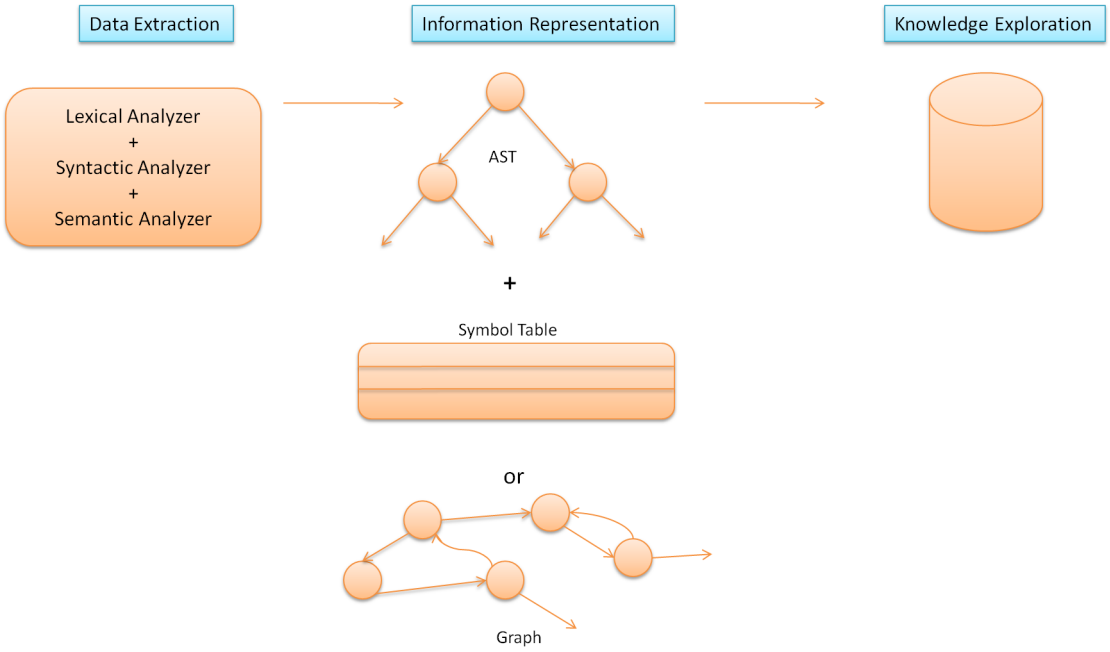

Figure 1.3: Components of code analysis

1.2.2 Information representation

After extracting from the code the relevant information, there is a need to represent it in a more abstract form. This is the second component of code analysis: store the collected data into an internal representation, such that data is kept grouped in meaningful parts and the relations among them are also stored to give sense to the whole. The main goal of this phase is to abstract a particular aspect of the program into a form more suitable for automated analysis. Essentially, an abstraction is a sound, property-preserving, transformation to a smaller domain. Some internal representations are produced directly by the parser (e.g. Abstract Syntax Tree (AST), Control Flow Graph (CFG), etc), while others require the result of prior analysis (e.g., dependence graphs requires prior pointer analysis).

Many internal representations raise from the compilers area. Generally, the most common internal representation is the graph (specially if it degenerates in forms such as trees) — the most widely used are the Control Flow Graph (CFG), the Call Graph, and the Abstract Syntax Tree (AST). The Value Dependence Graph (VDG) is another graph variant that improves (at least for some analysis) the results obtained using SSA form; VDG and SSA were both defined in section 1.1. VDG represents control flow as data flow and thus simplify analysis [WCES94].

Another used graph is the Dependence Graph (see section 1.1), introduced in the context of a work with parallelizing and highly optimizing compilers [FOW87], where vertices represent the statements and predicates of the program. These graphs have since been used in other analysis [HRB88, HR92, Bal02]. A related graph, the Module Dependency Graph (MDG), used by the Bunch tool, represents programs at a coarser level of granularity. Its vertices represents modules of the system and edges the dependencies between them [MMCG99].

Other sorts of graphs, also referred in the literature and defined in section 1.1, include Dy-

namic Call Graphs [QH04, PV06] (a dynamic call graph is intended to record an execution of a program) and XTA graphs [QH04] built in support of dynamic reachibility-based inter-procedural analysis. These techniques are required to analyze languages such as Java that include dynamic class loading.

Finally, the Trace Flow Graph is used to represent concurrent programs [CCO01].

Another known internal representation also referred in section 1.1 is the Single Static Assignment (SSA). SSA form simplifies and improves the precision of a variety of data-flow analyzes.

Also Finite-state Automata (FSA) are used to represent analysis of event-driven systems and the transitions in distributed programs where they provide a formalism for the abstraction of program models [Sch02].

In real applications, it is common to combine different kinds of graphs or AST with Identifiers Table (or similar mapping) in such a way the enriches and structures the information extracted.

All of the variants of graphs or other internal representations presented are actually used according to the type of analysis and the desired results of that analysis.

1.2.3 Knowledge Exploration

After organizing the data extracted into an intermediate representation that makes or transforms it into information, the third component of code analysis is aimed at knowledge inference. This process requires the inter-connection of the pieces of the information stored and their inter-relation with previous knowledge. This can be achieved using quantitative or qualitative methods. Concerning quantitative methods, the resort to program metrics is the most common used approach. Concerning qualitative methods, name analysis, text and data mining, and information retrieval are the most widely used. Visualization techniques are crucial for the effectiveness of that process.

According to Binkley [Bin07], the main strategies used to extract the knowledge from the Intermediate Representation could be classified as follows: *static* versus *dynamic*, *sound* versus *unsound*, *flow sensitive* versus *flow insensitive*, and *context sensitive* versus *context insensitive*.

Static vs dynamic

Static analysis analyze the program to obtain information that is valid for all possible executions. Dynamic analysis instrument the program to collect information as it runs. The results of a dynamic analysis are typically valid for the run in question, but make no guarantees for other runs. For example, a dynamic analysis for the problem of determining the values of global variables could simply record the values as they are assigned. A static analysis might analyze the program to find all statements that potentially affect the global variables, then analyze the statements to extract information about the assigned values.

Dynamic analysis has the advantage that detailed information about a single execution is typically much easier to obtain than comparably detailed information that is valid over all executions.

Another significant advantage of dynamic tools is the precision of the information that they provide, at least for the execution under consideration. Virtually all static analysis extract

properties that are only approximations of the properties that actually hold when the program runs. This imprecision means that a static analysis may provide information that is not accurate enough to be useful. If the static analysis is designed to detect errors (as opposed to simply extracting interesting properties), the approximations may cause the tool to report many false positives. Because dynamic analysis usually record complete information about the current execution, that approach does not suffer from these problems. The trade-off, of course, is that the properties extracted from one execution may not hold in all executions.

Some techniques sit in between. They take into account a collection of initial states that, for example, satisfy a predicate.

Sound vs unsound

A deductive system is sound with respect to a semantics if it only proves valid arguments.

So, a sound analysis makes correctness guarantees.

Sound static analysis produce information that is guaranteed to hold on all program executions; sound dynamic analysis produce information that is guaranteed to hold for the analyzed execution alone. Unsound analysis make no such guarantees. A sound analysis for determining the potential values of global variables might, for example, use pointer analysis to ensure that it correctly models the effect of indirect assignments that take place via pointers to global variables. An unsound analysis might simply scan the program to locate and analyze only assignments that use the global variable directly, by name. Because such an analysis ignores the effect of indirect assignments, it may fail to compute all of the potential values of global variables.

Unsound analysis can exploit information that is unavailable to sound analysis [JR00]. Examples of this kind of information include information present in comments and identifiers. Ratiu and Deissenboeck [RD06] described how to exploit non-structural information such as identifiers in maintaining and extracting the mapping between the source code and real real word concepts.

Maybe it could be "unsound" why an engineering will be interested in unsound analysis. However, in many cases, the information from an unsound analysis is correct, and even when incorrect, may provide a useful starting point for further investigation. Unsound analysis are therefore often quite useful for those faced with the task of understanding and maintaining legacy code.

The most important advantages of unsound analysis, however, are their ease of implementation and efficiency. Reconsider the two examples cited above for extracting the potential values of global variables. Pointer analysis is a complicated interprocedural analysis that requires a sophisticated program analysis infrastructure and a potentially time-consuming analysis of the entire program; locating direct assignments, on the other hand, requires nothing more than a simple linear scan of the program. An unsound analysis may thus be able to analyze programs that are simply beyond the reach of the corresponding sound analysis, and may be implemented with a small fraction of the implementation time and effort required for the sound analysis.

For all these reasons, unsound analysis will continue to be important.

A slightly different concept of sound analysis is the *safe analysis*.

Safe static analysis means that the answer is precise on "one side". For example, a reaching-

definitions computation can determine that certain assignments definitely do not reach a given use, but the remaining assignments may or not reach the use.

Sagiv et al. [SRW02] present a static analysis technique based on a three-valued logic, capturing indecision as a third value. Thus again using reaching-definition as an example, a definition could be labeled "reaches", "does not reach", or "might reach".

Flow sensitive vs Flow insensitive

Flow-sensitive analysis takes the execution order of the program's statements into account. It normally uses some form of iterative dataflow analysis to produce a potentially different analysis result for each program point. Flow-insensitive analysis do not take the execution order of the program's statements into account, and is therefore incapable of extracting any property that depends on this order. It often use some form of type-based or constraint based analysis to produce a single analysis result that is valid for the entire program.

For example, given the sequence `p = &a; q = p; p = &b;`, a flow-sensitive points-to analysis can determine that q does not point to b.

In contrast, a *flow-insensitive* analysis treats the statements of a program as an unordered collection and must produce conservative results that are safe for any order. In the above example, a flow-insensitive points-to analysis must include that q might point to a or b. This reduction in precision comes with a reduction in computational complexity.

Context sensitive vs Context insensitive

Many programming languages provide constructs such as procedures that can be used in different contexts. Roughly speaking, a *context-insensitive* analysis produces a single result that is used directly in all contexts.

A *context-sensitive* analysis produces a different result for each different analysis context. The two primary approaches are to reanalyze the construct for each new analysis context, or to analyze the construct once (typically in the absence of any information about the contexts in which it will be used) to obtain a single parameterized analysis result that can be specialized for each analysis context.

Context sensitivity is essential for analyzing modern programs in which abstractions (such as abstract datatypes and procedures) are pervasive.

1.3 Current code analysis challenges

In this section we present the challenges that are being posed to the code analysis. Many of this challenges are not related with only one of the components referred in the previous section; instead of it, each issue affects more than one component increasing the level of challenge. The relationship between those challenges and the code analysis components are depicted in Figure 1.4 in a Venn Diagram form, where: orange color refers to the first component (data extraction); green color refers to the second component (information representation); and the blue color refers to the third component (knowledge exploration).

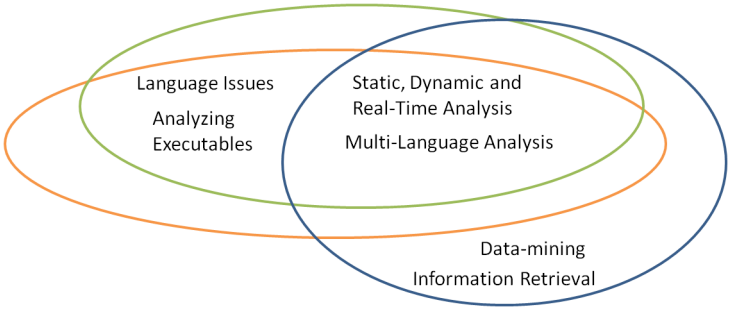

Figure 1.4: Relationship among code analysis

1.3.1 Language Issues

In the last few years, many enhancements have been done to programming languages. The introduction of concepts such as dynamic class loading and reflection in languages such as Java and C# contributes to the language evolution.

Language reflection provides a very versatile way of dynamically linking program components. It allows to create and manipulate objects of any classes without the need to hardcode the target classes ahead of time. These features make reflection especially useful for creating libraries that work with objects in very general ways. For example, reflection is often used in frameworks that persist objects to databases, XML, or other external formats.

Reflection has a couple of drawbacks. One is the performance issue. Reflection is much slower than direct code when used for field and method access. A more serious drawback for many applications is that using reflection can obscure what's actually going on inside the code. Programmers expect to see the logic of a program in the source code, and techniques such as reflection that bypass the source code can create maintenance problems. Reflection code is also more complex than the corresponding direct code.

Dynamic class loading is an important feature of the Java Virtual Machine (JVM). It provides to the Java platform the ability to install software components at run-time. It has a number of unique characteristics. First of all, lazy loading means that classes are loaded on demand and at the last moment possible. Second, dynamic class loading maintains the type safety of the JVM by adding link-time checks, which replace certain run-time checks and are performed only once. Finally, class loaders can be used to provide separate name spaces for various software components. For example, a browser can load applets from different web pages using separate class loaders, thus maintaining a degree of isolation between those applet classes.

This concepts and also the presence of casting, pointer arithmetic and the like make the task of parsing a difficult task.

Modern languages increasingly require tools for high precision source code analysis to handle only partially known behavior (such as generics in Java, plug-in components, reflection, user-defined types, and dynamic class loading).

These features increase flexibility at run-time and impose a more powerful dynamic analysis, but compromise static analysis.

1.3.2 Multi-Language Analysis

Many software systems are heterogeneous today, i.e., they are composed by components of different programming and specification languages. Analysis in current software development tools, e.g., Integrated Development Environments (IDEs), cannot process these mixed-language systems as a whole since they are too closely related to a particular programming languages and do not process mixed-language systems across language boundaries.

So, multi-language analysis grows more important as systems are built of many parts composed of many languages. Even a simple Java program could consist of Java-source and -bytecode components. A larger system, e.g., a WEB application, join SQL, HTML, and Java codes on the server site and additional languages on the client site. For example, the Visual Studio .Net environment merge languages such as ASP, HTML, C#, J#, and Visual Basic.

Below, is a fragment of a small WEB application, that illustrates such mix of languages.

```
1  <%@ Page Language="C#"%>
2
3  <script runat="server">
4      protected void Button1_Click(object sender, EventArgs e) {
5          Response.Redirect("Home.aspx");
6      }
7  </script>
8
9  <html>
10     <head>
11         <script type="text/javascript">
12             function ShowModalPopup() {
13                 var modal = $find('ModalPopupExtender');
14                 modal.show();
15             }
16         </script>
17     </head>
18     <body>
19     <form id="form1" runat="server">
20     <div>
21         <asp:Button ID="Button1" runat="server"
22             OnClick="Button1_Click" Text="Button" />
23     </div>
24     </form>
25 </body>
26 </html>
```

This example contains an ASP.Net web page file. The ASP web page that is basically an HTML file with some special ASP.Net elements and program code. When this page is requested on an ASP application server, the code is executed first, which results in a translated HTML code is sent to the client. The page contains *C#* code in a script region. This code defines the associated event to the button defined in the ASP code. The page also contains a special HTML element <asp:Button>, which represents a button. This element has an attribute *ID* with the value *Button*1. The ASP application server uses this *ID* to allow program code to refer to the <asp:Button> element and to modify it before it is sent to clients.

To support these mixed-language systems with automated analysis, information from all different sources ought to be retrieved and commonly processed. Only a system with a global

view allows for a global correct analysis.

Today's IDEs fail in cross-language analysis. At best, they can only handle several programming languages individually.

In this context of cross-language analysis, Strein et al. [SKL06] the reason for this gap is the lack of a common meta-model capturing program information for analysis that is common for a set of programming languages abstracting from details of each individual language and that is related to the source code level of abstraction in order to allow for code analysis. The authors propose an architecture for analysis composed by three major classes: information extracting front-ends, a component meta-model (model data-structure), and analysis components.

According with the three steps described in the previous section, these three components matches with the three components referred for the code analysis, where the common meta-model corresponds to the intermediate representation.

The common meta-model captures program information in a language independent way.

Different language specific front-ends extract information from program written in the respective languages. They use language specific analysis and capture information about the program in a language specific meta-model first. Information that is relevant for global analysis is also stored in the common model.

The front-ends retrieve the information represented in the common model to implement low-level analysis (e.g. to look-up declarations). Different high-level analysis access the common model. The common model represents information gained from analysis of a complete mixed-language program. Thus, concrete analysis based on this information are language agnostic and can handle cross-language relations.

In short, a front-end is responsible for parsing and analyzing specific languages, whereas the common model stores the relevant analysis information abstracting from language specific details. The common meta-model is accessed by language independent analysis. This meta-model does not need to be a union of all language concepts of all languages to support. Instead, it is sufficient to model only those language concepts, that are relevant to higher level analysis or to other languages.

Formally, each *front-end* supports a specific file type F that incorporates a set of supported languages: $F = \{L_1, L_2, ..., L_n\}$. A file type F-specific front-end $F^{\mathcal{F}}$ is defined by a triple:

$$F^{\mathcal{F}} = (\phi^F, \{\alpha^{L_1}, \alpha^{L_2}, ..., alpha^{L_n}\}, \{\sigma^{L_1}, \sigma^{L_2}, ..., \sigma^{L_n}\})$$

The front-end provides the *parsing function* ϕ^F that sorts file parts according to their languages into blocks and constructs syntax trees representing the different file blocks.

For each language L of such a block, the front-end defines the *syntax mapping* α^L, that maps language specific syntax trees AST^L to common meta-model trees AST.

For each language L the *semantic analysis function* σ^L constructs common semantic relations between nodes that are defined by the *syntax mapping*. σ^L is based on the common meta-model $\mathcal{M}$ as well as specific syntactic meta-model to handle language specificities. Through the common meta-model $\mathcal{M}$ it can indirectly access information created by front-ends for other languages. This way, it can be constructed cross-language relations for arbitrary language combinations.

The semantical relations include also dynamic relations that could actually only be computed at runtime, e.g., dynamic types in weakly or dynamic typed languages or dynamic call targets

in object-oriented languages. However, the computation of (non-trivial) dynamic properties using static-analysis is generally an undecidable problem.

At least the parsing, the syntax mappings, the semantic analysis functions need to be implemented for each new filetype. Also, might be necessary to extend the constructs in order to capture properties of a new language.

So, the key for a multi-language analysis is a common meta-model to capture the concepts of each programming language. However, as referred in subsection 1.3.1, parsing languages with mismatched concepts and with different principles is not an easy task, specially when dealing with dynamic languages.

1.3.3 Static, Dynamic and Real-Time analysis

Static analysis is usually faster than dynamic analysis but less precise. Therefore it is often desirable to retain information from static analysis for run-time verification, or to compare the results of both techniques. It would be desirable to share the same generic algorithm by static and dynamic analysis.

Martin et al. describe in [MLL05] an error detection tool that checks if a program conforms to certain design rules. This system automatically generates from a query a pair of complementary checkers: a static checker that finds all potential matches in an application and a dynamic checker that traps all matches precisely as they occur.

Slightly more sophisticated combinations often use static analysis to limit the need for instrumentation in the dynamic analysis. Path testing tools use this approach as does Martin et al.'s error detection tool, where "static results are also useful in reducing the number of instrumentation points for dynamic analysis. They report that the combination proves able to address a wide range of debugging and program queries.

Gupta et al. [GSH97] present an algorithm that integrates dynamic information from a program's execution into a static analysis. The resulting technique is more precise than static analysis and less costly than the dynamic analysis.

Heuzeroth et al. [HHHL03] consider the problem of reverse engineering design patterns using a more integrated combination of static and dynamic analysis. In this case, static is used first to extract structures regarding potential patterns and then dynamic analysis verifies that pattern candidates have the correct behavior. Here static analysis does more than improve the efficiency of the dynamic approach. The two truly compliment each other.

Closer still to true integration is a combination that, in essence, iterates the two to search for test data input. This technique applies a progression of ever more complex static analysis with search. This synergistic arrangement allow low-cost static analysis to remove "obvious" uninteresting paths. It then applies relatively naive, but inexpensive dynamic search. If more test is needed more sophisticated static and dynamic techniques are applied. All these techniques, however, fall short of a truly integrated combination of static and dynamic techniques. Future combinations should better integrate the two.

Another kind of analysis that should be considered is the real-time analysis. This research problem has two distinct facets: compile-time and run-time. Self-healing code[3] and instrumented code are run-time examples. Here analysis is being done real time while the program

[3]While no consensus-based definition of the term "self-healing" exists, intuitively, these systems automatically repair internal faults.

is executing. The archetypical example of this idea is *just-in-time* compilation.
Looking forward, more such processing can be done in real-time. For instance, code coverage and memory-leak analysis might be performed, at least partially, at compile time instead of at run-time. This has the advantage of providing information about a piece of code that is current focus of the programmer.

Other future challenges in code analysis will emerge, such as the combination of source code analysis with natural language analysis; real-time verification; and improved support for user interaction (rather than being asked to make a collection of similar low-level choices, tools will ask about higher level-patterns that can be used to avoid future questioning). Code analysis tools will also need to use information from edit, compile, link, and run-time and continue to include a combination of multiple views of a software system such as structure, behavior, and run-time snapshots, that is what is being proposed in this thesis.

1.3.4 Analyzing executables

In the past years a considerable amount of research activity to develop static-analysis tools to find bugs and vulnerabilities. However, most of the effort has been on static-analysis of source code, and the issue of analyzing executables was ignored. In the security context, this is particular unfortunate because source code analysis can fail to detect certain vulnerabilities due to the phenomenon: "What You See Is Not What You eXecute" (WYSINWYX). That is, there can be a mismatch between what a programmer intends and what is actually executed on the processor.
Thomas Reps et al. [RBL06] presents a number of reasons why analysis based on source code do not provide the right level of detail for checking certain kind of properties:

1. Source level tools are only applicable when source is available, which limits their usefulness in security applications (e.g. to analyzing code from open-source projects);

2. Analysis based on source code typically make assumptions. This often means that an analysis does not account for behaviors that are allowed by the compiler;

3. Programs make extensive use of libraries, including Dynamic Linked Libraries (DLL), which may not be available in source code form. Typically, source-level analysis are performing using code stubs that model the effects of library calls.

4. Programs are sometimes modified subsequent to compilation, e.g. to perform optimizations or insert instrumentation code [Wal91]. They also be modified to insert malicious code. Such modifications are not visible to tools that analyze source code.

5. The source code may have been written in more than one language. This complicates, as referred in previous subsection, the life of designers of tools that analyze source code because multiple languages must be supported, each with its own peculiarities.

6. Even if the source code is primarily written in one high-level language, it may contain inlined assembly code in selected places.

Thus, even if source code is available, a substantial amount of information is hidden from analysis that start from source code, which can cause bugs, security vulnerabilities, and

malicious behavior to be invisible to such tools. Moreover, a source-level analysis tool that exert to have greater fidelity to the program that is actually executed would have to duplicate all of the choices made by the compiler and optimizer, such an approach is destined to fail.
The main goal of the work done and presented in [RBL06] was to recover, from executables, Intermediate Representations (g̃lossir) that are similar to those that would be available had one started from source code, but expose the platform-specific details discussed above. Specifically, the authors are interested in recovering IRs that represent the following information:

- Control Flow Graphs (CFGs) with indirect jumps resolved;
- A Call Graph with indirect calls resolved;
- Information about the program's variables;
- Sets of used, killed, and possibly.killed variables for each CFG node;
- Data dependencies (including dependencies between instructions that involve memory accesses);
- Type information (e.g. base types, pointer types, and structs).

In IR recovery, there are numerous obstacles that must be overcome. In particular, in many situations debugging information is not available. So, the authors have designed IR-recovery techniques that do not rely on debugging information being present and are language-independent.
One of the main challenges in static analysis of low-level code is to recover information about memory access operations (e.g. the set of addresses accessed by each operation). The reasons for this difficulty are:

- While some memory operations use explicit memory addresses in the instruction (easy), others use indirect accessing via address expressions (difficult);
- Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed;
- There is no notion of type at the hardware level: address values are not intrinsically different from integer values;
- Memory accesses do not have to be aligned, so word-size address values could potentially be cobbled together from misaligned reads and writes.

As a proof of concepts exposed in [RBL06], the authors implement a set of tools: CodeSurfer/x86, WPDS++ and Path Inspector.
CodeSurfer/x86 recovers IRs from an executable that are similar to the IRs that source code analysis tools create.
WPDS++ [KRML04] is a library for answering generalized reachability queries on *weighted pushdown systems* (WPDSs) [RSJM05]. This library provides a mechanism for defining and solving model-checking and data-flow analysis problems.
The Path Inspector is a software model checker built on top of CodeSurfer and WPDS++.

To recover the IR the authors assume that the executable that is being analyzed follows a "standard compilation model". By this, they means that the executable has procedures, activation records, a global data region, and a heap; might use virtual functions and DLLs; maintains a runtime stack; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides at a fixed offset in the activation records for f; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at a fixed offsets in the activation records for f; the program's instruction occupy a fixed area of memory and are not self-modifying.

With this assumptions, this set of tools gave a major contribution in the variable and type discovery area, especially for aggregates (i.e., structures and arrays). The variable and type discovery phase of CodeSurfer/x86 recovers such information for variables that are allocated globally, locally (i.e. on the run-time stack), and dynamically (i.e. from the heap). An iterative strategy is used; with each round of the analysis, the notion of the program's variables and types is refined. The memory model that they use is an abstraction of the concrete (runtime) address space, and has two parts:

- **Memory-regions** Although in the concrete semantics the activation records for procedures, the heap, and the memory for global data are all part of one address space, for the purpose of analysis, they separate the address space into a set of disjoint areas, which are referred as memory-regions. Each memory-region represents a group of locations that have similar runtime properties.
- **A-loc** The second part of the memory model uses a set of proxies for variables, which are inferred for each memory-region. Such objects are called *a-locs*, which stands for "abstract locations". In addition to the a-locs identified for each memory-region, the registers represent an additional class of a-locs.

Many efforts have being made to improve the recovery of IRs through the analysis of executables. However, there is still a need to study this area to cover other aspects like dynamic languages, object-oriented programming languages and so on.

1.3.5 Information Retrieval

In the last years, Information Retrieval (IR) was blossomed with the grow of the Internet and the huge amount of information available in electronic form.

Some applications of IR to code analysis include automatic link extraction [ZB04], concept location [MSRM04], software and website modularization [GMMS07], reverse engineering [Mar03], software reuse impact analysis [SR03, FN87], quality assessment [LFB06], and software measurement [Hoe05, HSS01].

These techniques could be used to estimate a language model for each "document" (e.g. a source file, a class, an error log, etc) and then use a classifier (e.g. a classifier based on the Bayesian's theorem which relates the conditional and marginal probabilities of two random events) to score each. Much of this work has a strong focus on program identifiers [LMFB06]. Unlike other approaches that consider non-source documents (e.g. the requirements), this approach focuses exclusively on the code. It divides each source code module into two documents: one includes the comments and the other the executable source code.

To date, the application of IR has concentrated on processing the text from source and non-source software artifact (which can be just as important as source) using only a few developed

IR techniques. Given the growing importance of non-source documents, source code analysis should in time, develop new IR-based algorithms specifically designed for dealing with source code.

1.3.6 Data Mining

Recently the mining of software-related data repositories has started. Techniques such as the analysis of large amounts of data requires significant computing resources and the application of techniques such as pattern recognition [PM04], neural networks [LSL96], and decision trees [GFR06], which have advanced dramatically in recent years.

Most existing techniques has been conducted by software engineering researchers, who often reuse simple data mining techniques such as association mining and clustering. A wider selection of data mining techniques should see more general application that removes the requirement that existing systems fit the features provided by existing mining tools. For example, API usage patterns often involve more than two API method calls or involve orders among API method calls, leaving mining for frequent item sets insufficient. Finally, the mining of API usage patterns in development environments as well as many other tasks pose requirements that cannot be satisfied by reusing existing simple miners in a black-box way.

Data mining is also being applied to software comprehension. In [KT04], the authors propose a model and associated method to extract data from C++ source code which is subsequently to be mined, and evaluates a proposed framework for clustering such data to obtain useful knowledge.

Thus, there is demand for the adaptation or development of more advance data mining methods.

1.4 Applications of code analysis

Along the years, the source-code analysis techniques have being used for many engineering tasks, facing the challenges discussed in the previous sections and many others. This sections lists the applications of code analysis but only few of them will be discussed in detail.

The applications of code analysis are: architecture recovery [Sar03]; clone detection [MM01, LLWY03]; comprehension [Rug95]; debugging [FGKS91]; fault location [11005]; middleware [ICG07]; model checking in formal analysis [DHR+07]; model-driven development [FR07]; performance analysis [WFP07]; program evolution [BR00]; quality assessment [RBF96]; reverse engineering [CP07]; software maintenance [10405]; symbolic execution [KS06]; testing [Ber07, Har00]; tools and environments [Zel07]; verification [BCC+03]; and web application development [Jaz07, FdCHV08].

1.4.1 Debugging

Along the years, debugging and debuggers were a research topic that was decreasing in strength and increasing in quantity, maybe due to the ever increasing complexity of the problem imposed by more complex compiler back-ends and new language features, such the ones previous referred: reflection and dynamic class loading, among others. Some recent debugging innovations that counter this trend includes algorithmic debugging, delta debugging and statistical debugging.

Algorithmic debugging uses programmer responses to a series of questions generated automatically by the debugger. There are two goals for future algorithmic debuggers: first, reducing the number of questions asked in order to find the bug, and second, reducing the complexity of these questions [Sil06].

Delta debugging systematically narrows the difference between two executions: one that passes a test and one that fails [Zel01]. This is done by combining states from these two executions to automatically isolate failure causes. At present the combination is statically defined in terms of the input but a more sophisticated combination might use dependence information to narrow down the set of potential variables and statements to be considered.

The SOBER tool uses statistical methods to automatically localizer software faults any prior knowledge of the program semantics [12406]. Unlike existing statistical approaches that select predicates correlated with program failures, SOBER models the predicate evaluation in both correct and incorrect executions and regards a predicate as fault-relevant if its evaluation pattern in incorrect executions significantly diverges from that in correct ones. Featuring a rationale similar to that of hypothesis testing, SOBER quantifies the fault relevance of each predicate in a principled way.

1.4.2 Reverse engineering

Reverse engineering is an attempt to analyze source code to determine the know-how which has been used to create [CP07]. Pattern matching approaches to reverse engineering aim to incorporate domain knowledge and system documentation in the software architecture extraction process. Most existing approaches focus on structural relationships (such as the generalization and association relationships) to find design patterns. However, behavioral recovery, a more challenging task, should be possible using data mining approaches such as sequential pattern discovery. This is useful as some patterns are structurally identical but differ in behavior. Dynamic analysis can be useful in distinguishing such patterns.

1.4.3 Comprehension

The increasing size and complexity of software systems introduces new challenges in comprehending the overall structure of programs.

In this context, program comprehension is necessary to get a deeper understanding of a software application. This is necessary if the software applications need to be changed or extended and its original documentation is missing, incomplete, or inconsistent with the implementation of the software application. Source code analysis as performed by Rigi [MTO+92, TWSM94] or Software Bookshelf [FHK+02] is one approach for program comprehension. These approaches generate a source model that enables the generation of high level sequence and collaboration diagrams. Since the collaboration between different modules also depends on runtime data, dynamic tools such as Software Reconnaissance [84801, WC96], BEE++ [BGL93] or Form [SMS01] have been developed. These approaches identify the code that implements a certain feature by generating different execution traces.

But, for a comprehensive understanding of any software system, several complementary views need to be constructed, capturing information about different aspects of the system in question. The 4+1 Views model, introduced in [Kru95], for example, identifies four different architectural views: *logical* view of the system data, the *process* view of the system's thread

of control, the *physical* view describing the mapping of the software elements onto hardware, and the *development* view describing the organization of the software models during development. *Scenarios* of how the system is used in different types of situations are used to integrate, illustrate and validate the above views.

Chen and Rajlich [CS00] propose a semi-automatic method for feature[4] localization, in which an analyst browse the statically derived System Dependency Graph (SDG). The SDG describes detailed dependencies among subprograms, types, and variables at the level of expressions and statements. Even though navigation on the SDG is computer-aided, the analyst takes on all the search for a feature's implementation.

Understanding a system's implementation without prior knowledge is a hard task for reengineers in general. So, along the years many code analysis models have been proposed to aid program comprehension. However, it would be desirable to have multiple model representing alternative views. For enabling slicing and abstractions mechanisms cross the models, the semantic relations between them should be well defined. It would be useful to reflect modifications in one view directly in the other views. Moreover, for program comprehension, the environment should allow the user to easily navigate between static and dynamic views as well as between low and high level views (for instance, the user might want to select a component in one view and explore its role in the other views).

1.5 Tools for code analysis

Code analysis tools can help to acquire a complete understanding of the structure, behavior and functionality of the system being modified, or they can assist in the assessment of the impact of a change. Code analysis tools are also useful in post-maintenance testing (for example to generate cross-reference information, and to perform data flow analysis) and to produce specification and design level documents that record for future use the knowledge gained during a maintenance operation. Under the umbrella of reverse engineering, many tools are available that support the extraction of system abstractions and design information out of existing software systems.

In this section it will be described tools that are, in some way, related with the purpose of this thesis: interconnect multi-level code.

1.5.1 FxCop

FxCop [Mic08b] is an application that analyzes modules coded in assembly (targets modules included in the .NET Framework Common Language Runtime (CLR)) and reports information about the modules, such as possible design, file system localization, performance, and security improvements. Many of the issues concern violations of the programming and design rules set forth in the Design Guidelines for Class Library Developers [Mic08a], which are the Microsoft guidelines for writing robust and easily maintainable code by using the .NET Framework.

Figure 1.5 shows an working example, where: the left window displays the tree structure of the parsed assembly; the right window displays de errors and warnings resulting of code

[4] A feature f is a realized functional requirement.

analysis; and the bottom window displays either the pretty-print of the parsed code or the explanation for the errors/warnings signaled.

However, the tool has a few drawbacks: it only parses the assembly code and display it in a pretty form; it analyzes the assembly code and does not infer any kind of knowledge, because its analysis is based in a set of pre-defined and fixed rules; it does not have any kind of abstraction/visualization of the intermediate representation, obtained from the parsing, to aid program comprehension; it only analyzes assembly code file one at a time, not relating them, restricting the analysis and comprehension of an whole system to its components (usually a system is composed by many source code and assembly files — as happens with Web applications).

1.5.2 Lint

Lint [Joh78, Dar86] is a tool to examine C programs that compiled without errors, aiding to find bugs that had escaped detection.

Lint's capabilities include:

- complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit `extern` statements but are never referenced.
- attempts to detect variables that are used before being set. Lint detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use" since the actual use may occur at any later time, in a data dependent fashion.
- attempts to detect unreachable portions of the programs. It will complain about unlabeled statements immediately following goto, break, continue, or return statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases `while(1)` and `for(;;)` as infinite loops. Lint also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.
- enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

Obviously a drawback of the Lint tool is that is limited to the C language and uses a static approach,; does not covers modern languages with new improvements, such as the notion of object, class, reflection, or dynamic class loading.

1.5.3 CodeSonar and CodeSurfer

Both CodeSonar [Gra08a] and CodeSurfer [Gra08b] are tools from GramaTech.

CodeSonar is a source code analysis tool that performs a whole-program, interprocedural analysis on C/C++ code and identifies complex programming bugs that can result in system

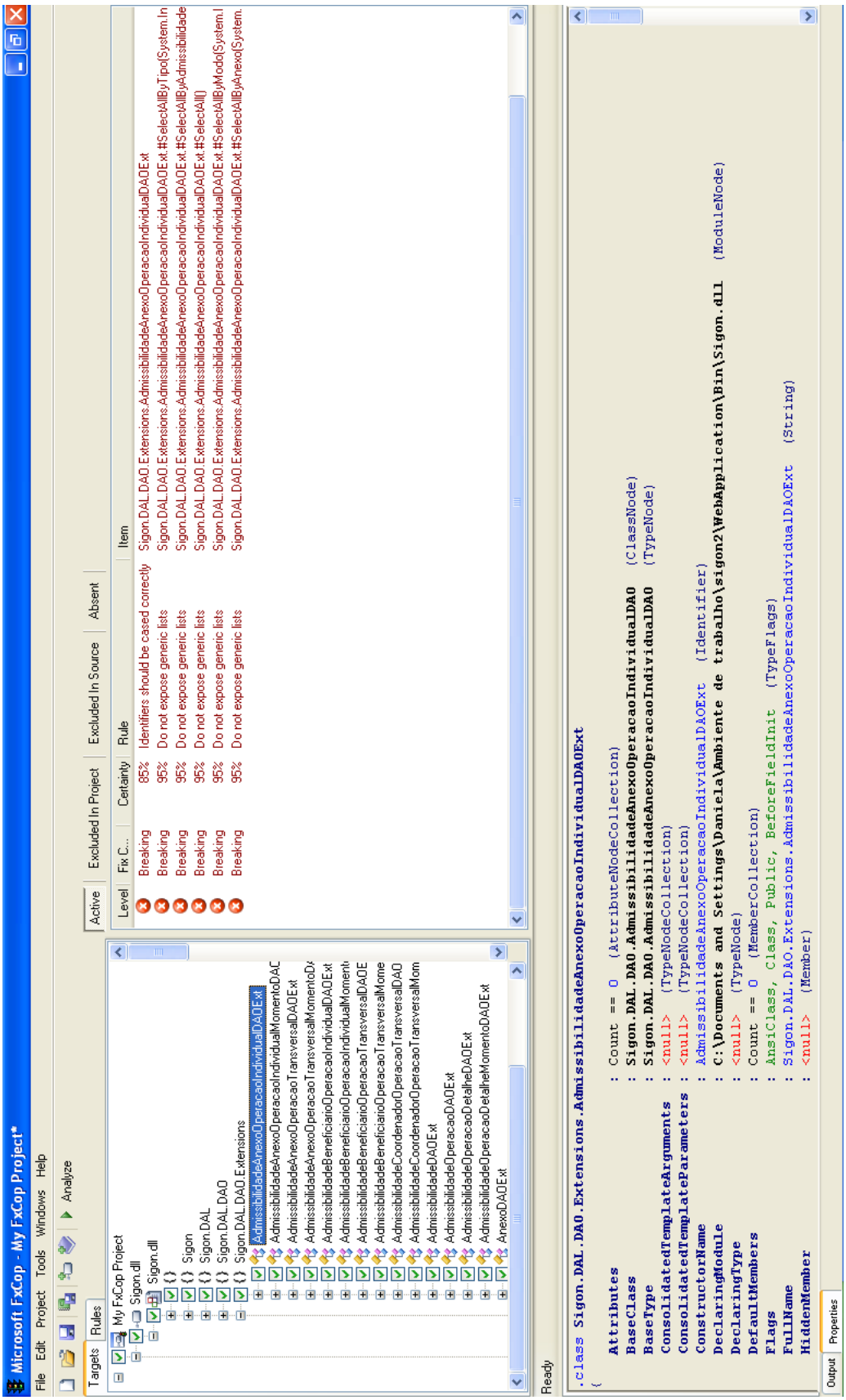

Figure 1.5: FxCop tool

crashes, memory corruption, and other serious problems. CodeSonar pinpoints problems at compile time that can take weeks to identify with traditional testing. The main goals of CodeSonar are:

- to detect and eliminate bugs early in the development cycle, when problems are easier and less expensive to fix;
- to avoid having to debug defects that can be pinpointed quickly and simply with automated analysis; and
- to catch problems that test suites miss.

CodeSurfer, related to CodeSonar, is a program-understanding tool that makes manual review of code easier and faster. CodeSurfer does a precise analysis. Program constructs — including preprocessor directives, macros, and C++ templates — are analyzed correctly. CodeSurfer calculates a variety of representations that can be explored through the graphical user interface or accessed through the optional programming API. Some of its feature include:

- Whole-Program Analysis: see effects between files;
- Pointer Analysis: see which pointers point to which variables and procedures;
- Call Graphs: see a complete call graph, including functions called indirectly via pointers;
- Impact Analysis: see what statements depend on a selected statement;
- Dataflow Analysis: pinpoint where a variable was assigned its value.
- Control Dependence Analysis: see the code that influences a statement's execution.

Again, the main drawback of this tools is that are language dependent (is this case of C/C++).

Chapter 2

State-of-the-Art: Slicing

Since Weiser first proposed the notion of slicing in 1979 in his PhD thesis [Wei79], hundreds of papers have been proposed in this area. Tens of variants have been studied, as well was algorithms to compute them. Different notions of slicing have different properties and different applications. These notions vary from Weiser's syntax-preserving static slicing to amorphous slicing which is not syntax-preserving; algorithms can be based on dataflow equations, information flow relations or dependence graphs.

Slicing was first developed to facilitate program debugging [M.93, ADS93, WL86], but it is then found helpful in many aspects of the software development life cycle, including software testing [Bin98, HD95], software metrics [OT93, Lak93], software maintenance [CLM96, GL91a], program comprehension [LFM96, HHF+01], component re-use [BE93, CLM95], program integration [BHR95, HPR89b] and so on.

In this chapter, slicing techniques are presented including static slicing, dynamic slicing and the latest slicing techniques. We also discuss the contribution of each work and compare the major difference between them.

The remainder of this chapter is organized as follow. In section 2.1 is presented the concept of program slicing and its variants. In section 2.1.12 are presented the basic methods used to pose program slicing in practice. In section 2.1.11 are discussed the relationship among the different slicing techniques discussed in the section 2.1. In section 2.2 is focused in the basic slicing approaches. In section 2.3 is reviewed the non-static slicing approaches. In section 2.4 is reviewed the applications of program slicing. In section 2.5 is presented some tools using the program slicing approach.

2.1 The Concept of Program Slicing

In this section it is presented the original static slice definition and also its most popular variants. At the end of each subsection, the respective concept will be clarified through. The examples are based on the program introduced hereafter in subsection 2.1.1.

2.1.1 Program example

Listing 2.1 below corresponds to a program, taken from [CCL98], that will be used as the running example for all the next subsection aiming at illustrating each concept introduced.

That program takes the integers n, *test* and a sequence of n integers a as input and compute the integers *possum*, *posprod*, *negsum* and *negprod*. The integers *possum* and *negsum* accumulate the sum of the positive numbers and of the absolute value of the negative numbers in the sequence, respectively. The integers *posprod* and *negprod* accumulate the products of the positive numbers and the absolute value of the negative numbers in the sequence, respectively. Whenever an input a is zero, the greatest sum and the greatest product are reset if the value of *test* is non zero. The program prints the greatest sum and the greatest product computed.

Listing 2.1: Program example 2

```
1  main() {
2      int a, test, n, i, posprod, negprod, possum, negsum, sum, prod;
3      scanf("%d",&test);  scanf("%d",&n);
4      i = posprod = negprod = 1;
5      possum = negsum = 0;
6      while (i <= n) {
7          scanf("%d",&a);
8          if (a > 0) {
9              possum += a;
10             posprod *= a;
11         }
12         else if (a < 0) {
13             negsum -= a;
14             negprod *= (-a);
15         }
16         else if (test) {
17             if (possum >= negsum) {
18                 possum = 0;
19             }
20             else { negsum = 0; }
21             if (posprod >= negprod) {
22                 posprod = 1;
23             }
24             else {
25                 negprod = 1;
26             }
27         }
28         i++;
29     }
30     if (possum >= negsum) {
31         sum = possum;
32     }
33     else { sum = negsum; }
34     if ( posprod >= negprod) {
35         prod = posprod;
36     }
37     else { prod = negprod; }
38     printf("Sum: %d\n", sum);
39     printf("Product: %d\n", prod);
40 }
```

2.1.2 Static slicing

Program slicing, in its original version, is a decomposition technique that extracts from a program the statements relevant to a particular computation. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a *slicing criterion*.

Definition 16 *A* static slicing criterion *of a program P consists of a pair $C = (p, V_s)$, where p is a statement in P and V_s is a subset of the variables in P.*

A slicing criterion $C = (p, V_s)$ determines a projection function which selects from any state trajectory only the ordered pairs starting with p and restricts the variable-to-value mapping function σ to only the variables in V_s.

Definition 17 *Let $C = (p, V_s)$ be a static slicing criterion of a program P and $T =< (p_1, \sigma_1), (p_2, \sigma_2), ..., (p_k, \sigma_k) >$ a state trajectory of P on input I. $\forall i, 1 \leq i \leq k$:*

$$Proj'_C(p_i, \sigma_i) = \begin{cases} \lambda & if p_i \neq p \\ < (p_i, \sigma_i | V_s) > & if p_i = p \end{cases}$$

where $\sigma_i|V_s$ is σ_i restricted to the domain V_s, and λ is the empty string.

The extension of $Proj'$ to the entire trajectory is defined as the concatenation of the result of the application of the function to the single pairs of the trajectory:

$$Proj_C(T) = Proj'_C(p_1, \sigma_1)...Proj'_C(p_k, \sigma_k)$$

A program slice is therefore defined behaviorally as any subset of a program which preserves a specified projections in its behavior.

Definition 18 *A* static slice *of a program P on a static slicing criterion $C = (p, V_s)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts, on input I, with state trajectory T, then P' also halts, with the same input I, with the trajectory T', and $Proj_C(T) = Proj_C(T')$.*

The task of computing program slices is called *program slicing*.
Weiser defined a program slice S as a reduced, *executable program* obtained from a program P removing statements, such that S preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point.
Executable means that the slice is not only a closure of statements, but also can be compiled and run. *Non-executable* slices are often smaller and thus more helpful in program comprehension.
The slices mentioned so far are computed by gathering statements and control predicates by way of a *backward traversal* of the program, starting at the slicing criterion. Therefore, these slices are referred to as **backward slices** [Tip95]. In [BC85], Bergeretti and Carré were the first to define a notion of a *forward* slice. A **forward slice** is a kind of ripple effect analysis, this is, it consists of all statements and control predicates dependent on the slicing criterion.

A statement is dependent of the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine if the statement under consideration is executed or not.

Both *backward* or *forward* slices are classified as *static* slices. **Static** means that only statically available information is used for computing slices, this is, all possible executions of the program are taken into account; no specific input I is taken into account.

Since the original version proposed by Weiser [Wei81], various slightly different notions of program slices, which are not static, have been proposed, as well as a number of methods to compute slices. The main reason for this diversity is the fact that different applications require different program properties of slices.

The last two concepts presented (*dicing* and *chopping* in section 2.1.9 and 2.1.10, respectively) are two variations on the slicing theme but very related to slicing.

Listing 2.2 emphasizes the variable *sum* (in red color) and the variables affect by its value (in blue color).

Listing 2.2: Program with *sum* variable emphasized

```
1  main() {
2      int a, test, n, i, posprod, negprod, possum, negsum, sum, prod;
3      scanf("%d",&test);  scanf("%d",&n);
4      i = posprod = negprod = 1;
5      possum = negsum = 0;
6      while (i <= n) {
7          scanf("%d",&a);
8          if (a > 0) {
9              possum += a;
10             posprod *= a;
11         }
12         else if (a < 0) {
13             negsum -= a;
14             negprod *= (-a);
15         }
16         else if (test) {
17             if (possum >= negsum) {
18                 possum = 0;
19             }
20             else { negsum = 0; }
21             if (posprod >= negprod) {
22                 posprod = 1;
23             }
24             else {
25                 negprod = 1;
26             }
27         }
28         i++;
29     }
30     if (possum >= negsum) {
31         sum = possum;
32     }
33     else { sum = negsum; }
```

```
34      if ( posprod >= negprod) {
35          prod = posprod;
36      }
37      else { prod = negprod; }
38      printf("Sum: %d\n", sum);
39      printf("Product: %d\n", product);
40  }
```

At a first glance, if we only focus at variable *sum* in program 2.1 it is easy to infer that its value depends on values of *possum* and *negsum*. Listing 2.3 shows a static slice of the program 2.1 on the slicing criterion $C = (38, sum)$[1].

Listing 2.3: A static slice of program 2.1

```
1   main() {
2       int a, test, n, i, possum, negsum, sum;
3       scanf("%d",&test);  scanf("%d",&n);
4       i = 1;
5       possum = negsum = 0;
6       while (i <= n) {
7           scanf("%d",&a);
8           if (a > 0) {
9               possum += a;
10          }
11          else if (a < 0) {
12              negsum -= a;
13          }
14          else if (test) {
15              if (possum >= negsum) {
16                  possum = 0;
17              }
18              else { negsum = 0; }
19          }
20          i++;
21      }
22      if (possum >= negsum) {
23          sum = possum;
24      }
25      else { sum = negsum; }
26      printf("Sum: %d\n", sum);
27  }
```

2.1.3 Dynamic slicing

Korel and Laski [KL88, KL90] proposed an alternative slicing definition, named *dynamic slicing*, where a slice is constructed with respect to only one execution of the program corresponding just to one given input. It does not include the statements that have no relevance for that particular input.

Definition 19 *A* dynamic slicing criterion *of a program P executed on input I is a triple $C = (I, p, V_s)$ where p is a statement in P and V_s is a subset of the variables in P.*

[1] Whenever not ambiguous, statements will be referred by their line numbers.

Definition 20 *A* dynamic slice *of a program* P *on a dynamic slicing criterion* $C = (I, p, V_s)$ *is any syntactically correct and executable program* P' *obtained from* P *by deleting zero or more statements, and whenever* P *halts, on input* I*, with state trajectory* T*, then* P' *also halts, on the same input* I*, with state trajectory* T'*, and* $Proj_{(p,V_s)}(T) = Proj_{(p,V_s)}(T')$.

Due to run-time handling of arrays and pointer variables, dynamic slicing treats each element of an array individually, whereas static slicing considers each definition or use of any array element as a definition or use of the entire array [JZR91]. Similarly, dynamic slicing distinguishes which objects are pointed to by pointer variables during a program execution.
In section 2.3, this concept of *dynamic slicing* will be detailed and algorithms for its implementation will be also presented.

Listing 2.4 shows a dynamic slice of the program 2.1 on the slicing criterion $C = (I, 38, sum)$ where $I =< (test, 0), (n, 2), (a_1, 0), (a_2, 2) >$[2].

Listing 2.4: A dynamic slice of program 2.1

```
1  main() {
2      int a, test, n, i, possum, negsum, sum;
3      scanf("%d",&test);  scanf("%d",&n);
4      i = 1;
5      possum = negsum = 0;
6      while (i <= n) {
7          scanf("%d",&a);
8          if (a > 0) {
9              possum += a;
10         }
11         i++;
12     }
13     if (possum >= negsum) {
14         sum = possum;
15     }
16     printf("Sum: %d\n", sum);
17 }
```

In the first loop iteration, hence the value of a is zero and so none of the statements in the if expression is executed, the whole conditional branch is excluded from the program slice. However, at the second loop iteration, the if statement is executed and so is included in the program slice, being excluded the else statements and the last else if statements.
Notice that the dynamic slicing in Listing 2.4 is a subprogram of the static slice in Listing 2.3 — all the statements that will never be executed, under the values of the input, are excluded.

2.1.4 Quasi-static slicing

Venkastesh introduced in 1991 the *quasi-static* slicing in [Ven91], which is a slicing method between static slicing and dynamic slicing. A quasi-static slice is constructed with respect to some values of the input data provided to the program. It is used to analyze the behavior of the program when some input variables are fixed while others vary.

[2]The subscripts refer to different occurrences of the input variable a within the different loop iterations

Definition 21 *A* quasi-static slicing criterion *of a program P is a quadruple $C = (V_i', I', p, V_s)$ where p is a statement in P; V_i is the set of input variables of a program P and $V_i' \subseteq V_i$; and I' is the input data just for the subset of variables in V_i'.*

Definition 22 *A* quasi-static slice *of a program* P *on a quasi-static slicing criterion $C = (V_i', I', p, V)$ is any syntactically correct and executable program* P' *that is obtained from* P *by deleting zero or more statements, and whenever* P *halts, on input* I*, with state trajectory* T*, then* P' *also halts, on input* I*, with state trajectory* T'*, and $Proj_{(p,V)}(T) = Proj_{(p,V)}(T')$.*

Definition 23 *Let V_i be the set of input variables of a program P and $V_i' \subseteq V_i$. Let I' be an input for the variables in V_i'. A* completion *I of I' is any input for the variables in V_i, such that $I' \subseteq I$.*

Each completion I of I' identifies a trajectory T. We can associate I' with the set of trajectories that are produced by its completions. A quasi-static slice is any subset of the program which reproduces the original behavior on each of these trajectories.
It is straightforward to see that the quasi-static slicing includes both the static and dynamic slicing's. Indeed, when the set of variables V_i' is empty, quasi-static slicing reduces to static slicing, while for $V_i' = V_i$ a quasi-static slice coincides with a dynamic slice.

According to *De Lucia* in [Luc01], the notion of quasi static slicing is closely related to *partial evaluation* or *mixed computation* [BJE88], a technique to specialize programs with respect to partial inputs. By specifying the values of some of the input variables, constant propagation and simplification can be used to reduce expressions to constants. In this way, the values of some program predicates can be evaluated, thus allowing the deletion of branches which are not executed on the particular partial input. Quasi static slices are computed on specialized programs.
As told above, the need for quasi-static slicing arises from applications where the value of some input variables is fixed while the behavior of the program must be analyzed when other input values vary.
Listing 2.5 shows a quasi-static slice of the program 2.1 on the slicing criterion $C = (I', 38, sum)$ where $I' =< (test, 0) >$.

Listing 2.5: A quasi-static slice of program 2.1

```
1  main() {
2      int a, test, n, i, possum, negsum, sum;
3      scanf("%d",&test);  scanf("%d",&n);
4      i = 1;
5      possum = negsum = 0;
6      while (i <= n) {
7          scanf("%d",&a);
8          if (a > 0) {
9              possum += a;
10         }
11         else if (a < 0) {
12             negsum -= a;
13         }
14         i++;
```

```
15      }
16      if (possum >= negsum) {
17          sum = possum;
18      }
19      else { sum = negsum; }
20      printf("Sum: %d\n", sum);
21  }
```

Hence the value of variable *test* is zero, the else if branch is excluded from static slice. All the other conditional branches stay as part of the final quasi-static slice.

2.1.5 Conditioned slicing

Canfora et al presented the *conditioned slicing* in [CCL98]. A conditioned slice consists of a subset of program statements which preserves the behavior of the original program with respect to a slicing criterion for any set of program executions. The set of initial states of the program that characterize these executions is specified in terms of a first order logic formula on the input.

Definition 24 *Let V_i be the set of input variables of a program P, and F be a first order logic formula on the variables in V_i. A* conditioned slicing criterion *of a program P is a triple $C = (F(V_i), p, V_s)$ where p is a statement in P and V_s is the subset of the variables in P which will be analyzed in the slice.*

Definition 25 *Let V_i be a set of input variables of a program P and $F(V_i)$ be a first order logic formula on the variables in V_i. A* satisfaction *for $F(V_i)$ is any partial input I to the program for the variables in V_i that satisfies the formula F. The* satisfaction set *$S(F(V_i))$ is the set of all possible satisfactions for $F(V_i)$.*

If V_i' is a subset of the input variables of the program P and $F(V_i')$ is a first order logic formula on the variables in V_i', each completion $I \in S(F(V_i))$ of $I' \in S(F(V_i))$, identifies a trajectory T. A conditioned slice is any subset of the program which reproduces the original behavior on each of these trajectories.

Definition 26 *A* conditioned slice *of a program* P *on a conditioned slicing criterion $C = (F(V_i), p, V_s)$ is any syntactically correct and executable program* P' *such that: P' is obtained from* P *by deleting zero or more statements; whenever* P *halts, on input* I*, with state trajectory* T*, where $I \in C(I', V_i')$, $I' \in S(F(V_i))$, V_i' is the set of input variables of P, and S is the satisfaction set, then* P' *also halts, on input* I*, with state trajectory* T'*, and $Proj_{(p,V_s)}(T) = Proj_{(p,V_s)}(T')$.*

A conditioned slice can be computed by first simplifying the program w.r.t. the condition on the input (i.e., discarding infeasible paths with respect to the input condition) and then computing a slice on the reduced program. A symbolic executor [Kin76] can be used to compute the reduced program, also called conditioned program in [CCLL94]. Although the identification of the infeasible paths of a conditioned program is in general an *undecidable* problem, in most cases implications between conditions can be automatically evaluated by

a theorem prover. In [CCL98] conditioned slices are interactively computed: the software engineer is required to make decisions that the symbolic executor cannot make.

Conditioned slicing allows a better decomposition of the program giving human readers the possibility to analyze code fragments with respect to different perspectives.

Actually, conditioned slicing is a framework of statement deleting[3] based methods, this is, the conditioned slicing criterion can be specified to obtain any form of slice.

Listing 2.6 shows a conditioned slice of the program 2.1 on the slicing criterion $C = (F(V_i), 38, sum)$ where $V_i = \{n\} \bigcup_{1 \leq i \leq n} \{a_i\}$ and $F(V_i) = \forall i, 1 \leq i \leq n, a_i > 0$. The condition F imposes that all input values for the variable a are positive. This allows to be discard from the static slice of Listing 2.1 all the statements dependent on the condition $a < 0$ or $a == 0$.

Listing 2.6: A conditioned slice of program 2.1

```
1  main() {
2      int a, test, n, i, possum, negsum, sum;
3      scanf("%d",&test);  scanf("%d",&n);
4      i = 1;
5      possum = negsum = 0;
6      while (i <= n) {
7          scanf("%d",&a);
8          if (a > 0) {
9              possum += a;
10         }
11         i++;
12     }
13     if (possum >= negsum) {
14         sum = possum;
15     }
16     printf("Sum: %d\n", sum);
17 }
```

2.1.6 Simultaneous dynamic slicing

Hall proposed the *simultaneous dynamic slicing* in [Hal95], which computes slices with respect to a set of program executions. This slicing method is called *simultaneous dynamic program slicing* because it extends dynamic slicing and simultaneously applies it to a set of test cases, rather than just one test case.

Definition 27 *Let* $\{T_1, T_2, ..., T_k\}$ *be a set of trajectories of length* l_1, l_2, l_k*, respectively, of a program* P *on input* $\{I_1, I_2, ..., I_k\}$*. A* simultaneous dynamic slicing criterion *of* P *executed on each of the input* I_j*,* $1 \leq j \leq k$*, is a triple* $C = (\{I_1, I_2, ..., I_k\}, p, V_s)$ *where* p *is a statement in* P *and* V_s *is a subset of the variables in* P*.*

Definition 28 *A* simultaneous dynamic slice *of a program* P *on a simultaneous dynamic slicing criterion* $C = (\{I_1, I_2, ..., I_k\}, p, V)$ *is any syntactically correct and executable program* P' *that is obtained from* P *by deleting zero or more statements, and whenever* P *halts, on*

[3] Statement deletion means deleting a statement or a control predicate from a program.

input I_j, $1 \leq j \leq m$, *with state trajectory* T_j, *then* P' *also halts, on input* I_j, *with state trajectory* T'_j, *and* $Proj_{(p,V_s)}(T_j) = Proj_{(p,V_s)}(T'_j)$.

A simultaneous program slice on a set of tests is not simply given by the union of the dynamic slices on the component test cases.

In [Hal95], Hall proposed an iterative algorithm that, starting from an initial set of statements, incrementally builds the simultaneous dynamic slice, by computing at each iteration a larger dynamic slice.

Listing 2.7 shows a simultaneous dynamic slice of the program 2.1 on the slicing criterion $C = (\{I_1, I_2\}, 38, sum)$ where $I_1 =< (test, 0), (n, 2), (a_1, 0), (a_2, 0) >$ and $I_2 =< (test, 1), (n, 2), (a_1, 0), (a_2, 2) >$.

Listing 2.7: A simultaneous dynamic slice of program 2.1

```
1  main() {
2      int a, test, n, i, possum, negsum, sum;
3      scanf("%d",&test);  scanf("%d",&n);
4      i = 1;
5      possum = negsum = 0;
6      while (i <= n) {
7          scanf("%d",&a);
8          if (a > 0) {
9              possum += a;
10         }
11         else if (a < 0) {}
12         else if (test) {
13             if (possum >= negsum) {
14                 possum = 0;
15             }
16         }
17         i++;
18     }
19     if (possum >= negsum) {
20         sum = possum;
21     }
22     printf("Sum: %d\n", sum);
23 }
```

2.1.7 Union slicing

Beszedes et al. [BFS+02, BG02] introduced the concept of *union slice* and the computing algorithm. A union slice is the union of dynamic slices for a finite set of test cases; actually is very similar to simultaneous dynamic program slicing. A union slice is an approximation of a static slice and is much smaller than the static one.

The *union slicing criterion* is the same as the considered in the simultaneous dynamic slicing.

Definition 29 *An* union slice *of a program* P *with different executions using the inputs* $X = \{I_1, I_2, ..., I_n\}$, *w.r.t. a slicing criterion* $C = (X, p, V_s)$, *is defined as follows:*

$$UnionSlice(X, p, V_s) = \bigcup_{I_k \in X} DynSlice(I_k, p, V_s)$$

where $DynSlice(I_k, i, V_s)$ contains those statements that influenced the values of the variables in V at the specific statement p.

Combined with static slices, the union slices can help to reduce the size of program parts that need to be investigated by concentrating on the most important parts first. The authors performed a series of experiments with three medium size C programs. The results suggest that union slices are in most cases far smaller than the static slices, and that the growth rate of union slices (by adding more test cases) significantly declines after several representative executions of the program. Thus, union slices are useful in software maintenance.

Daninic et al [DLH04] presented an algorithm for computing executable union slices, using conditioned slicing. The work showed that the executable union slices are not only applicable for program comprehension, but also for component reuse guided by software testing.

De Lucia et al [LHHK03] studied the properties of unions of slices and found that the union of two static slices in not necessarily a valid slice, based on Weiser's definition of a static slice. They argue that a way to get valid union slices is to propose algorithms that take into account simultaneously the execution traces of the slicing criteria, as in the simultaneous dynamic slicing algorithm proposed by Hall [Hal95].

Listing 2.8 shows an union slice of the program 2.1 on the slicing criterion $C = (I_1 \bigcup I_2, 38, sum)$ where $I_1 =< (test, 0), (n, 2), (a_1, 0), (a_2, 2) >$ and $I_2 =< (test, 0), (n, 3), (a_1, 1), (a_2, 0), (a_3, -1) >$.

Listing 2.8: An union slice of program 2.1

```
1  main() {
2      int a, test, n, i, possum, negsum, sum;
3      scanf("%d",&test);  scanf("%d",&n);
4      i = 1;
5      possum = negsum = 0;
6      while (i <= n) {
7          scanf("%d",&a);
8          if (a > 0) {
9              possum += a;
10         }
11         else if (a < 0) {
12             negsum -= a;
13         }
14         i++;
15     }
16     if (possum >= negsum) {
17         sum = possum;
18     }
19     printf("Sum: %d\n", sum);
20 }
```

2.1.8 Other concepts

There are a number of other related approaches that use different definitions of slicing to compute subsets of program statements that exhibit a particular behavior. All these approaches add information to the slicing criterion to reduce the size of the computed slices.

Constrained slicing

Field et al introduce in [FRT95] the concept of **constrained slice** to indicate slices that can be computed w.r.t. any set of constraints. Their approach is based on an intermediate representation for imperative programs, named PIM, and exploits graph rewriting techniques based on dynamic dependence tracking that model symbolic execution. The slices extracted are not executable. The authors are interested in the semantic aspect of more complex program transformations rather than in simple statement deletion.

Amorphous slicing

Harman et al introduced **amorphous slicing** in [HBD03]. Amorphous slicing removes the limitation of *statement deletion* as the only means of simplification. Like a traditional slice, an amorphous program serves a projection of the semantics of the original program from which it is constructed. However, it can be computed by applying a broader range of transformation rules, including statement deletion.

Hybrid slicing

Gupta et al presented the **hybrid slicing** in [GSH97], which incorporate both static and dynamic information. They proposed a slicing technique that exploits information readily available during debugging when computing slices statically.

2.1.9 Dicing

Dicing was a concept first introduced by *Lyle et al* in [LW86]. A **program dice** is defined as the set difference between the static slices of an incorrect variable and that of a correct variable, this is, the set of statements that potentially affect the computation of incorrect variable while do not affect the computation of the correct one. It is a fault location technique for further reducing the number of statements that need to be examined when debugging.
Later, in 1993, *Chen et al* [CC93], have proposed the dynamic program dicing.

2.1.10 Chopping

Chopping, introduced by Jackson [JR94], is a generalization of slicing. Although expressible as a combination of intersections and unions of *forward* and *backward* slices, *chopping* seems to be a fairly natural notion in its own right.
Two sets of instances form the criterion: source, a set of definitions, and sink, a set of uses. Chopping a program identifies a subset of its statements that account for all influences of the source on the sink. A conventional backward slice is a chop in which all the sink instances belong to the same site, and the source set contains every variable at every site. A chop is confined to a single procedure. The instances in source and sink must be within the procedure, and chopping only identifies statements in the text of the procedure itself.

After a survey of all the variants of the original slicing concept (static slicing) and its most important variant — the *dynamic slicing* — will be detailed in section 2.2 and 2.3 the original approach.

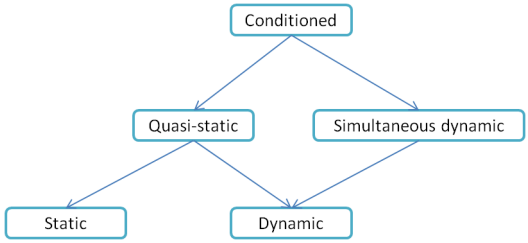

Figure 2.1: Relationships between program slicing models

2.1.11 Relationships among program slicing models

The slicing models discussed in the previous section can be classified according to a partial ordering relation, called *subsume relation*, based on the sets of program inputs specified by the slicing criteria. Indeed, for each of these slicing models, a slice preserves the behavior of the original program on all the trajectories identified by the set of program inputs specified by the slicing criterion.
In 1998, Canfora et al [CCL98] presented the concept of **subsume relation**.

Definition 30 *A program slicing model* SM_1 subsumes *a program slicing model* SM_2 *if for each slicing criterion defined according to* SM_2 *there exists an equivalent slicing criterion defined according to* SM_1 *that specifies the same set of program inputs.*

A slicing model is **stronger** than the slicing models it subsumes, because it is able to specify and compute slices w.r.t. a broader set of slicing criteria. Consequently, any slice computed according to a slicing model can also be computed with a stronger model. The subsume relation defines an hierarchy on the statement deletion based.
According to their definition of subsumes relation, the conditioned slicing subsumes any other model. Figure 2.1.11 shows the subsume hierarchy.
It is argued that the set of slicing models (static, dynamic, quasi-static, simultaneous, conditioned) is partially ordered with respect to subsume relation:

- Quasi-static slicing subsumes static slicing;
- Quasi-static slicing subsumes dynamic slicing;
- Simultaneous dynamic slicing subsumes dynamic slicing;
- Conditioned slicing subsumes quasi-static;
- Conditioned slicing subsumes simultaneous dynamic slicing; and
- There is no relation between static slicing and dynamic slicing and between quasi-static slicing and simultaneous dynamic slicing.

In an attempt to formalize slicing concepts, Binkley et al defined in [BDG+04, BDG+06] subsume relation in terms of syntactic ordering and semantic equivalence. This formalization establish a precise relationship between various forms of dynamic slicing and static slicing, counteracting the Canfora affirmation that there is no relation between static and dynamic slicing.

2.1.12 Methods for Program Slicing

According to Tip [Tip95] classification, there are three major kinds of approaches in program slicing:

- Dataflow equations;
- Information-flow relations; and
- Dependence graph based approaches.

The Weiser's original approach is a kind of method based on iteration of **dataflow equations**. In this approach, slices are computed in an iterative process, by computing consecutive sets of relevant statements for each node in the CFG. The algorithm first computes directly relevant statements for each node in the CFG, and then indirectly relevant statements are gradually added to the slice. The process stops when no more relevant statements are found.

Information flow relations for programs presented by Bergeretti [BC85] can also be used to compute slices. In this kind of approach, several types of relations are defined and computed in a syntax-directed, bottom-up manner. With these information-flow relations, slices can be obtained by relational calculus.

The most popular kind of slicing approach, the **dependence graph** approach, was proposed by Ottenstein and Ottenstein [OO84] and restate the problem of static slicing in terms of a reachability problem in a PDG. A slicing criterion is identified with a vertex in the PDG, and a slice correspond to all PDG vertices from which the vertex under consideration can be reached. Usually, the data dependencies used in program slicing are flow dependencies corresponding to the DEF and REF sets defined in section 1.1.

In these approaches using PDG, slicing can be divided into two steps. In the first step, the dependence graph of the program are constructed, and then the algorithm produce slices by doing graph reachability analysis over it.

As defined in section 1.1, a dependence graph is a directed graph using vertexes to represent program statements and edges to represent dependencies. So, the graph reachability analysis can be done by traversing edges on the dependence graph from a node representing the slicing criteria. A dependence graph can represent not only dependencies but also other relations such as process communications and so on. Different slices can be obtained by constructing different dependence graphs.

2.2 Static slicing

In this section it is discussed the basic static slicing approaches. Each subsection is divided according to the methods presented in previous section.

2.2.1 Basic slicing algorithms

In this subsection, are presented algorithms for static slicing of structured programs without non-scalar variables, procedures and interprocess communication.

Dataflow equations

The original concept of program slicing [Wei81] was first proposed as the iterative solution to a dataflow problem specified using the program's control flow graph (CFG).

Definition 31 *A slice is* statement-minimal *if no other slice for the same criterion contains fewer statements.*

Weiser argues that statement-minimal slices are not necessarily unique, and that the problem of determining statement-minimal slices is undecidable.
Many researchers have investigated this problem, and various approaches result in good approximations. Some techniques are based on data-flow equations [KL88, LR87, Wei81] while others use graph representations of the program [AH90, Agr94, BH93a, Bin93, CF94, HRB88, JR94].
An approximation of statement-minimal slices are computed in an iterative process [Tip95], by computing consecutive sets of relevant variables for each node in the CFG. First, the directly relevant variables are determined, by only taking data dependencies into account. Below, the notation $i \rightarrow_{\mathsf{CFG}} j$ indicates the existence of an edge in the CFG from node i to node j. For a slicing criterion $C = (n, V)$ (where n denotes the number line), the set of directly relevant variables at node i of the CFG, $R^0_C(i)$ is defined as follows:

- $R^0_C(i) = V$, when $i = n$;
- For every $i \rightarrow_C FGj$, $R^0_C(i)$ contains all variables v such that either
  - $v \in R^0_C(j)$ and $v \notin DEF(i)$;
  - $v \in REF(i)$ and $DEF(i) \bigcap R^0_C(j) \neq \emptyset$.

From this, a set of *directly relevant statements*, S^0_C, is derived. S^0_C is defined as the set of all nodes i which define a variable v that is relevant at a successor of i in the CFG:

$$S^0_C = \{i | DEF(i) \bigcap R^0_C(j) \neq \emptyset, i \rightarrow_{\mathsf{CFG}} j\}$$

Information-flow relations

Bergeretti and Carré, in [BC85], proposed another approach that defines slices in terms of information-flow relations derived from a program in a syntax-directed fashion. The authors have defined a set of information-flow relations for sequences of statements, conditional statements and loop statements.

Dependence graph based approach

It were Karl Ottenstein and Linda Ottenstein [OO84] the first of many to define slicing as a reachability problem in a dependence graph representation of a program. They use the PDG for static slicing of single-procedure programs. The statements and expressions of a program constitute the vertices of a PDG, and edges correspond to data dependencies and control dependencies between statements (section 1.1). The key issue is that the partial ordering of

the vertices induced by the dependence edges must be obeyed so as to preserve the semantics of the program.

In the PDG's of Horwitz et al [HPR88, HRB88] a distinction is made between loop-carried and loop-independent flow dependencies. It is argued that the PDG variant of [HPR88] is minimal in the sense that removing any of the dependence edges, or disregarding the distinction between loop-carried and loop-independent flow edges would result in inequivalent programs having isomorphic PDGs.

The PDG variant considered in [OO84] shows considerably more detain than that of [HRB88]. In particular, there is a vertex for each (sub)expression in the program, and file descriptors appear explicitly as well.

In all dependence graph based approaches, the slicing criterion is identified with a vertex v in the PDG.

2.2.2 Slicing programs with arbitrary control flow

Dataflow equations

In intraprocedural program slicing, the critical problem is to determine which predicates to be included in the slice when the program contains jump statements.

Lyle reports in [Lyl84] that the original slicing algorithm proposed by Weiser was able to determine which predicates to be included in the slice even when the program contains jump statements, It did not, however, make any attempt to determine the relevant jump statements themselves to be included in the slice. Thus, Weiser's algorithm may yields incorrect slices in the presence of unstructured control flow. Lyle presents a conservative solution for dealing with `goto` statements. His algorithm produces slices including every goto statement that has a non-empty set of active variables associated with it.

Gallagher [Gal90, GL91b] also use a variation of Weiser's method. In the algorithm, a `goto` statement is included in the slice if it jumps to a label of an included statement.

Jian, Zhou and Robson [JZR91] have also proposed a set of rules to determine which jump statements to include in a slice.

Agrawal shows in [Agr94] that either Gallagher algorithm or Jian et al algorithm does not produce correct slices in all cases.

Dependence graph based approach

Ball and Horwitz [BH93a, Bal93] and Choi and Ferrante [CF94] discovered independently that conventional PDG-based slicing algorithms produce incorrect results in the presence of unstructured control flow: slices may compute values at the criterion that differ from what the original program does. These problems are due to the fact that the algorithms do not determine correctly when unconditional jumps such as break, goto, and continue statements are required in a slice. They proposed two similar algorithms to determine the relevant jump statements to include in a slice. Both of them require that jumps be represented as pseudo-predicates and the control dependence graph of a program be constructed from an augmented flow graph of the program. However, Choi and Ferrante distinguish two disadvantages of the slicing approach based on augmented PDGs (APDG). First, APDGs requires more space than conventional PDGsand their construction takes more time. Second, non-executable control

dependence edges gives rise to spurious dependencies in some cases.

In their second approach, Choi and Ferrante also proposed another algorithm to construct an executable slice in the presence of jump statements when a "slice" is not constrained to be a subprogram of the original one. The algorithm constructs new jump statements to add to the slice to ensure that other statement in it are executed in the correct order.

The main difference between the approach by Ball and Horwitz and the first approach of Choi and Ferrante is that the latter use a slightly more limited example language: conditional and unconditional goto's are present, but no structured control flow constructs. Although Choi and Ferrante argue that these constructs can be transformed into conditional and unconditional goto's. Ball and Horwitz show that, for certain cases, this results in overly large slices.

Both groups have been proposed two formal proofs to show that their algorithms compute correct slices.

Agrawal [Agr94] proposed an algorithm has the same precision as that of the above two algorithms. He observes that a conditional jump statement of the form `if P then goto L` must be included in the slice if the predicate `P` is in the slice because another statement in the slice is control dependent on it. This algorithm is appealing in that it leaves the flow-graph and the PDG of the program intact and uses a separate graph to store the additional required information. It lends itself to substantial simplification, when the program under consideration is a structured program. Also, the simplified algorithm directly leads to a conservative approximation algorithm that permits on-the-fly detection of the relevant jump statements while applying the conventional slicing algorithms.

Harman and Danicic [HD98] defined an extension of Agrawal's algorithm that produces smaller slices by using a refined criterion for adding jump statements (from the original program) to the slice computed using Ottenstein's algorithm for building and slicing the PDG [OO84].

Kumar and Horwitz [KH02] extended the previous work on program slicing by providing a new definition of "correct" slices, by introducing a representation for C-style switch statements, and by defining a new way to compute control dependencies and to slice a PDG so as to compute more precise slices of programs that include jumps and switches.

2.2.3 Interprocedural slicing methods

Dataflow equations

Weiser describes a two-step approach for computing interprocedural static slices in [Wei81]. In the first step, a slice is computed for the procedure P which contains the original slicing criterion. The effect of a procedure call on the set of relevant variables is approximated using interprocedural summary information [Bar78]. For a procedure P, this information consists of a set $MOD(P)$ of variables that may be modified by P, and a set of $USE(P)$ of variables that may be used by P, taken into account any procedures called by P. The fact of Weiser's algorithm does not take into account which output parameters are dependent on which input parameters is a cause of imprecision. This is illustrated in program 2.9 listed below. The Weiser's interprocedural slicing algorithm will compute the slice listed in program 2.10. This slice contains the statement `int a = 17;` due to the spurious dependence between variable a before the call, and variable d after the call.

Listing 2.9: Interprocedural sample program

```
1  void simpleAssign(int v, int w, int x, int y) {
2      x = v;
3      y = w;
4  }
5
6  main() {
7      int a = 17;
8      int b = 18;
9      int c, d;
10     simpleAssign(a,b,c,d);
11     printf("Result: %d\n", d);
12 }
```

Listing 2.10: Weiser's Interprocedural slice of program 2.9

```
1  void simpleAssign(int v, int w, int x, int y) {
2      y = w;
3  }
4
5  main() {
6      int a = 17;
7      int b = 18;
8      int c, d;
9      simpleAssign(a,b,c,d);
10 }
```

In the second step of Weiser's algorithm new criteria are generated for:

a) Procedures Q called by P;

b) Procedures R that call P.

The two steps described above are repeated until no new criteria occur. The criteria of a) consists of all pairs (n_Q, V_Q) where n_Q is the last statement of Q and V_Q is the set of relevant variables in P which is in the scope of Q (where formals are substituted by actual). The criteria of b) consists of all pairs (n_R, V_R) such that N_R is a call to P in R, and V_R is the set of relevant variables at the first statement of P which is in the scope of R (actuals are substituted by formals). The generation of new criteria is formalized by way of functions $UP(\mathcal{S})$ and $DOWN(\mathcal{S})$ which map a set $\mathcal{S}$ of slicing criteria in a procedure P to a set of criteria in procedures that call P, and a set of criteria in procedures called by P, respectively. The closure $UP \bigcup DOWN^*(\{C\})$ contains all criteria necessary to compute an interprocedural slice, given an initial criterion C. Worst-case assumptions have to be made when a program calls external procedures, and the source code is unavailable.

Horwitz, Reps and Binkley report that Weiser's algorithm for interprocedural slicing is unnecessarily inaccurate because of what they refer to as the "calling context" problem, i.e., the transitive closure operation fails to account for the calling context of a called procedure. In a nutshell, the problem is that when the computation 'descends' into a procedure Q that is called from a procedure Q, not only P. This corresponds to execution paths which enter Q from P and exit Q to a different procedure P'.

Tip [Tip95] conjecture that the calling context problem of Weiser's algorithm can be fixed by observing that the criteria in the UP sets are only needed to include procedures that transitively call the procedure containing the initial criterion. Once this is done, only $DOWN$ sets need to be computed.

Hwang, Du and Chou [JDC88] proposed an iterative solution for interprocedural static slicing based on replacing recursive calls by instances of the procedure body. The slice is recomputed in each iteration until a fixed point is found (i.e., no new statement are added to a slice). This approach do not suffer from the calling context problem because expansion of recursive calls does not lead to considering infeasible execution paths. However, Reps [RHSR94, Rep96] showed that for a certain family P^k of recursive programs, this algorithm takes time $O(2^k)$, i.e., exponential in the length of the program.

Dependence graph based approach

Interprocedural slicing as a graph reachability problem requires extending of the PDG and, unlike the addition of data types or unstructured control flow, it also requires modifying the slicing algorithm. The PDG modifications represents call statements, procedure entry, parameters, and parameter passing. The algorithm change is necessary to correctly account for procedure calling context.

Horwitzs, Reps and Binkley [HRB88] introduce the notion of *System Dependence Graph* (SDG) for the dependence graphs that represents multi-procedure programs. Figure 2.2.3 shows the SDG corresponding to the program 2.11 listed below.

Listing 2.11: Example system and its SDG

```
1  void Add(int* a, int b) {
2      *a = *a + b;
3  }
4
5  void Increment(int z) {
6      Add(&z,1);
7  }
8
9  void A(int x, int y) {
10     Add(&x,y);
11     Increment(&y);
12 }
13
14 void main() {
15     int sum = 0;
16     int i = 1;
17     while (i < 11) {
18         A(&sum, i);
19     }
20     printf("Sum: %d\n", sum);
21 }
```

Interprocedural slicing can be defined as a reachability problem using the SDG, just as intraprocedural slicing is defined as a reachability problem using the PDG. The slices obtained using this approach are the same as those obtained using Weiser's interprocedural slicing

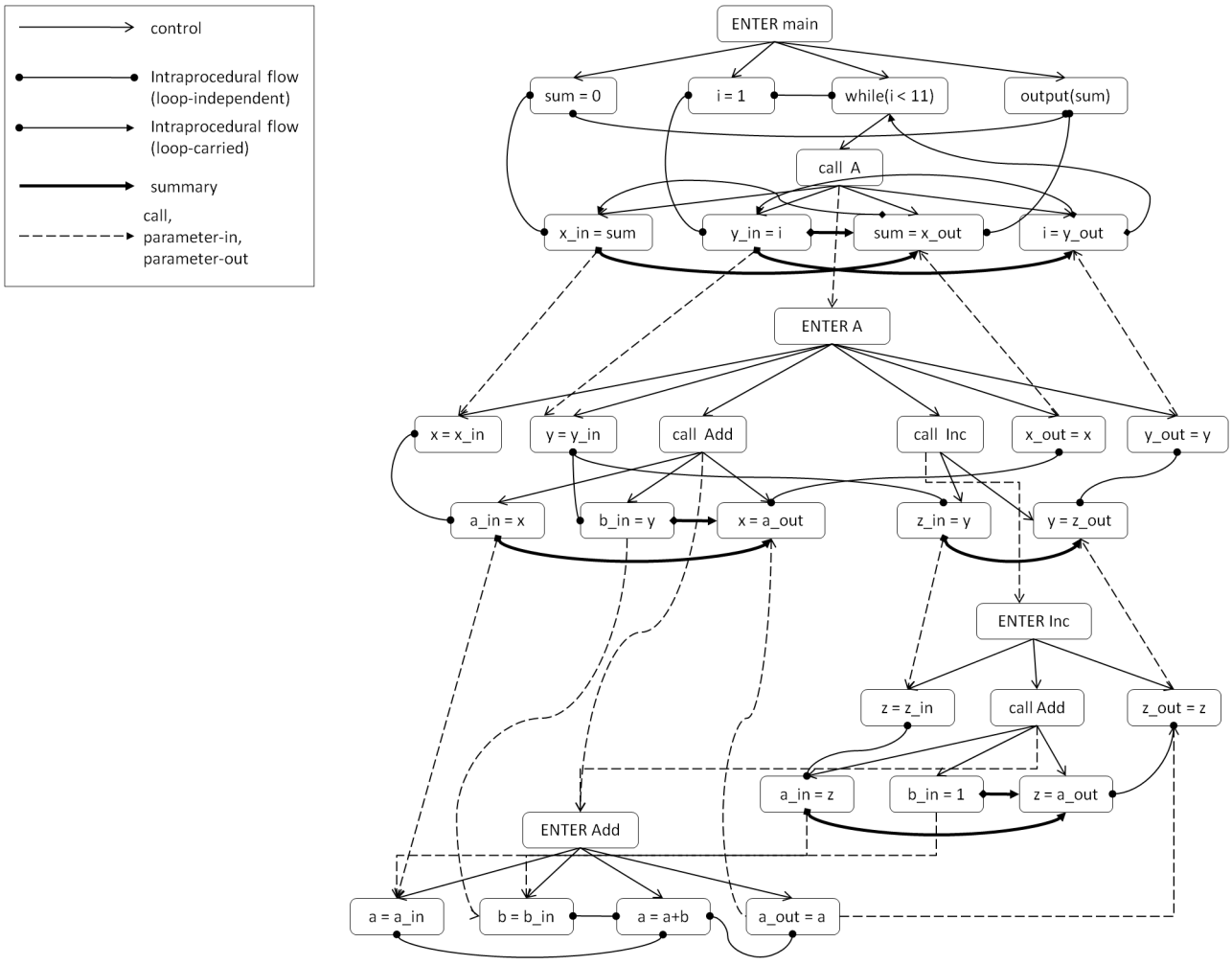

Figure 2.2: Example system and its SDG

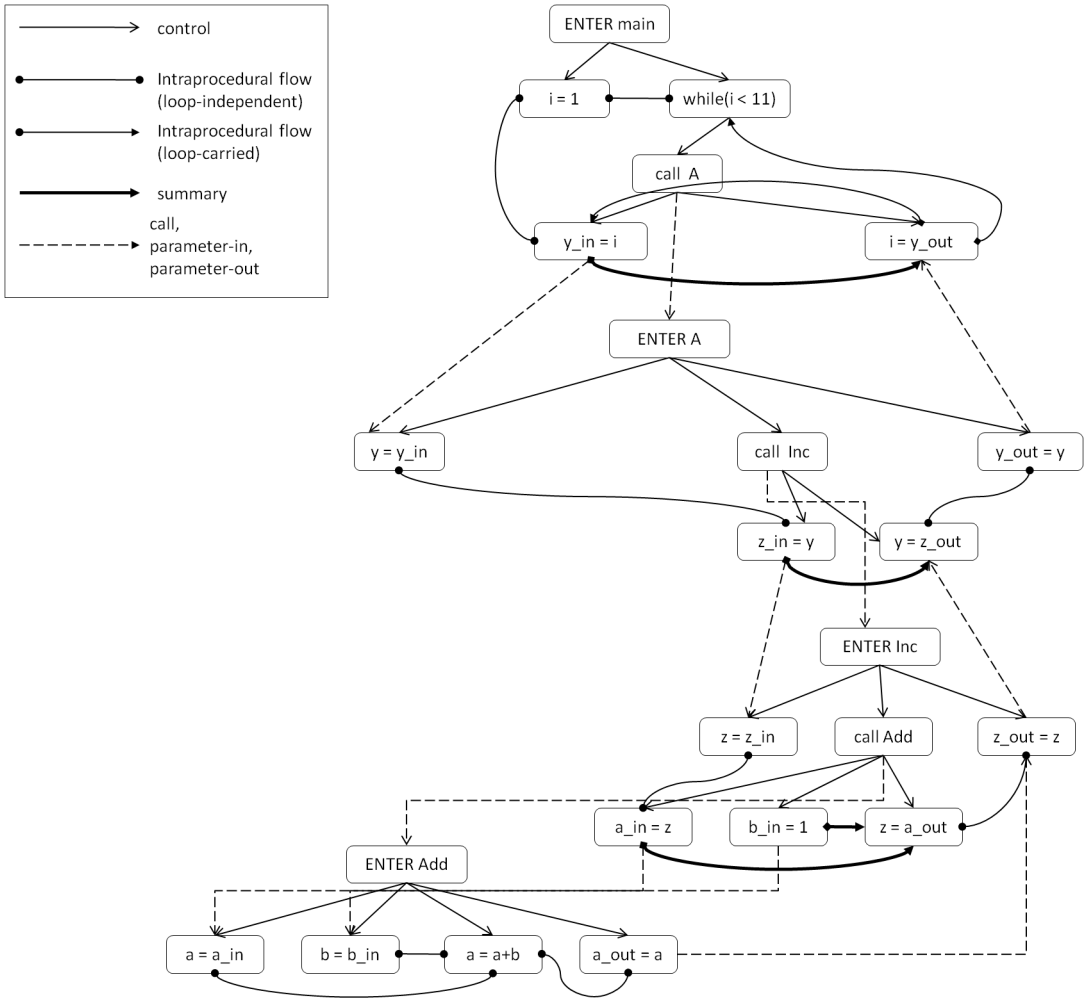

Figure 2.3: The SDG from Figure 2.2.3 sliced w.r.t. the formal-out vertex for parameter z in procedure *Increment*, together with the system to which it corresponds.

method [Wei84]. However, is approach does not produce slices that are as precise as possible, because it considers paths in the graph that are not possible execution paths. For example, there is a path in the SDG shown in Figure 2.2.3 from the vertex of procedure *main* labeled "$x_in = sum$" to the vertex of *main* labeled "$i = y_out$". However this path corresponds to procedure *Add* being called by procedure A, returning to procedure *Increment*, which is not possible. The value of i after the call to procedure A is independent of the value of *sum* before the call, and so the vertex labeled "$x_in = sum$" should not be included in the slice w.r.t. the vertex labeled "$i = y_out$". Figure 2.2.3 shows this slice.

To achieve more precise interprocedural slices, an interprocedural slice w.r.t. vertex s is computed using two passes over the graph. Summary edges permit moving across a call procedure; thus, there is no need to keep track of calling context explicitly to ensure that only legal execution paths are traversed. Both passes operate on the SDG, traversing edges to find the set of vertices that can reach a given set of vertices along certain kinds of edges. Informally, if s is in procedure P then pass 1 identifies vertices that reach s and are either

in P itself or procedures that (transitively) call P [BG96] . The traversal in pass 1 does not descend into procedures called by P or its callers. Pass 2 identifies vertices in called procedures that induce the summary edges used to move across call sites in pass 1.

The traversal in pass 1 starts from s and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but not along parameter-out edges. The traversal in pass 2 starts from all vertices reached in pass 1 and goes backwards along flow edges, control edges, summary edges, and parameter-out edges, but not along call, or parameter-in edges. The result of an interprocedural slice of a graph G with respect to vertex set S, denoted by $Slice(G, S)$, consists of the sets of vertices encountered during by pass 1 and pass 2, and the set of edges induce by this vertex set.

$Slice(G, S)$ is a subgraph of G. However it may be infeasible (i.e., it may be not the SDG of any system). The problem arises when $Slice(G, S)$ includes mismatched parameters: different call-sites on a procedure include different parameters. There are two causes of mismatches: missing actual-in vertices and missing actual-out vertices. Making such systems syntactically legal by simply adding missing parameters leaves semantically unsatisfactory systems [Bin93]. In order to include the program components necessary to compute a safe value for the parameter represented at missing actual-in vertex v, the vertices in the pass 2 slice of G taken with respect to v must be added to the original slice. A pass 2 slice includes the minimal number of components necessary to produce a semantically correct system. The addition of pass 2 slices is repeated until no further actual-in vertex mismatches exist.

The second cause of parameter mismatches is missing actual-out vertices. Because missing actual-out vertices represent dead-code no additional slicing is necessary. Actual-out mismatches are removed by simply adding missing actual-out vertices to the slice.

The details of this algorithm are given in [Bin93].

Several extensions of Horwitz-Reps-Binkley (HRB) algorithm have been presented. Lakhotia [Lak92] adapted the idea of lattice theory to interprocedural slicing and presented a slicing algorithm based on the augmented SDG in which a tag is contained for each vertex of SDG. Different from HRB algorithm, this one only need one traverse on the SDG. Binkley extended HRB algorithm to produce executable interprocedural program slices in [Bin93].

Clarke et al. [CFR+99] extended HRB algorithm to VHDL (Very High Speed Integrated Circuit Hardware Description Language), using an approach based on capturing the operational semantics of VHDL in traditional constructs. Their algorithm first maps the VHDL constructs onto traditional program language constructs and then slices using a language-independent approach.

Orso et al. [OSH01] proposed a SDG-based incremental slicing technique, in which slices are computed based on types of data dependencies. They classified the data dependencies into different types. The scope of a slice can be increased in steps, by incorporating additional types of data dependencies at each steps.

To the problem of SDG constructing, Forgacsy and Gyimóthy [FG] presented a method to reduce the SDG. Livadas and Croll [LC94] extended the SDG and proposed a method to construct SDG directly from parser trees. Their algorithm is conceptually much simpler, but it cannot handle recursion. Kiss [KJLG03] presented an approach to construct SDG from the binary executable programs and proposed an algorithm to slice on them. Sinha, Harrold and Rothermal [SHR99] extended the SDG to represented interprocedural control dependencies. Their extension is based on Augmented Control Flow Graph (ACFG), a CFG augmented with

edges to represent interprocedural control dependencies. Hisley et al. [HBP02] extended the SDG to threaded System Dependence Graph (tSDG) in order to represent non-sequential programs.

Information-flow relations

Bergeretti and Carré explains in [BC85] how the effect of procedure calls can be approximated. Exact dependencies between input and output parameters are determined by slicing the called procedure with respect to which output parameter (i.e., the computation of the μ relation for the procedure). Then, each procedure call is replaced by a set of assignments, where each output parameter is assigned to a fictitious expression that contains the input parameters it depends on. As only feasible execution paths are considered, this approach does not suffer from the calling context problem. A call to a side-effect free function can be modeled by replacing it with a fictitious expression containing all actual parameters. Note that the computed slices are not truly interprocedural since no attempt is done to slice procedures other than the main program.

2.2.4 Slicing in the presence of composite datatypes and pointers

Dependence graph based approach

When slicing, there are two approaches to handle arrays. A simple approach for arrays is to treat each array as a whole [Lyl84]. According to Lyle, any update to an element of an array is regarded as an update and a reference of the entire array. However, this approach leads to unnecessary large slices. To be more precise requires distinguishing the elements of array. And this needs dependence analysis.

The PDG variant of Ottenstein and Ottenstein [OO84] contains a vertex for each sub-expression; special *select* and *update* operators serve to access elements of an array.

Banerjee [Ban88] presented the Extended GDC Test. It can be applied to analyze the general objects (multi-dimensional arrays and nested trapezoidal loops). The test is derived from number theory. The single equation $a_1x_1+a_2x_2+...+a_nx_n = b$ has an integer solution if and only if $gdc(a_i)$ divides b. This give us an exact test for single-dimensional arrays ignoring bounds. And it can be extended to multi-dimensional arrays.

In the presence of pointers, situations may occur where two or more variables refer to the same memory location. This phenomenon is commonly called *aliasing*. Algorithms for determining potential aliases can be found in [CBC93, LR92]. Slicing in the presence of aliasing requires a generalization of the notion of data dependence to take potential aliases into account. Tip [Tip95] have presented the definition of *potentially data dependent*.

Definition 32 *A statement s is* potentially data dependent *on a statement s' if:*

i) *s defines a variable X';*

ii) *s' uses a variable X;*

iii) *X and X' are potential aliases; and*

iv) *there exists a path from s to s' in the CFG where X is not necessarily defined.*

Such paths may contain definitions to potential aliases of X.

A slightly different approach is pursued by Horwitz, Pfeiffer and Reps [HPR89a]. Instead of defining data dependence in terms of potential definitions and uses of variables, they defined this notion in terms of potential definitions and uses of *abstract memory locations.*

However, Landi [Lan92] have shown that precise pointer analysis is undecidable. So the analysis has to do a trade-offs between cost and precision. There are several dimensions that affect the trade-offs. How a pointer analysis addresses each of these dimensions helps to categorize the analysis.

Besides the data dependence, in the presence of pointers, the reaching definition also need to be changed, and the *l-valued* expression have to be taken into account.

Definition 33 *An* l-valued *expression is any expression which may occur as the left-hand side of an assignment.*

Jiang [JZR91] presented an algorithm for slicing C programs with pointers and arrays. Unfortunately, the approach appears to be flawed. There are statements incorrectly omitted, resulting in inaccurate slices.

2.3 Dynamic slicing

In this section is is discussed the dynamic slicing approaches.

2.3.1 Basic algorithms for dynamic slicing

Dataflow equations

It was Korel and Laski [KL88, KL90] who first proposed the notion of dynamic slicing. As it was defined in section 2.1.3, a dynamic slice is a part of a program that affects the concerned variable in a particular program execution. As only one execution is taken into account, dynamic program slicing may significantly reduce the size of the slice as compared to static slicing.

Most dynamic slices are computed w.r.t. an *execution history* or *trajectory*. This history records the execution of statements as the program executes. The execution of a statement produces an occurrence of the statement in the trajectory. Thus, the trajectory is a list of statement occurrences.

Two example execution histories are shown below for the program 2.12. Superscripts are used to differentiate between the occurrences of a statement. For example, statement 2 executes twice for the second execution producing 2^1 and 2^2.

In order to compute dynamic slices, Korel and Laski introduce three dynamic flow concepts which formalize the dependencies between occurrences of statements in a trajectory.

Definition 34 Definition Use *(DU). v^i DU $u^j \Leftrightarrow v^i$ appears before u^j in the execution history and there is a variable x defined at v^i, used at u^j, but not defined by any occurrence between v^i and u^j.*

Listing 2.12: Two execution histories

```
1   scanf("%d", &n);
2   for (i = 1; i < n; i++) {
3       a = 2;
4       if (c1) {
5           if (c2) {
6               a = 4;
7           }
8           else {
9               a = 6;
10          }
11      }
12      z = a;
13  }
14  printf("Result: %d\n", z);
```

Execution history 1
Input $n = 1$, $c1$ and $c2$ both true:
$< 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 12^1, 2^2, 14^1 >$

Execution history 2
Input $n = 2$, $c1$ and $c2$ false on the first iteration and true on the second:
$< 1^1, 2^1, 3^1, 4^1, 12^1, 2^2, 3^2, 4^2, 5^1, 6^1, 12^2, 2^3, 14^1 >$

This definition captures flow dependence that arise when one occurrence represent the assignment of a variable and another use of that variable. For example, in program 2.12 listed above, when $c1$ and $c2$ are both false, there is a flow dependence from statement 3 to statement 12.

Definition 35 Test Control *(TC).* v^i *TU* $u^j \Leftrightarrow$ *u is control dependent on v (in the static sense) and for all occurrences* w^k *between* v^i *and* u^j*, w is control dependent on v.*

This second definition captures control dependence. The only difference between this definition and the static control dependence definition is that multiple occurrence of predicates exist.

Definition 36 Identity Relation *(IR).* v^i *IR* $u^j \Leftrightarrow v = u$.

Dynamic slices are computed in an iterative way, by determining successive sets S^i of directly and indirectly relevant statements. For a slicing criterion $C = (I^q, p, V)$ the initial approximation S^0 contains the last definition of the variables in V in the trajectory, as well as the test actions in the trajectory on which I^q is control dependent. Approximation S^{i+1} is defined as follows:

$$S^{i+1} = S^i \bigcup A^{i+1}$$

where A^{i+1} consists of:

$$A^{i+1} = X^p | X^p \notin S^i, (X^p, Y^t) \in (DU \bigcup TC \bigcup IR) for some Y^t \in S^i, p < q$$

The dynamic slice if obtained from the fixpoint S_C of this process (as q is finite, this always exists): any statement X for an instance X^p occurs in S_C will be in the slice.
Program 2.13 shows the Korel and Laski slice of the program shown in program 2.12 taken with respect to $(3^2, 2, \{a\})$.

Listing 2.13: A dynamic slice of the program listed in program 2.12 and its execution history

```
1   scanf("%d", &n);
```

```
2     for (i = 1; i < n; i++) {
3         a = 2;
4     }
```

This slice is computed as follows: $DU = \{(1^1, 2^1), (3^1, 12^1), (6^1, 12^2), (12^2, 14^1)\}$
$TC = \{(2^1, 3^1), (2^1, 4^1), (2^1, 12^1), (2^1, 3^2), (2^1, 4^2), (2^1, 12^2), (2^2, 3^2), (2^2, 4^2), (2^2, 12^2), (4^2, 5^1), (5^1, 6^1)\}$
$IR = \{(2^1, 2^2), (2^1, 2^3), (2^2, 2^3), (3^1, 3^2), (4^1, 4^2), (12^1, 12^2)\}$
$S^0 = \{2^1\}$
$S^1 = \{1^1, 2^1\}$
$S^2 = \{1^1, 2^1\}\} = S^1$, thus the iteration ends.
$S = \{3^1\} \bigcup \{1^1, 2^1\} = \{1^1, 2^1, 3^1\}$.

Information-flow relations

Gopal [Gop91] have proposed an approach were *dynamic dependence relations* are used to compute dynamic slices. He introduces dynamic versions of Bergeretti and Carré information-flow relations. The $\overline{\lambda_S}$ relations contains all pairs (v, e) such that statement e depends on the input value of v when program S is executed. Relation $\overline{\mu_S}$ contains all pairs (e, v) such that the output value of v depends on the execution of statement e. A pair (v, v') is in the relation $\overline{\rho_S}$ if the output value of v' depends on the input value of v. In this definitions it is presumed that S is executed for some fixed input.

For empty statements, assignments, and statement sequences Gopal's relations are exactly the same as for the static case.

Dependence graph based approach

Miller and Choi [MC88] first proposed the notion of dynamic dependence graph. However, their method mainly concentrates on parallel program debugging and flowback analysis[4]

Agrawal and Horgan [AH90] developed an approach for using dependence graphs to compute non-executable dynamic slices. Their first two algorithms for computing dynamic slices are inaccurate. The initial approach uses the PDG and marks the vertices that are executed for a given test. A dynamic slice is computed by computing a static slice in the subgraph of the PDG that is induced by the marked vertices. By construction, this slice only contains vertices that are executed. This solution is imprecise because it does not detect situations where there exists a flow edge in the PDG between a marked vertex v_1 and a marked vertex v_2, but where the definitions of v_1 are not actually used at v_2.

The second approach consists of marking PDG edges as the corresponding dependencies arise during execution. Again, the slice is obtained by traversing the PDG, but this time only along marked edges. Unfortunately , this approach still produces imprecise slices in the presence of loops because an edge that is marked in some loop iteration will be present in all subsequent iterations, even when the same dependence does not recur.

This approach computes imprecise slices because it does not account for the fact that different occurrences of a statement in the execution history may be (transitively) dependent on

[4]Flowback analysis is a powerful technique for debugging programs. It allows the programmer to examine dynamic dependencies in a program's execution history without having to re-execute the program.

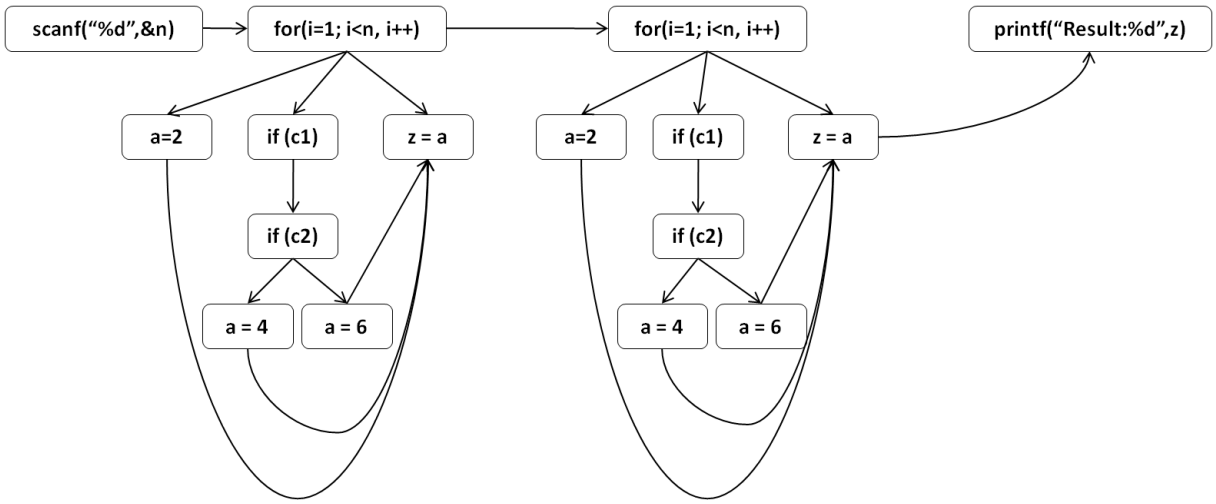

Figure 2.4: The DDG for the example listed in program 2.12.

different statements. This observation motivate their third solution: create a distinct vertex in the dependence graph for each occurrence of a statement in the execution history. This kind of graph is referred as *Dynamic Dependence Graph* (DDG). A dynamic slicing criterion is identified with a vertex in the DDG, and a dynamic slice is computed by determining all DDG vertices from which the criterion can be reached.

A dynamic slice can now be defined in terms of a variable, an execution history, and an occurrence of a statement in the execution history. The slice contains only those statements whose execution had some effect on the value of the variable at the occurrence of the statement in the execution history.

Figure 2.4 shows the DDG for program 2.12, considering the slice on variable z at statement 14, where c_1 and c_2 false on the first iteration and true on the second.

Goswami and Mall [GM02] presented a dynamic algorithm based on the notion of compact dynamic dependence graph (CDDG). The control dependence edges of the CDDG are constructed statically while the data-flow dependence edges are constructed dynamically.

Mund et al. [MMS03] found that CDDG-based approaches may not produce correct result in some cases. They proposed three intraprocedural dynamic slicing methods, two based on marked PDG and another based on their notion of *Unstructured Program Dependence Graph* (UPDG) which can be used for unstructured programs. Their first method also based on the marking and unmarking of edges, while the other two based on the runtime marking and unmarking of nodes. It is claimed that all the three algorithms are precise and more space and time efficient than former algorithms.

Zhang et Gupta [ZG04] found that different dynamic dependence could be expressed by one edge in the dependence graph. They presented a practical dynamic slicing algorithm which is based upon a novel representation of the dynamic dependence graph that is highly compact and rapidly traversable.

Further, Zhang et al. [ZGZ04] studied the statistical characteristics of dynamic slices by experiments. Based on the forward slicing methods, they introduced a way of using reduced ordered binary decision diagrams (roBDDs) to represent a set of dynamic slices. Within this

technique, the space and time requirements of maintaining dynamic slices are greatly reduced. Thus, the efficiency of dynamic slicing can be improved.

2.3.2 Slicing programs with arbitrary control flow

Dependence graph based approach

Korel [Kor97a, Kor97b] used a removable block based approach to handle jump statements in dynamic program slicing. This approach can produce correct slices in the presence of unstructured programs.

Huynh and Song [HS97] then extended the forward dynamic slicing method presented in [KY94] to handle jump statements. However, their algorithm can handle unstructured programs having only structured jumps.

Mund, Mall and Sarkar [MMS03] proposed a notion of jump dependence. Based on this notion, they build the *Unstructured Program Dependence Graph* (UPDG) as the intermediate representation of a program. Their slicing algorithm based on UPDG can produce precise slices.

Faragó and Gergely [FG02] handled jump statements for the forward dynamic slicing by building a transformed D/U structure for all relevant statements. This method can be applied to goto, break, continue and switch statements of C programs.

2.3.3 Interprocedural slicing methods

Dependence graph based approach

Several approaches have been presented concerning on interprocedural dynamic slicing.

In [ADS91], Agrawal et al. consider dynamic slicing of procedures with various parameter-passing mechanisms. *Call-by-value* parameter-passing is modeled by a sequence of assignments $f_1 = a_1; ...; f_n = a_n;$[5] which is executed before the procedure is entered. In order to determine the memory cells for the correct activation record, the USE (see section 1.1) sets for the actual parameters a_i are determined before the procedure in entered, and the DEF sets for the formal parameters f_i after the procedure is entered.

For *Call-by-value-result* parameter passing, additional assignments of formal parameters to actual parameters have to be performed upon exit from the procedure.

Call-by-reference parameter-passing does not require any actions specific to dynamic slicing, as the same memory cell is associated with corresponding actual and formal parameters a_i and f_i.

Notice that in this approach dynamic data dependencies based on definitions and uses of memory location are used. This way, two potential problems are avoided. First, the use of global variables inside procedures does not pose any problems. Second, no alias analysis is required.

Kamkar et al. [KSF92, KFS93b] further discussed the problem of intraprocedural dynamic slicing. They proposed a method that primarily concerned with procedure level slices. That is, they study the problem of determining the set of *call sites* in a program that affect the value of a variable at a particular call site.

[5] It is assumed that a procedure P with formal parameters $f_1, ..., f_n$ is called with actual parameters $a_1, ..., a_n$

During execution, a *dynamic dependence summary graph* is constructed. The vertices of this graph, referred to as *procedure instances*, correspond to procedure activations annotated with their parameters. The edges of the summary graph are either activations edges corresponding to procedure calls, or summary dependence edges. The latter type reflects transitive data and control dependencies between input and output parameters of procedure instances.
A slicing criterion is defined as a pair consisting of a procedure instance, and an input or output parameter of the associated procedure. After constructing the summary graph, a slice with respect to a slicing criterion is determined in two steps. First, the parts of the summary graph from which the criterion can be reached is determined; this subgraph is referred to as an execution slice. Vertices of an execution slice are *partial* procedure instances, because some parameters may be "sliced away". An interprocedural program slice consists of all call sites in the program for which a partial instance occurs in the execution slice.

2.3.4 Slicing in the presence of composite datatypes and pointers

Dataflow equations

Korel and Laski [KL90] consider slicing in the presence of composite variables by regarding each element of an array, or field of a record as a distinct variable. Dynamic data structures are treated as *two* distinct entities, namely the pointer itself and the object being pointed to. For dynamically allocated objects, they propose a solution where a unique name is assigned to each object.

Dependence graph based approach

Agrawal et al. [ADS91] present a dependence graph based algorithm for dyanmic slicing in the presence of composite datatypes and pointers. Their solution consist of expressing DEF and USE sets in terms of *actual memory locations* provided by the compiler. The algorithm presented is similar to that for static slicing in the presence of composite datatypes and pointers by the same authors.
Faragó [FG02] also discussed the problem of handling poniters, arrays and structures for C programs when doing forward dynamic slicing. Abstract memory locations are used in this method and program instrumentation is used to extract these locations.

2.4 Applications of Program Slicing

As discussed in the previous sections, program slicing is a well-recognized technique that is used mainly at source code level to highlight code statements that impact upon other statements. Slicing has many applications because it allows a program to be simplified by focusing attention on a sub-computation of interest for a chosen purpose. In this section we present some of the applications of program slicing.

2.4.1 Debugging

The original motivation for program slicing was to aid the location of faults during debugging activities. The idea was that the slice would contain the fault, but would not contain lines

of code that could not have caused the failure observed. This is achieved by setting the slice criterion to the variable for which an incorrect value is observed.

Clearly slice cannot be used to identify bugs such as missing initialization of a variable. If the original program does not contain a line of code the slice will not contain it either. Although slicing cannot identify omission errors, Harman have argued that slicing can be used to aid the detection of such errors [HBD03].

In debugging, one is often interested in a specific execution of a program that exhibits anomalous behavior. Dynamic slices are particular useful here because they only reflect the actual dependencies of that execution, resulting in smaller slices than static ones. In his thesis, Agrawal discussed how static and dynamic slicing can be used for semi-automated debugging of programs. He proposed an approach where the user gradually 'zooms out' from the location where the bug manifested itself by repeatedly considering larger data and control slices.

Slicing is also useful in algorithmic debugging [FSKG92]. An algorithm debugger partially automates the task of localizing a bug by comparing the intended program behavior with the actual program behavior. The intended behavior is obtained by asking the user whether or not a procedure (program unit) behaves correctly. Using the answers given by the user, the location of the bug can be determined at the unit level.

Debugging was also the motivation for program dicing and latter program chopping (see section 2.1.10). Dicing uses the information that some variables fail some tests, while other variables pass all tests, to automatically identify a set of statements likely to contain the bug [LW87]. The techinque of program chopping identifies the statement that transmit values from a statement t to a statement s. A program chop is useful in debugging when a change at t causes an incorrect result to be produced at s. The statements in $chop(t, s)$ are the statements that transmit the effect of the change at t to s. Debugging attention should be focused here. In the absence of procedures, $chop(t, s)$ is simply the intersection of the forward slice taken with respect to t and the backward slice taken with respect to s can be viewed as a generalized kind of program dice.

2.4.2 Software Maintenance

Software maintainers are faced with the upkeep of programs after their initial release and experiment the same problems as program integrators: understanding existing software and making changes without having a negative impact on the unchanged part.

Gallagher and Lyle [GL91b] use the notion of decomposition slice of programs. A decomposition slice with respect to a variable v consists of all statements that may affect the observable value of v at some point; it is defined as the union of slices with respect to v at any statement that outputs v, and the last statement of the program. Essentially, they decompose a program into a set of components (reduced programs), and each of them captures part of the original program's behavior. The main observation of [GL91b] is that independent statements in a slice do not affect the data and control flow in the complement. This results in the follow guidelines for modification:

- Independent statements may be deleted from a decomposition slice;
- Assignments to independent variables may be added anywhere in a decomposition slice;
- Logical expressions and output statements may be added anywhere in a decomposition

slice;

- New control statements that surround any dependent statements will affect the complement's behavior.

Slicing can also be used to identify reusable functions [CLLF94, CCLL94, CLM95, CLM96, LV97]. Canfora et al. presented a method to identify functional abstraction in existing code [CLLF94]. In this approach, program slicing is used to isolate the external functions of a system and these are then decomposed into more elementary components by intersection slices. They also found that conditioned slice could be used to extract procedures from program functionality.

2.4.3 Reverse engineering

Besides above application in software maintenance, program slicing can be used in reverse engineering [BE93, JR94]. Reverse engineering concerns the problem of comprehending the current design of a program and the way this design differs from the initial development. This involves abstracting out of the source code, the design decisions and rationale from the initial development (design recognition) and understanding algorithms chosen (algorithm recognition).

Beck and Eichmann [BE93] applied program slicing techniques to reverse engineering by using it to assist in the comprehension of large software systems, through traditional slicing techniques at the statement level, and through a new technique, *interface slicing*, at the module level. A dependence model for reverse engineering should treat procedures in a modular fashion and should be fine-grained, distinguishing dependencies that are due to different variables. Jackson and Rollins [JR94] proposed an improved PDG that satisfies both, while retaining the advantages of PDG. They proposed an algorithm to compute chopping from their dependence graph which can produce more accurate results than algorithms based directly on the PDG.

2.4.4 Comprehension

Program comprehension is a vital software engineering and maintenance activity. It is necessary to facilitate reuse, inspection, maintenance, reverse engineering, reengineering, migration, and extension of existing software systems.

Slicing revealed to be helpful in the comprehension phase of maintenance. De Lucia et al. [LFM96] used conditioned slicing to facilitate program comprehension. Quasi-static slicing can also be used in program comprehension. These techniques share the property that a slice is constructed with respect to a condition in addition to the traditional static slicing and thus can give the maintainer the possibility to analyze code fragments with respect to different perspectives.

Indeed, slicing can be used in many aspects of program comprehension: Harman, Sivagurunathan and Daninic [HSD98, SHS02] used program slicing in understanding dynamic memory access properties. Komondoor and Horwitz [KH01] presented an approach that use PDG and slicing to find duplicate code fragments in C programs. Henrard et al. [HEH+98] made use of program slicing in database understanding. Korel and Rilling used dynamic slicing to help understand the program execution [KR97].

2.4.5 Testing

Software maintainers are also faced with the task of regression testing: retesting software after a modification. This process may involve running the modified program on a large number of test cases, even after the smallest of changes. Although the effort required to make a small change may be minimal, the effort required to retest a program after such a change may be substantial. Several algorithms based on program slicing have been proposed to reduce the cost of regression testing.

A program satisfies a conventional *data flow testing* criterion if all def-use pairs occur in a successful test.

Duesterwald, Gupta and Soffa [DGS92] propose a more rigorous testing criterion, based on program slicing: each def-use pair must be exercised in a successful test; moreover it must be *output influencing*, i.e., have an influence on at least one output value. A def-use pair is output-influencing if it occurs in an *output slice* (a slice with respect to an output statement). It is up to user, or an automatic test generator to construct enough tests such that all def-use pairs are tested.

Kamkar, Shahmerhi and Fritzon [KFS93a] extended the previous work to multi-procedure programs. To this end, they define appropriate notions of interprocedural def-use pairs. The interprocedural dynamic slicing method of [KFS93b, KSF92] is used to determine which interprocedural def-use pairs have an effect on a correct output value, for a given test. The summary-graph presentation is slightly modified by annotating vertices and edges with def-use information. This way, the set of def-use pairs exercised by a slice can be determined efficiently.

In [GHS92], Gupta, Harrold and Soffa describe an approach to regression testing where slicing techniques are used. Backward and forward slices serve to determine the program parts affected by the change, and only tests which execute affected def-use pairs need to be executed again. Conceptually, slices are computed by backward and forward traversals of the CFG of a program, starting at the point of modification.

In [BH93b], Bates and Horwitz used a variation of the PDG notion of [HPR89b] for incremental program testing. Bates and Horwitz presented test selection algorithms for all the vertices and flow-edges test data adequacy criterion. They proved that statements in the same class are exercised by the same tests. This work only considers single procedure programs.

Binkley [Bin97] presented two complementary algorithms for reducing the cost of regression testing that operate on programs with procedures and procedure calls.

2.4.6 Measurement

Cohesion and **coupling** are two important metrics in software measurement.

Cohesion is an attribute of a software unit that measures the "relatedness" of the unit. It has been qualitatively characterized as coincidental, logical, procedural, communicational, sequential and functional; coincidental is the weakest and functional is the strongest.

Several approaches using program slicing to measure cohesion have been presented.

It was Longworth [Lon85] the first to study the use of program slicing as indicator of cohesion.

Ott and Thuss [OT89] then noted the visual relationship that existes between the slices of a module and its cohesion as depicted in a slice profile. Certain inconsistencies noted by

Longworth were eliminated through the use of metric slices [OB92, Ott92, OT93, Thu88]. A metric slices takes into account both uses and used by data relationships; that is, they are the union of Horwitz et al.'s backward and forward slices.

Bieman and Ott [BO93] examined the functional cohesion of procedures using a data slice abstraction. A data slice is a backward and forward static slice that uses data tokens rather than statements as the unit of decomposition. Their approach identifies the data tokens that lie on more than one slice as the "glue" that bind separate components together. Cohesion is measured in terms of the relative number of glue tokens, tokens that lie on more that one data slice, and super-glue tokens, tokens that lie on all data slices in a procedure, and the adhesiveness of the tokens.

Coupling is the measure of how one module depends upon or affects the behavior of another. Harman et al [HOSD] proposed a method of using program slicing to measure coupling. It is claimed that this method produce more precise measurement than information flow based metrics.

2.5 Tools using Program Slicing

In this section we present some tools that uses program slicing to aid at program understanding and comprehension.

2.5.1 CodeSurfer

As referred in subsection 1.5.3, CodeSurfer [Gra08b] is a static analysis tool designed to support advanced program understanding based on the dependence-graph representation of a program. CodeSurfer is thus named because it allows surfing of programs akin to surfing the world-wide web [AT01].

CodeSurfer builds an intermediate representation called the system dependence graph (SDG—see section 1.1). The program slices are computed using graph reachability over the SDG. CodeSurfer's output goes through two preprocessing steps before slicing begins [BGH07].

The first identifies intraprocedural strongly connected components (SCCs) and replaces them with a single representative vertex. The key observation here is that any slice that includes a vertex from an SCC will include all the vertices from that SCC; thus, there is a great potential for saving effort by avoiding redundant work [BH03]. Once discovered, SCC formation is done by moving all edges of represented vertices to the representative. The edgeless vertices are retained to maintain the mapping back to the source. While slicing, the slicer need never encounter them.

The second preprocessing step reorders the vertices of each procedure into topological order. This is possible because cycles have been removed by the SCC formation. Topological sorting improves memory performance — in particular, cache performance [BH03]. After preprocessing, two kinds of slices are computed: backward and forward interprocedural slicing.

Operations that highlight forward and backward slices show the impact of a given statement on the rest of the program (forward slicing), and the impact of the rest of a program on a given statement (backward slicing). Operations that highlight paths between nodes in the dependence graph (chops) show ways in which the program points are interdependent (or independent).

2.5.2 JSlice

JSlice [WR08, WRG] was the first dynamic slicing tool for Java programs. This slicer proceeds by traversing a compact representation of a bytecode trace and constructs the slice as a set of bytecodes; this slice is then transformed to the source code level with the help of Java class files. This slicing method is complicated by Java's stack-based architecture which require to simulate a stack during trace traversal.

Since the execution traces are often huge, the authors of the tool develop a space efficient representation of the bytecode stream for a Java program execution. This compressed trace is constructed on-the-fly during program execution. The dynamic slicer performs backward traversal of this compressed trace directly to retrieve data/control dependencies, that is, slicing does not involve costly trace decompression.

2.5.3 Unravel

Unravel [LWG+95] is a static program slicer developed at the National Institute of Standards and Technology as part of a research project. It slices ANSI-C programs. The limitations of Unravel are in the treatment of unions, forks, and pointers to functions. The tool is divided into three main components:

- a source code analysis component to collect information necessary for the computation of program slices;
- a link component to connect information from separate source files together;
- and an interactive slicing component: to extract program components that the software quality assurance auditor can use; and to extract program statements for answering questions about the software being audited.

By combining program slices with logical set operations, Unravel can identify code that is executed in more than one computation.

2.5.4 HaSlicer

HaSlicer [Rod06, RB06] is a prototype of a slicer for functional programs written in Haskell. The tool was built for the identification of possible coherent components from monolithic code. It covers both backward and forward slicing using a *Functional Dependence Graph* (FDG), an extension to functional languages of PDG.

In [RB06] the authors discuss how the tool can be used to component identification through the extraction process from source code and the incorporation of a visual interface over the generated FDG to support user interaction.

2.5.5 Other tools

There are other tools that use program slicing: CodeGenie [LBO+07, Lop08] is a tool that implements a test-driven approach to search for code available on large-scale code repositories in order to reuse the fragments found; and GDB-Slice [Bes] which implements a novel efficient algorithm to compute slices [GABF99] in GDB (GNU Project debugger) through the GCC (GNU C Compiler Collection).

Bibliography

[10405] Csmr '05: Proceedings of the ninth european conference on software maintenance and reengineering, 2005.

[11005] Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005. Member-Atif M. Memon and Student Member-Qing Xie.

[12406] Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, 2006. Member-Chao Liu and Member-Long Fei and Member-Xifeng Yan and Senior Member-Jiawei Han and Member-Samuel P. Midkiff.

[84801] Aiding program comprehension by static and dynamic feature analysis. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 602, Washington, DC, USA, 2001. IEEE Computer Society.

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Symposium on Testing, Analysis, and Verification*, pages 60–73, 1991.

[ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 302–312, New York, NY, USA, 1994. ACM.

[AH90] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, June 1990.

[AT01] Paul Anderson and Tim Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering*, 2001.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[Bal93] Thomas J. Ball. The use of control-flow and control dependence in software tools. Technical Report CS-TR-1993-1169, 1993.

[Bal02] FranCcoise Balmas. Using dependence graphs as a support to document programs. In *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, page 145, Washington, DC, USA, 2002. IEEE Computer Society.

[Ban88] Utpal K. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, 21(9):724–736, 1978.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM.

[BDG+04] Dave Binkley, Sebastian Danicic, Tibor Gyimothy, Mark Harman, Akos Kiss, and Lahcen Ouarbya. Formalizing executable dynamic and forward slicing. *scam*, 00:43–52, 2004.

[BDG+06] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic program slicing. *Theor. Comput. Sci.*, 360(1):23–41, 2006.

[BE93] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[Bes] Arpad Beszedes. Gnu gdb slice. http://www.sed.hu/gdbslice/.

[BFS+02] A. Beszedes, C. Farago, Z. Szabo, J. Csirik, and T. Gyimothy. Union slices for program maintenance, 2002.

[BG96] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[BG02] A. Beszedes and T. Gyimothy. Union slices for the approximation of the precise slice, 2002.

[BGH07] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2):8, 2007.

[BGL93] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. *SIGPLAN Not.*, 28(10):65–82, 1993.

[BH93a] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging*, pages 206–222, 1993.

[BH93b] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, New York, NY, USA, 1993. ACM.

[BH03] David Binkley and Mark Harman. Results from a large scale study of performance optimization techniques for source code analyses based on graph reachability algorithms, 2003.

[BHR95] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[Bin93] David Binkley. Precise executable interprocedural slices. *LOPLAS*, 2(1-4):31–45, 1993.

[Bin97] David Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, 1997.

[Bin98] David Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11-12):583–594, 1998.

[Bin07] David Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[BJE88] D. Bjorner, Neil D. Jones, and A. P. Ershov, editors. *Partial Evaluation and Mixed Computation.* Elsevier Science Inc., New York, NY, USA, 1988.

[BO93] James M. Bieman and Linda M. Ott. Measuring functional cohesion. Technical Report CS-93-109, Fort Collins, CO, USA, 24 June 1993.

[BR00] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1993. ACM.

[CC93] T. Y. Chen and Y. Y Cheung. Dynamic program dicing. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 378–385, Washington, DC, USA, 1993. IEEE Computer Society.

[CCL98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998. Special issue on program slicing.

[CCLL94] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. Software salvaging based on conditions. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 424–433, Washington, DC, USA, 1994. IEEE Computer Society.

[CCO01] Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. Flaver: A finite state verification technique for software systems title2:. Technical report, Amherst, MA, USA, 2001.

[CF94] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, 1994.

[CFR+99] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[CLLF94] Gerardo Canfora, Andrea Di Luccia, Giuseppe Di Lucca, and A. R. Fasolino. Slicing large programs to isolate reusable functions. In *Proceedings of EUROMICRO Conference*, pages 140–147. IEEE CS Press, 1994.

[CLM95] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. Identifying reusable functions using specification driven program slicing: a case study. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 124, Washington, DC, USA, 1995. IEEE Computer Society.

[CLM96] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance*, 8(3):145–178, 1996.

[CP07] Gerardo CanforaHarman and Massimiliano Di Penta. New frontiers of reverse engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.

[CS00] Chen and Skiena. A case study in genome-level fragment assembly. *BIOINF: Bioinformatics*, 16, 2000.

[Dar86] Ian F. Darwin. *Checking C programs with lint*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1986.

[DGS92] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Rigorous data flow testing through output influences. In *Proceeding on Second Irvine Software Symposium*, pages 131–145, 1992.

[DHR+07] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *FOSE '07: 2007 Future of Software Engineering*, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society.

[DLH04] Sebastian Danicic, Andrea De Lucia, and Mark Harman. Building executable union slices using conditioned slicing. *iwpc*, 00:89, 2004.

[FdCHV08] Rúben Fonseca, Daniela da Cruz, Pedro Henriques, and Maria João Varanda. How to interconnect operational and behavioral views of web applications. In *ICPC '08: IEEE International Conference on Program Comprehension*, Amsterdam, The Netherlands, 2008. IEEE Computer Society.

[FG] Istvan Forgács and Tibor Gyimóthy. An efficient interprocedural slicing method for large programs.

[FG02] Csaba Faragó and Tamás Gergely. Handling pointers and unstructured statements in the forward computed dynamic slice algorithm. *Acta Cybern.*, 15(4):489–508, 2002.

[FGKS91] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri. Generalized algorithmic debugging and testing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 317–326, New York, NY, USA, 1991. ACM.

[FHK+02] Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kerny Wong. The software bookshelf. pages 295–339, 2002.

[FN87] William B. Frakes and Brian A. Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a compiler*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1988.

[FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–392, New York, NY, USA, 1995. ACM.

[FSKG92] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, 1992.

[GABF99] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 303–321, London, UK, 1999. Springer-Verlag.

[Gal90] Keith Brian Gallagher. *Using program slicing in software maintenance*. PhD thesis, Catonsville, MD, USA, 1990.

[GFR06] João Gama, Ricardo Fernandes, and Ricardo Rocha. Decision trees for mining data streams. *Intell. Data Anal.*, 10(1):23–45, 2006.

[GHS92] R. Gupta, M. Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 299–308, 1992.

[GL91a] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[GL91b] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[GM02] D. Goswami and R. Mall. An efficient method for computing dynamic program slices. *Inf. Process. Lett.*, 81(2):111–117, 2002.

[GMMS07] Ulrich Güntzer, Rudolf Müller, Stefan Müller, and Ralf-Dieter Schimkat. Retrieval for decision support resources by structured models. *Decis. Support Syst.*, 43(4):1117–1132, 2007.

[Gop91] R. Gopal. Dynamic program slicing based on dependence relations. In *Proceedings of the Software Maintenance'91 Conference*, pages 191–200, Sorrento, Italy, October 1991.

[Gra08a] GramaTech. A code-analysis tool that identifies complex bugs at compile time. http://www.grammatech.com/products/codesonar/, 2008.

[Gra08b] GramaTech. A code browser that understands pointers, indirect function calls, and whole-program effects. http://www.grammatech.com/products/codesurfer/, 2008.

[GSH97] Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397, 1997.

[Hal95] R.J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2:33–53, 1995. An algorithm to automatically extract a correctly functioning subset of the code of a system is presented. The technique is based on computing a simultaneous dynamic program slice of the code for a set of representative inputs. Experiments show that the algorithm produces significantly smaller subsets than with existing methods.

[Har00] Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM.

[HBD03] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *J. Syst. Softw.*, 68(1):45–64, 2003.

[HBP02] Dixie Hisley, Matthew J. Bridges, and Lori L. Pollock. Static interprocedural slicing of shared memory parallel programs. In *PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 658–664. CSREA Press, 2002.

[HD95] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification & Reliability*, 5(3):143–162, 1995.

[HD98] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):415–441, 1998.

[HEH+98] Jean Henrard, Vincent Englebert, Jean-Marc Hick, Didier Roland, and Jean-Luc Hainaut. Program understanding in databases reverse engineering. In *Database and Expert Systems Applications*, pages 70–79, 1998.

[HHF+01] Mark Harman, Rob Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. *icsm*, 00:138, 2001.

[HHHL03] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

[Hoe05] Urs Hoelzle. Google: or how i learned to love terabytes. *SIGMETRICS Perform. Eval. Rev.*, 33(1):1–1, 2005.

[HOSD] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of function coupling.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, San Diego, California, 1988.

[HPR89a] Susan Horwitz, P. Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 28–40, New York, NY, USA, 1989. ACM.

[HPR89b] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[HS97] D. Huynh and Y. Song. Forward computation of dynamic slicing in the presence of structured jump statements. In *Proceedings of ISACC*, pages 73–81, 1997.

[HSD98] Mark Harman, Yoga Sivagurunathan, and Sebastian Danicic. Analysis of dynamic memory access using amorphous slicing. In *ICSM*, pages 336–, 1998.

[HSS01] Uri Hanani, Bracha Shapira, and Peretz Shoval. Information filtering: Overview of issues, research and systems. *User Modeling and User-Adapted Interaction*, 11(3):203–259, 2001.

[ICG07] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A perspective on the future of middleware-based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[Jaz07] Mehdi Jazayeri. Some trends in web application development. In *FOSE '07: 2007 Future of Software Engineering*, pages 199–213, Washington, DC, USA, 2007. IEEE Computer Society.

[JDC88] Hwang J.C., M.W. Du, and C.R. Chou. Finding program slices for recursive procedures. In *Proceedings of IEEE COMPSAC 88*, Washington, DC, 1988. IEEE Computer Society.

[Joh78] Stephen Johnson. Lint, a c program checker, 1978.

[JR94] Daniel Jackson and Eugene J. Rollins. A new model of program dependence for reverse engineering. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 2–10, New York, NY, USA, 1994. ACM.

[JR00] Daniel Jackson and Martin Rinard. Software analysis: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, New York, NY, USA, 2000. ACM.

[JZR91] J. Jiang, X. Zhou, , and D.J. Robson. Program slicing for c - the problems in implementation. In *Proceedings of Conference on Software Maintenance*, pages 182–190. IEEE CSPress, 1991.

[KFS93a] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceeding on Conference on Software Maintenance*, pages 386–395, 1993.

[KFS93b] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Three approaches to interprocedural dynamic slicing. *Microprocess. Microprogram.*, 38(1-5):625–636, 1993.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–??, 2001.

[KH02] Sumit Kumar and Susan Horwitz. Better slicing of programs with jumps and switches. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 96–112, London, UK, 2002. Springer-Verlag.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KJLG03] Akos Kiss, Judit Jasz, Gabor Lehotai, and Tibor Gyimothy. Interprocedural static slicing of binary executables. *scam*, 00:118, 2003.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[KL90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[Kor97a] Bogdan Korel. Computation of dynamic program slices for unstructured programs. *IEEE Trans. Softw. Eng.*, 23(1):17–34, 1997.

[Kor97b] Bogdan Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, January 1997.

[KR97] Bogdan Korel and Jurgen Rilling. Dynamic program slicing in understanding of program execution. *wpc*, 0:80, 1997.

[KRML04] Nick Kidd, Thomas Reps, David Melski, and Akash Lal. Wpds++: A c++ library for weighted pushdown systems, 2004.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[KS06] Sarfraz Khurshid and Yuk Lai Suen. Generalizing symbolic execution to library classes. *SIGSOFT Softw. Eng. Notes*, 31(1):103–110, 2006.

[KSF92] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384, London, UK, 1992. Springer-Verlag.

[KT04] Yiannis Kanellopoulos and Christos Tjortjis. Data mining source code to facilitate program comprehension: Experiments on clustering data retrieved from c++ programs. *iwpc*, 00:214, 2004.

[KY94] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 66–79, New York, NY, USA, 1994. ACM.

[Lak92] Arun Lakhotia. Improved interprocedural slicing algorithm, 1992.

[Lak93] Arun Lakhotia. Rule-based approach to computing module cohesion. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 35–44, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[LBO+07] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, New York, NY, USA, 2007. ACM.

[LC94] Panos E. Livadas and Stephen Croll. System dependence graphs based on parse trees and their use in software maintenance. *Information Sciences*, 76(3-4):197–232, 1994.

[LFB06] Dawn J. Lawrie, Henry Feild, and David Binkley. Leveraged quality assessment using information retrieval techniques. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 149–158, Washington, DC, USA, 2006. IEEE Computer Society.

[LFM96] Andrea De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of the 4th Workshop on Program Comprehension*, pages 9–18, 1996.

[LHHK03] Andrea De Lucia, Mark Harman, Robert Hierons, and Jens Krinke. Unions of slices are not slices. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR 2003)*, 2003.

[LLWY03] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. Towards a clone detection benchmark suite and results archive. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 285, Washington, DC, USA, 2003. IEEE Computer Society.

[LMFB06] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[Lon85] H. D. Longworth. Slice-based program metrics. Master's thesis, 1985.

[Lop08] Crista Lopes. Codegenie. http://sourcerer.ics.uci.edu/codegenie/, 2008.

[LR87] Hareton K. N. Leung and Hassan K. Reghbati. Comments on program slicing. *IEEE Trans. Softw. Eng.*, 13(12):1370–1371, 1987.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27(7):235–248, 1992.

[LSL96] Hongjun Lu, Rudy Setiono, and Huan Liu. Effective data mining using neural networks. *IEEE Trans. on Knowl. and Data Eng.*, 8(6):957–961, 1996.

[Luc01] Andrea De Lucia. Program slicing: Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, Novembro 2001.

[LV97] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, April 1997.

[LW86] Jim Lyle and Mark Weiser. Experiments on slicing-based debugging tools. In *Proceedings of the 1st Conference on Empirical Studies of Programming*, pages 187–197, Norwood, New Jersey, 1986. Ablex publishing.

[LW87] Jim Lyle and Mark Weiser. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–883, 1987.

[LWG+95] Jim Lyle, D. Wallace, J. Graham, Keith Gallagher, J. Poole, and David Binkley. Unravel: A case tool to assist evaluation of high integrity software, 1995.

[Lyl84] James Robert Lyle. *Evaluating variations on program slicing for debugging (dataflow, ada)*. PhD thesis, College Park, MD, USA, 1984.

[M.93] Kamkar M. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD thesis, Linkoping University, Sweden, 1993.

[Mar03] Andrian Marcus. *Semantic-driven program analysis*. PhD thesis, Kent, OH, USA, 2003. Director-Jonathan I. Maletic.

[MC88] B. P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. *SIGPLAN Not.*, 23(7):135–144, 1988.

[Mic08a] Microsoft. Design guidelines for class library developers. http://msdn.microsoft.com/en-us/library/czefa0ke(VS.71).aspx, 2008.

[Mic08b] Microsoft. Fxcop. http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx, 2008.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, 2005.

[MM01] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 107, Washington, DC, USA, 2001. IEEE Computer Society.

[MMCG99] Spiros Mancoridis, Brian S. Mitchell, Y. Chen, and Emden R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 50, Washington, DC, USA, 1999. IEEE Computer Society.

[MMS03] G. B. Mund, Rajib Mall, and S. Sarkar. Computation of intraprocedural dynamic program slices. *Information & Software Technology*, 45(8):499–512, 2003.

[MSRM04] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[MTO+92] Hausi A. Müller, Scott R. Tilley, Mehmet A. Orgun, B. D. Corrie, and Nazim H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments,* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88–98, December 1992.

[OB92] Linda Ottenstein and James Bieman. Effects of software changes on module cohesion. In *International Conference on Software Maintenance*, pages 345–353, 1992.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes*, 9(3):177–184, 1984.

[OSH01] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *ICSM*, pages 158–, 2001.

[OT89] Linda Ottenstein and Jeffrey J. Thuss. The relationship between slices and module cohesion. In *International Conference on Software Engineering*, pages 198–204, 1989.

[OT93] Linda Ottenstein and J. Thuss. Slice based metrics for estimating cohesion, 1993.

[Ott92] Linda Ottenstein. Using slice profiles and metrics during software maintenance, 1992.

[PM04] Sankar K. Pal and Pabitra Mitra. *Pattern Recognition Algorithms for Data Mining: Scalability, Knowledge Discovery, and Soft Granular Computing.* Chapman & Hall, Ltd., London, UK, UK, 2004.

[PV06] Sokhom Pheng and Clark Verbrugge. Dynamic data structure analysis for java programs. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 191–201, Washington, DC, USA, 2006. IEEE Computer Society.

[QH04] Feng Qian and Laurie Hendren. Towards dynamic interprocedural analysis in jvms. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 11–11, Berkeley, CA, USA, 2004. USENIX Association.

[RB06] Nuno Rodrigues and Luis Soares Barbosa. Component identification through program slicing. In L. S. Barbosa and Z. Liu, editors, *Proc. of FACS'05 (2nd Int. Workshop on Formal Approaches to Component Software)*, volume 160, pages 291–304, UNU-IIST, Macau, 2006. Elect. Notes in Theor. Comp. Sci., Elsevier.

[RBF96] Ph.D. Ronald B. Finkbine. Metrics and models in software quality engineering. *SIGSOFT Softw. Eng. Notes*, 21(1):89, 1996.

[RBL06] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, New York, NY, USA, 2006. ACM.

[RD06] Daniel Ratiu and Florian Deissenboeck. How programs represent reality (and how they don't). In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 83–92, Washington, DC, USA, 2006. IEEE Computer Society.

[Rep96] Thomas W. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33(8):739–757, 1996.

[RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[Rod06] Nuno Rodrigues. Haslicer. http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx, 2006.

[RSJM05] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[Rug95] S. Rugaber. Program comprehension, 1995.

[Sar03] Kamran Sartipi. Software architecture recovery based on pattern matching. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 293, Washington, DC, USA, 2003. IEEE Computer Society.

[Sch02] David A. Schmidt. Structure-preserving binary relations for program abstraction. pages 245–265, 2002.

[SHR99] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *International Conference on Software Engineering*, pages 432–441, 1999.

[SHS02] Yoga Sivagurunathan, Mark Harman, and Bala Sivagurunathan. Slice-based dynamic memory modelling – a case study, 2002.

[Sil06] Josep Silva. A comparative study of algorithmic debugging strategies. In *LOPSTR*, pages 143–159, 2006.

[SKL06] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-language program analysis and refactoring. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 207–216, Washington, DC, USA, 2006. IEEE Computer Society.

[SMS01] Timothhy S. Souder, Spiros Mancoridis, and Maher Salah. Form: A framework for creating views of program executions. In *ICSM*, pages 612–, 2001.

[SR03] Eric J. Stierna and Neil C. Rowe. Applying information-retrieval methods to software reuse: a case study. *Inf. Process. Manage.*, 39(1):67–74, 2003.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[Thu88] J.J. Thuss. An investigation into slice based cohesion metrics. Master's thesis, 1988.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[TWSM94] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119, New York, NY, USA, 1991. ACM.

[Wal91] David W. Wall. Systems for late code modification. In *Code Generation*, pages 275–293, 1991.

[WC96] Norman Wilde and Christopher Casey. Early field experience with the software reconnaissance technique for program comprehension. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 312–318, Washington, DC, USA, 1996. IEEE Computer Society.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–310, New York, NY, USA, 1994. ACM.

[Wei79] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, Ann Arbor, MI, USA, 1979.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[WFP07] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[WL86] Mark Weiser and Jim Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[WR08] Tao Wang and Abhik Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.*, 30(2):1–49, 2008.

[WRG] Tao Wang, Abhik Roychoudhury, and Liang Guo. Jslice. http://jslice.sourceforge.net/.

[ZB04] Jihong Zeng and Peter A. Bloniarz. From keywords to links: an automatic approach. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, page 283, Washington, DC, USA, 2004. IEEE Computer Society.

[Zel01] Andreas Zeller. Automated debugging: Are we close. *Computer*, 34(11):26–31, 2001.

[Zel07] Andreas Zeller. The future of programming environments: Integration, synergy, and assistance. In *FOSE '07: 2007 Future of Software Engineering*, pages 316–325, Washington, DC, USA, 2007. IEEE Computer Society.

[ZG04] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing, 2004.

[ZGZ04] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 502–511, Washington, DC, USA, 2004. IEEE Computer Society.