Advanced Slicing of Sequential and Concurrent Programs

Jens Krinke

April 2003

Dissertation zur Erlangung des Doktorgrades in den Naturwissenschaften

Acknowledgments

First of all, I wish to thank my adviser Gregor Snelting for providing the support, freedom and protection to do research without pressure. A big 'sorry' goes to him, to the second reviewer, Tim Teitelbaum, and to David Melski, because this thesis has become longer than expected—thanks for reading it and the valuable comments.

Without the love and support of Uta von Holten, this work would not have been possible. She always believed in me, although this thesis took too long to finish. I thank my parents for enabling my career.

A big 'thank you' goes to my students who helped a lot by implementing parts of the presented techniques. First of all the former students at TU Braunschweig: Frank Ehrich, who implemented the user interface of the VALSOFT system (14.1.6) including the graphical visualization of program dependence graphs, slices and chops (9.1), Christian Bruns implemented the constant propagation (11.2.2), Torsten Königshagen implemented the call graph construction (11.2.1), Stefan Konst implemented the approximate dynamic slicing (14.1.8), and Carsten Schulz implemented the common subexpression elimination (11.2.3). Students from Universität Passau were: Silvia Breu implemented the chop visualization (10.4), Daniel Gmach implemented the textual visualization of slices (9.2) and duplicated code (12.2.2), Alexander Wrobel implemented the distance-limited slicing (10.1), and Thomas Zimmermann implemented some chopping algorithms (10.2).

Special thanks go to my colleagues in Passau, Torsten Robschink, Mirko Streckenbach, and Maximilian Störzer, who took over some of my teaching and administration duties while I finished this thesis. I also have to thank my former colleagues from Braunschweig, Bernd Fischer and Andreas Zeller for inspiration; they had the luck to finish their thesis earlier.

The Programming group at Universität Passau kindly provided the computing power that was needed to do the evaluations. Silvia Breu helped to improve the English of this thesis.

GrammaTech kindly provided the CodeSurfer slicing tool and Darren Atkinson provided the Icaria slicer.

The VALSOFT project started as a cooperation with the Physikalisch-Technische Bundesanstalt (PTB) and LINEAS GmbH in Braunschweig, funded by the former Bundesministerium für Bildung und Forschung (FKZ 01 IS 513

C9). Later on, funding was provided by the Deutsche Forschungsgemeinschaft (FKZ Sn11/5-1 and Sn11/5-2).

Abstract

Program slicing is a technique to identify statements that may influence the computations in other statements. Despite the ongoing research of almost 25 years, program slicing still has problems that prevent a widespread use: Sometimes, slices are too big to understand and too expensive and complicated to be computed for real-life programs. This thesis presents solutions to these problems: It contains various approaches which help the user to understand a slice more easily by making it more focused on the user's problem. All of these approaches have been implemented in the VALSOFT system and thorough evaluations of the proposed algorithms are presented.

The underlying data structures used for slicing are program dependence graphs. They can also be used for different purposes: A new approach to clone detection based on identifying similar subgraphs in program dependence graphs is presented; it is able to detect modified clones better than other tools.

In the theoretical part, this thesis presents a high-precision approach to slice concurrent procedural programs despite that optimal slicing is known to be undecidable. It is the first approach to slice concurrent programs that does not rely on inlining of called procedures.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

> *Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.*
>
> Mark Weiser [Wei84]

Program slicing answers the question "Which statements may affect the computation at a different statement?", something every programmer asks once in a while. After Weiser's first publication on slicing in 1979, almost 25 years have passed and various approaches to compute slices have evolved. Usually, inventions in computer science are adopted widely after around 10 years. Why are slicing techniques not easily available yet? William Griswold gave a talk at PASTE 2001 [Gri01] on that topic: *Making Slicing Practical: The Final Mile*. He pointed out why slicing is still not widely used today. The two main problems are:

1. Available slicers are slow and imprecise.
2. Slicing 'as-it-stands' is inadequate to essential software-engineering needs.

Not everybody agrees with his opinion. However, his first argument is based on the observation that research has generated fast and precise approaches but scaling the algorithms for real-world programs with million lines of code is still an issue. Precision of slicers for sequential imperative languages has reached a high level, but it is still a challenge for the analysis of concurrent programs—only lately is slicing done for languages with explicit concurrency like in Ada

or Java. The second argument is still valid: Usually, slices are hard to understand. This is partly due to bad user interfaces, but is mainly related to the problem that slicing 'dumps' the results onto the user without any explanation.

This thesis will try to show how these problems and challenges can be tackled. Therefore, the three main topics are:

1. Present ways to slice concurrent programs more precisely.

2. Help the user to understand a slice more easily by making it more focused on the user's problem.

3. Give indications of the problems and consequences of slicing algorithms for future developers.

Furthermore, this thesis gives a self-contained introduction to program slicing. It does not try to give a complete survey because since Tip's excellent survey [Tip95][1] the literature relevant to slicing has exploded: CiteSeer [LGB99] recently reported 257 citations of Weiser's slicing article [Wei84] (and 95 for [Wei82]). This thesis only contains 187 references where at least 108 have been published after Tip's survey.

1.1 Slicing

A slice extracts those statements from a program that potentially have an influence on a specific statement of interest, which is the slicing criterion. Originally, slicing was defined by Weiser in 1979; he presented an approach to compute slices based on iterative data flow analysis [Wei79, Wei84]. The other main approach to slicing uses reachability analysis in program dependence graphs [FOW87]. Program dependence graphs mainly consist of nodes representing the statements of a program and control and data dependence edges:

- Control dependence between two statement nodes exists if one statement controls the execution of the other.

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

An example PDG is shown in figure 1.1 on the facing page, where control dependence is drawn in dashed lines and data dependence in solid ones. How control and data dependence is computed will be presented in chapter 2.

[1] Tip's survey [Tip95] has been followed by some others [BG96, HG98, DL01, HH01].

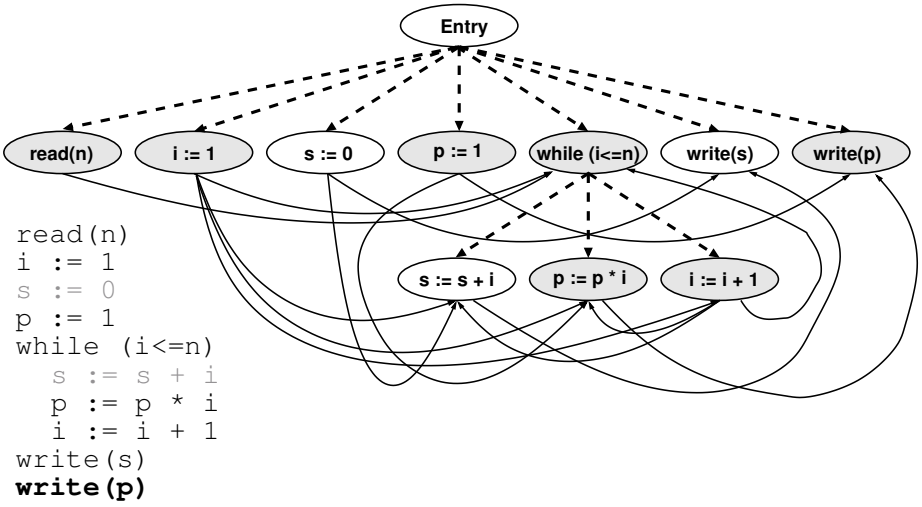


Figure 1.1: A program dependence graph

1.1.1 Slicing Sequential Programs

Example 1.1 (Slicing without Procedures): Figure 1.2 shows a first example where a program without procedures shall be sliced. To compute the slice for the statement `print a`, we just have to follow the shown dependences backwards. This example contains two data dependences and the slice includes the assignment to `a` and the read statement for `b`.

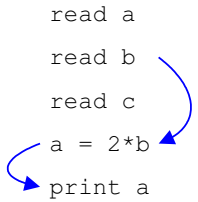


Figure 1.2: A procedure-less program

In all examples of this introduction, we will ignore control dependence and just focus on data dependence for simplicity of presentation. Also, we will always slice backwards from the `print a` statement.

Slicing without procedures is trivial: Just find reachable nodes in the PDG [FOW87]. The underlying assumption is that all paths are *realizable*. This means that a possible execution of the program exists for any path that executes the statements in the same order. Chapter 3 will discuss this in detail.

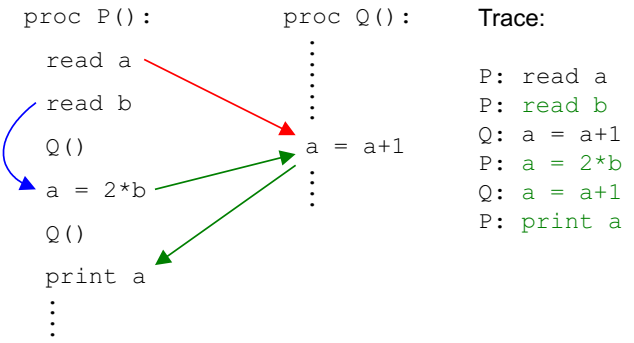


Figure 1.3: A program with two procedures

Example 1.2 (Slicing with Procedures): Now, the example is extended by adding procedures in figure 1.3. If we ignore the calling context and just do a traversal of the data dependences, we would add the `read a` statement into the slice for `print a`. This is wrong because this statement has clearly no influence on the `print a` statement. The `read a` statement just has an influence on the first call of procedure `Q` but `a` is redefined before the second call to procedure `Q` through the assignment `a=2*b` in procedure `P`.

Such an analysis is called *context-insensitive* because the calling context is ignored. Paths are now considered realizable only if they obey the calling context. Thus, slicing is *context-sensitive* if only realizable paths are traversed. Context-sensitive slicing is solvable efficiently—one has to generate summary edges at call sites [HRB90]: Summary edges represent the transitive dependences of called procedures at call sites. How procedural programs are analyzed will be discussed in chapter 6.

Within the implemented infrastructure to compute PDGs for ANSI C programs, various slicing algorithms have been implemented and evaluated. One of the evaluations in chapter 7 will show that context-insensitive slicing is very imprecise in comparison with context-sensitive slicing. This shows that context-sensitive slicing is highly preferable because the loss of precision is not acceptable. A surprising result is that the simple context-insensitive slicing is *slower* than the more complex context-sensitive slicing. The reason is that the context-sensitive algorithm has to visit many fewer nodes during traversal due to its higher precision.

1.1.2 Slicing Concurrent Programs

Now, let's move on to concurrent programs. In concurrent programs that share variables another type of dependence arises: *interference*. Interference occurs when a variable is defined in one thread and used in a concurrently executing thread.

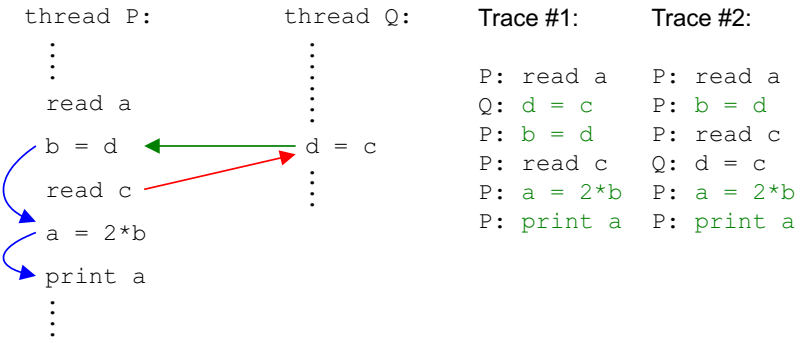


Figure 1.4: A program with two threads

Example 1.3 (Slicing Concurrent Programs): In the example in figure 1.4 we have two threads `P` and `Q` that execute in parallel. In this example, there are two interference dependences: One is due to a definition and a usage of variable `d`, the other is due to accesses to variable `c`.

A simple traversal of interference during slicing will make the slice imprecise because interference may lead to unrealizable paths again. In the example in figure 1.4, a simple traversal will include the `read c` statement into the slice. But there is no possible execution where the `read c` statement has an influence on the assignment `b=d`. A matching execution would require *time travel* because the assignment `b=d` is always executed before the `read c` statement. A path through multiple threads is now realizable if it contains a valid execution chronology. However, even when only realizable paths are considered, the slice will not be as precise as possible. The reason for this imprecision is that concurrently executing threads may *kill* definitions of other threads.

Example 1.4: In the example in figure 1.5 on the next page, the `read a` statement is reachable from the `print a` statement via a realizable path. But there is no possible execution where the `read` statement has an influence on the `print` statement when assuming that statements are atomic. Either the `read` statement reaches the usage in thread `Q` but is redefined afterwards through the assignment `a=2*b` in thread `P`, or the `read` statement is immediately redefined by the assignment `a=2*b` before it can reach the usage in thread `Q`.

Müller-Olm has shown that precise context-sensitive slicing of concurrent programs is undecidable in general [MOS01]. Therefore, we have to use conservative approximations to analyze concurrent programs. A naive approximation would allow time travel, causing an unacceptable loss of precision. Also, we cannot use summary edges to be context-sensitive because summary edges would *ignore* the effects of parallel executing threads. Again, reverting to context-insensitive slicing would cause an unacceptable loss of precision.

To be able to provide precise slicing without summary edges, new slicing algorithms presented in chapter 7 have been developed based on capturing

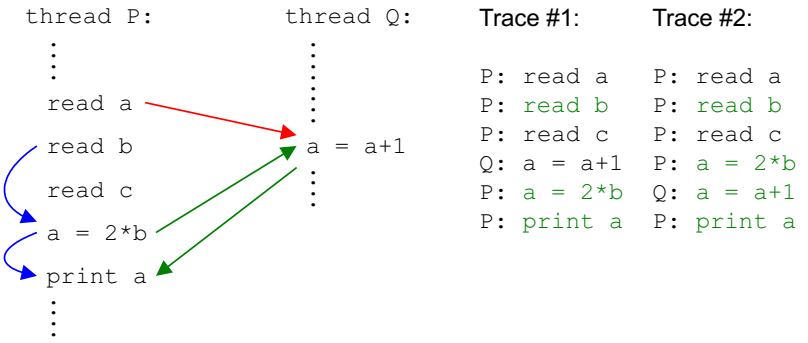


Figure 1.5: Another program with two threads

the calling context through *call strings* [SP81]. Call strings can be seen as a representation of call stacks. They are used frequently for context-sensitive program analysis, e.g. pointer analysis. The call strings are propagated along the edges of the PDG: At edges that connect procedures the call string is used to check that a call always returns to the right call site. Thus, call strings are never propagated along unrealizable paths.

The basic idea for the high-precision approach to slice concurrent programs presented in chapter 8 is the adaption of the call string approach to concurrent programs. The context is now captured through one call string for each thread. The context is then a tuple of call strings which is propagated along the edges in PDGs.

A combined approach avoids combinatorial explosion of call strings: Summary edges are used to compute the slice within threads. Additionally, call strings are only generated and propagated along interference edges if the slice crosses threads. With this approach many fewer contexts are propagated.

This only outlines the idea of the approach—this thesis will present in its first two parts the foundations and algorithms for slicing sequential and concurrent programs in detail. Additionally, a major third part of this thesis will presents optimizations and advanced applications of slicing.

1.2 Applications of Slicing

Slicing has found its way into various applications. Nowadays it is probably mostly used in the area of software maintenance and reengineering. In the following some applications are mentioned to show the broadness but most will not be brought up in this thesis again.

Debugging

Debugging was the first application of program slicing: Weiser [Wei82] realized that programmers mentally ignore statements that cannot have an influence on a statement revealing a bug. Program slicing computes this abstraction and allows one to focus on potentially influencing statements. Dicing [LW87] can be used to focus even more, when additional statements with correct behavior can be identified. Dynamic slicing can be used to focus on relevant statements for one specific execution revealing a bug [ADS93]. More work can be found in [FR01].

Testing

Program slicing can be used to divide a program into smaller programs specific to a test case [Bin92]. This reduces the time needed for regression testing because only a subset of the test cases has to be repeated [Bin98].

In [HD95] *robustness slices*, which give approximate answer to the question whether a program is robust, are presented. Equivalent mutants can be detected by slicing [HHD99].

Other work to be mentioned can be found in [GHS92, BH93b].

Program Differencing and Integration

Program *differencing* is the problem of finding differences between two programs (or between two versions of a program). Semantic differences can be found using program dependence graphs. Program *integration* is the problem of merging two program variants into a single program. With program dependence graphs it can be assured that differences between the variants have no conflicting influence on the shared program parts [HPR89, Hor90, HR91, HR92, RY89].

Software Maintenance

If a change has to be applied to a program, forward slicing can be used to identify the potential impact of the change: The forward slice reveals the part of the program that is influenced by a criterion statement and therefore is affected by a modification to that statement. Decomposition slicing [GL91] uses variables instead of statements as criteria.

Function Extraction and Restructuring

Slicing can also be used for *function extraction* [LV93, LV97]: Extractable functions are identified by slices specified by a set of input variables, a set of output variables and a final statement. *Function restructuring* separates a single function into independent ones; such independent parts can be identified by slicing [LD98, LD99].

Cohesion Measurement

Ott et al [OT89, Ott92, OT93, BO94, OB98] use slicing to measure functional cohesion. They define data slices which are a combination of forward and backward slices. Such slices are computed for output parameters of functions and the amount of overlapping indicates weak or strong functional cohesion.

Other Applications

Slicing is used in model construction to slice away irrelevant code, transition system models are only build for the reduced code [HDZ00]. Slicing is also used to decompose tasks in real-time systems [GH97]. It has even been used for debugging and testing spreadsheets [RRB99] or type checking programs [DT97, TD01].

1.3 Overview

This thesis is structured into three parts: First, intraprocedural analysis, second, interprocedural analysis and slicing of sequential and concurrent programs, and last, another main part: applications of dependence graphs and slicing.

The first part starts with an introduction to data flow analysis in the next chapter. It is followed by an introduction to slicing based on Weiser's original definition and on program dependence graphs. Chapter 4 presents *fine-grained* program dependence graphs and how ANSI C programs can be represented with them. The last chapter of the first part presents an approach to slicing concurrent programs (without procedures).

The second part is structured similarly to the first one: It starts with an introduction to interprocedural data flow analysis. Interprocedural slicing based on program dependence graphs is presented next in chapter 7, which also contains an evaluation of various algorithms. The last chapter of part two describes the new approach to slicing concurrent procedural programs.

The third and main part contains applications. The starting chapter presents two approaches to visualize dependence graphs both graphically and textually. Chapter 10 contains various solutions to the problem that a traditional slice is not focused and usually too large. That chapter also presents and evaluates various chopping algorithms. Chapter 11 shows some approaches to reducing the size of dependence graphs and to increasing their precision. Clone detection based on program dependence graphs is addressed in chapter 12. Path conditions answer the question why a specific statement is in a slice; they are presented in chapter 13. Chapter 14 describes the VALSOFT system in which most of the presented work has been implemented. The last chapter completes this thesis with conclusions.

1.4 Accomplishments

This thesis is self-contained as much as possible and therefore contains presentations of other authors' work. Besides a thorough presentation of slicing, the accomplishments of this thesis are:

- A fine-grained program dependence graph (chapter 4), which is able to represent ANSI C programs including non-deterministic execution order. It is a self-contained intermediate representation and the base of clone detection and path condition computation.
- A high-precision approach to slicing concurrent procedure-less programs (chapter 5). A preliminary version has been published as [Kri98].
- A new approach to slicing concurrent procedural programs (chapter 8). This context-sensitive approach reaches a high precision, despite that precise or optimal slicing is undecidable. This is the first approach that does not need inlining and is able to slice concurrent recursive programs.
- Some variations of slicing and chopping algorithms within interprocedural program dependence graphs and a thorough evaluation of these algorithms (sections 7.3, 7.4, 10.2). Most of this has already been published in [Kri02].
- Fundamental ideas to visualizing dependence graphs (chapter 9), realized in Ehrich's master's thesis [Ehr96].
- Some methods to make the results of slicing more focused (sections 10.1 and 10.3) or more abstract (section 10.4).
- Techniques to reduce the size of program dependence graphs without worsening the precision of slicing (section 11.1).
- An approach to clone detection based on program dependence graphs (chapter 12). This approach has a higher detection rate for modified clones than other approaches, because it identifies similar semantics instead of similar texts. After publication in [Kri01], the benefits and drawbacks of this approach have been evaluated in a clone detection contest.
- Methods to generate path conditions for complex data structures, procedures and concurrent programs (sections 13.2, 13.4 and 13.5). The general approach of path conditions was introduced by Snelting [Sne96] and developed further by Robschink [Rob, RS02, SRK03].
- The design of the VALSOFT system and implementation of the data flow analysis, dependence graph construction and various slicing and chopping algorithms within it (chapter 14).

Part I

Intraprocedural Analysis

Chapter 2

Intraprocedural Data Flow Analysis

The following introduction to intraprocedural data flow analysis is based on the problem of *Reaching Definitions*. A definition is a statement that assigns some value to a variable. Usually, reaching definitions are not defined by the statements in a program, but in terms of the control flow graph, which represents the flow of control between the statements of a program. A definition is said to reach a given statement if there is a path from the definition to the given statement without another definition of the same variable. Such a second definition would *kill* the first—which will no longer reach the given statement on *that* path (it may still reach the given statement on a different path).

2.1 Control Flow Analysis

To analyze a program, the *control flow*—the possible execution sequences of the statements—must be understood first. This is obvious with only well-structured constructs. However, as soon as constructs like goto statements are involved that may lead to unstructured programs, the identification of possible control flow is non-trivial. For this reason, a graph-based representation is usually used. This so-called *flow graph* is built out of the nodes representing the statements and the edges representing the flow of control in between. In contrast to analysis for optimization purposes, we don't merge statements to *basic blocks*—instead, our goal is that any node represents at most one side effect.

A *control flow graph* (CFG) is a directed attributed graph $G = (N, E, n^s, n^e, \nu)$ with node set N and edge set E. The statements and predicates are represented by nodes $n \in N$ and the control flow between statements is represented by *control flow edges* $(n, m) \in E$, written as $n \rightharpoonup m$. E contains control flow edge e, iff the statement represented by node $\mathsf{target}(e)$ may immediately be executed after the statement represented by $\mathsf{source}(e)$, i.e. no other statement is executed

```
1    sum = 0
2    mul = 1
3    a = 1
4    b = read()
5    while (a <= b) {
6      sum = sum + a
7      mul = mul * a
8      a = a + 1
9    }
10   write(sum)
11   write(mul)
```

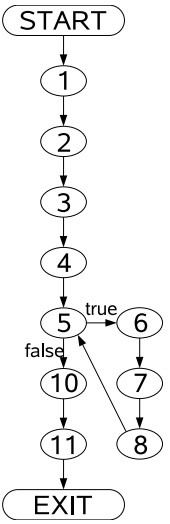


Figure 2.1: Example program and its CFG

in between. Two special nodes $n^s \in N$ and $n^e \in N$ are distinguished, the START node n^s and the EXIT node n^e, which represent beginning and end of the program. Node n^s does not have predecessors and node n^e does not have successors. The function $\nu : E \to \{\mathsf{true}, \mathsf{false}, \epsilon\}$ is a mapping of the edges to their attributes: If the statement represented by $\mathsf{source}(e)$ has an out-degree > 1, it contains a predicate controlling which statement is executed afterwards and the outgoing edges are attributed with true or false—the outcome of the predicate. Other edges are marked with ϵ, which means not attributed.

Example 2.1: Figure 2.1 shows a small program and its control flow graph. The nodes are numbered with the corresponding line number of the represented statement.

The variables that are referenced at node[1] n are denoted by $\mathsf{ref}(n)$, the variables that are defined (or assigned) at n are denoted by $\mathsf{def}(n)$. Both functions can be bound more formally to the nodes by extending the definition of the control flow graph to $G = (N, E, n^s, n^e, \mu, \nu)$, where μ is a function mapping nodes to their attributes. It is defined as $\mu(n) = (\mathsf{def}(n), \mathsf{ref}(n))$ or $\mu_{\mathsf{def}}(n) = \mathsf{def}(n)$, $\mu_{\mathsf{ref}}(n) = \mathsf{ref}(n)$.

Most interesting problems are questions as to whether something (some information or property) at a node n *reaches* a different node m. Part of such a question is whether node m is *reachable* from n at all. Formally, a node n is *reachable* ($m \rightharpoonup^\star n$) from another node m, if there is a path $\langle m, \ldots, n \rangle$ in G, i. e. "$\rightharpoonup^\star$" is the transitive, reflexive closure of "$\rightharpoonup$". Normally it is assumed

[1] In the rest of this work we will use "node" and "statement" interchangeable, as they are bijectively mapped.

that for any path in the CFG a corresponding execution of the program exists, such that all statements are executed in the same order as the nodes of the path. Of course, this is only a conservative approximation, as paths might exist that are actually *unrealizable*. However, in general this is undecidable and therefore all paths are assumed *realizable*.

If we pick some statements out of the node sequence of a path they are a *witness* of a possible path:

Definition 2.1 (Witness)
A sequence $\langle n_1, \ldots, n_k \rangle$ of nodes is called a *witness*, iff $n_i \rightharpoonup^\star n_{i+1}$ for all $1 \leqslant i < k$.

This means that a sequence of nodes is a witness, if all nodes are part of a path through the CFG in the same order as in the sequence. Of course, every path is a witness of itself, but a sequence can be a witness of multiple different paths.

Normally, the nodes of the flow-graph can be constructed in a syntax-directed way during parsing. Most of the edges can be constructed that way, too. Edges for goto statements etc. can be back-patched.

2.2 Data Flow Analysis

The purpose of data flow analysis is to compute the behavior of the analyzed program in respect to the generated data, usually used for program optimizations, i.e. to check the circumstances for the application of optimizations. The easiest and most widespread solutions are analyses that iterate over the control flow graph (called *iterative data flow analysis*). Another approach omits the control flow graph and does the analysis directly on top of the (abstract) syntax tree (called *syntax-directed data flow analysis*). Both approaches are presented next.

2.2.1 Iterative Data Flow Analysis

The discussion of the reaching definitions example is resumed by formalizing the problem:

Definition 2.2 (Reaching Definitions)
For any node $n \in N$ in flow-graph G, let $\mathsf{def}(n)$ be the set of variables defined at n. A definition of variable v at a node n ($v \in \mathsf{def}(n)$) *reaches* a (not necessarily different) node n', if a path $P = \langle n_1, \ldots, n_k \rangle$ in G exists, such that

1. $k > 1$
2. $n_1 = n \wedge n_k = n'$
3. $\forall 1 < i < k : v \notin \mathsf{def}(n_i)$

The computation of reaching definitions can be described within a data flow analysis framework. Therefore, the local abstract semantics are described with a transfer function over the set of definitions x that reach a node n:

$$[\![n]\!](x) = \big(x - \mathsf{kill}(n)\big) \cup \mathsf{gen}(n)$$

This equation means that definitions reaching the entry of a node n which are not killed by n, reach the exit of n together with definitions generated by n. Definitions can be represented in multiple ways. One possibility is to represent them by node-variable pairs (n, v) containing the variable v that is defined at n:

$$\begin{aligned} \mathsf{gen}(n) &= \{(n, v) \mid v \in \mathsf{def}(n)\} \\ \mathsf{kill}(n) &= \{(n', v) \mid v \in \mathsf{def}(n) \wedge v \in \mathsf{def}(n')\} \end{aligned}$$

Such representations can be simplified by enumerating all occurring definitions: the set D contains all definitions of a program, the sets D_v contain all definitions for a specific variable v and the sets D_n contain all definitions at a node n. Then the equations are

$$\begin{aligned} \mathsf{gen}(n) &= D_n \\ \mathsf{kill}(n) &= \bigcup_{v \in \mathsf{def}(n)} D_v \end{aligned}$$

which are constant functions.

The next step is to define the abstract semantics over a path $p = \langle n_1, \ldots, n_k \rangle$, with the identity function $[\![]\!]_{id}(x) = x$:

$$[\![p]\!] = \begin{cases} [\![]\!]_{id} & p = \langle\rangle \\ [\![\langle n_2, \ldots, n_k \rangle]\!] \circ [\![n_1]\!] & \text{otherwise} \end{cases}$$

A definition reaches a node n, if a path from the START node n^s to n exists, on which the definition is not killed according to definition 2.2 on the page before. At the START node, no definition is available (an empty set of definitions). If there exists more than one path, the reaching definitions of all paths are merged (by union in this case):

$$\mathsf{RD}_{\mathsf{MOP}}(n) = \bigcup_{p = \langle s, \ldots, n \rangle} [\![p]\!](\emptyset)$$

This is an instance of a *meet-over-all-paths (MOP)* solution. In presence of loops there are infinite paths, which make the computation of the MOP solution impossible. Therefore, only the *minimal-fixed-point (MFP)* solution[2] is computed:

$$\mathsf{RD}_{\mathsf{MFP}}(n) = \begin{cases} \emptyset & n = s \\ [\![n]\!](\bigcup_{m \rightharpoonup n} \mathsf{RD}_{\mathsf{MFP}}(m)) & \end{cases}$$

[2]Depending on the data flow problem, it can also be the maximal fixed point.

Because of the properties of the transfer functions and the data flow sets, the MFP and the MOP solution coincide: The data flow sets form a (complete) semi-lattice and the transfer functions form a distributive function space:

Definition 2.3 (Lattice)
A *lattice* $\mathcal{L} = (\mathcal{C}, \sqcap, \sqcup)$ consists of a set of values $\mathcal{C}$, a *meet* operation $\sqcap$ and a *join* operation $\sqcup$, such that

1. $\forall x \in \mathcal{C} : x \sqcup x = x$ and $x \sqcap x = x$ (idempotency)
2. $\forall x, y \in \mathcal{C} : x \sqcup y = y \sqcup x \wedge x \sqcap y = y \sqcap x$ (commutativity)
3. $\forall x, y, z \in \mathcal{C} : (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \wedge (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ (associativity)
4. $\forall x, y \in \mathcal{C} : x \sqcup (x \sqcap y) = x$ and $x \sqcap (x \sqcup y) = x$ (absorption)

With only one operator $\mathcal{L} = (\mathcal{C}, \sqcup)$ is a *semi-lattice*. A (semi-) lattice induces a partial order $\sqsubseteq$ on the elements of $\mathcal{C}$.

Lattices are used to represent the *data flow facts*, in the example of reaching definitions the sets of definitions. Most data flow facts are sets, where the lattice is the powerset. This can be represented as bit-vectors: if the set of data flow facts is D, than $\mathcal{C} = 2^D$.

Definition 2.4 (Monotone Function Space)
A set of functions $\mathcal{F}$ defined on semi-lattice $\mathcal{L} = (\mathcal{C}, \sqcup)$ is a *monotone function space*, if

1. $\forall f \in \mathcal{F} : \forall x, y \in \mathcal{C} : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ (monotonicity)
2. $\exists f_{id} \in \mathcal{F} : \forall x \in \mathcal{C} : f_{id}(x) = x$ (identity function)
3. $\forall f, g \in \mathcal{F} : f \circ g \in \mathcal{F} \wedge \forall x \in \mathcal{C} : f \circ g(x) = f(g(x))$
   (closed under composition)

Definition 2.5 (Data Flow Analysis Framework)
A *monotone data flow analysis framework* $\mathcal{A} = (\mathcal{L}, \mathcal{F}, \sqcup)$ consists of

1. the (complete) semi-lattice $\mathcal{L} = (\mathcal{C}, \sqcup)$ with meet $\sqcup$ for the data flow facts, and
2. the monotone function space $\mathcal{F}$ defined on $\mathcal{L}$.

A monotone data flow analysis framework is a *distributive data flow analysis framework*, if $\forall f \in \mathcal{F} : \forall x, y \in \mathcal{C} : f(x \sqcup y) = f(x) \sqcup f(y)$.

The iterative algorithm 2.1 on the following page of a monotone data flow analysis framework computes the MFP solution; the monotonicity guarantees termination and the existence of a fix-point. The MFP solution is a correct albeit not necessarily precise solution of the data flow problem. It coincides

Algorithm 2.1 Iterative computation of the MFP solution

Input: The control flow graph $G = (N, E, n^s, n^e)$
The function space $\mathcal{F}$ of transfer function $[\![n]\!]$
The semi-lattice $\mathcal{L} = (\mathcal{C}, \sqcup)$ of the data flow information
An initial value $i \in \mathcal{C}$
Output: The mapping in from nodes to its data flow information

foreach $n \in N$ **do**
  $in[n] = \bot$
$in[n^s] = i$
Initialize the worklist:
$W = \{n \mid n^s \rightharpoonup n \in E\}$
while $W \neq \emptyset$, *worklist is not empty* **do**
  $W = W/\{n\}$, *remove one element from worklist*
  $v = \bigsqcup_{m \rightharpoonup n \in E} [\![m]\!](in[m])$
  if $v \neq in[n]$ **then**
    $in[n] = v$
    $W = W \cup \{m \mid n \rightharpoonup m \in E\}$
return in

with the correct and precise MOP solution if it is based on a distributive data flow analysis framework.

The equations of the reaching definitions fit in a distributive data flow analysis framework:

- The semi-lattice $\mathcal{L}$ is the powerset of the set D of all definitions in a program ($\mathcal{C} = 2^D$), with meet operator $\sqcup = \cup$.
- The function space $\mathcal{F}$ is the set of all transfer functions $[\![n]\!]$ including the identity $[\![]\!]_{id}$.

It can be proved that the function space is a distributive function space and therefore the algorithm for the MFP solution computes the correct and precise MOP solution.

2.2.2 Computation of def and ref for ANSI C

Up to now, the existence of def and ref has been assumed. However, the computation of def and ref are non-trivial in presence of complex data structures. This thesis focuses on the analysis of ANSI C [Int90], which makes the analysis of expressions a complex task when the original program is not transformed into a simpler intermediate representation first. The execution order in expressions is (mostly) undefined in ANSI C and expressions cannot be represented in a control flow graph structure. Therefore the expressions' subtrees are copied from the abstract syntax tree into the nodes of the control flow graph. The rest

$$
\begin{aligned}
E &\rightarrow s \\
E &\rightarrow c \\
E &\rightarrow E_1\texttt{+}E_2 \\
E &\rightarrow E_1\texttt{=}E_2
\end{aligned}
$$

Figure 2.2: A simple grammar with assignments as expressions

$$
\begin{aligned}
E_1 \rightarrow s_2 &: (d_1, r_1, a_1) = (\emptyset, \emptyset, \{(n, v_s)\}) \\
E_1 \rightarrow c &: (d_1, r_1, a_1) = (\emptyset, \emptyset, \emptyset) \\
E_1 \rightarrow E_2\texttt{+}E_3 &: (d_1, r_1, a_1) = (d_2 \cup d_3, r_2 \cup r_3 \cup a_2 \cup a_3, \emptyset) \\
E_1 \rightarrow E_2\texttt{=}E_3 &: (d_1, r_1, a_1) = (d_2 \cup d_3 \cup a_2, r_2 \cup r_3 \cup a_3, \emptyset)
\end{aligned}
$$

Figure 2.3: Computation of def, ref and acc

of this section shows how def and ref can be computed in a syntax-directed way via the copied expression subtrees.

Figure 2.2 shows a grammar for simple expressions to start with. As in C, expressions can be used as *l-* or *r-values*, which means as targets of assignments (to the *left*) or as expressions to be evaluated (to the *right* of assignments). This has to be distinguished if a variable s (a symbol) is accessed in an expression. Therefore, a third set acc is used in addition to def and ref which captures the accessed variables. For production $E \rightarrow s$ the access of s if represented by (n, v): variable v is accessed at the node n in the abstract syntax tree. Because it is not clear if the access to v is a use or a definition, only acc is set to $\{(n, v)\}$, while ref and def are empty. An access to a constant ($E \rightarrow c$) generates no access to a variable. An operation on expressions like in production $E \rightarrow E_1\texttt{+}E_2$ causes the evaluation of E_1 and E_2 and any variable that is accessed in E_1 or E_2 must now be put into ref. A definition through an assignment $E \rightarrow E_1\texttt{=}E_2$ causes the variables accessed in E_1 to be defined and put into def, while the variables accessed in E_2 are evaluated and put into ref. This behavior is represented as equations in figure 2.3, where (d, r, a) is used as a shorthand for $(\mathsf{def}, \mathsf{ref}, \mathsf{acc})$.

Arrays

Now, a production $E \rightarrow a\texttt{[}E_1\texttt{]}$ is added to allow array usage in the program. In general it is undecidable if two array usages $a\texttt{[}E_1\texttt{]}$ and $a\texttt{[}E_2\texttt{]}$ access the same array element. Therefore all elements of an array are accesses to the array variable. An assignment to an array element only kills one element and leaves the other untouched. There are two approaches to modeling this:

1. Any assignment to an array element is assumed to be a *non-killing* assignment, i.e. a new definition is created, but none eliminated. The equations are extended with

$$E_1 \to E_2[E_3] \quad : \quad (d_1, r_1, a_1) = (d_2 \cup d_3, r_2 \cup r_3 \cup a_3, a_2)$$

   and also the previous definition of $\mathsf{kill}(n)$ must be changed that no array variable is killed:

$$\mathsf{kill}(n) \quad = \quad \{(n', v) \mid v \in \mathsf{def}(n) \wedge v \in \mathsf{def}(n') \wedge v \text{ is not an array}\}$$

2. Any assignment to an array element is assumed to be a *killing modification*, i.e. the array is used before killed (the array is modified). Then the equations are extended with

$$E_1 \to E_2[E_3] \quad : \quad (d_1, r_1, a_1) = (d_2 \cup d_3, r_2 \cup r_3 \cup a_3 \cup a_2, a_2)$$

   and the definition of $\mathsf{kill}(n)$ stays unchanged.

Both approaches are conservative approximations and must be chosen dependent on the intended application. Due to other reasons which will be explained later, the killing modification is the preferred way in this work.

Structures and Unions

ANSI C allows aggregation of types to complex data types, called structures and unions. A variable of structure or union type contains fields that can be accessed without accessing the other fields. To handle structures and unions *scalar replacement* is used. Any access to a variable (or a field of a variable) is replaced by accesses to the contained fields as scalar variables. Because fields can be structures or unions themselves, this is done recursively. Let $\mathrm{SR}(v)$ return the set of variables generated by scalar replacement for a variable v. Then the equations are extended and modified with:

$$E_1 \to E_2.f \quad : \quad (d_1, r_1, a_1) = (d_2, r_2, \{(n, v_i.f) \mid (n, v_i) \in a_2\})$$

The scalar replacement is done in the $\mathsf{kill}(n)$ and $\mathsf{gen}(n)$ functions:

$$\begin{aligned} \mathsf{gen}(n) &= \{(n, v) \mid v' \in \mathsf{def}(n) \wedge v \in \mathrm{SR}(v)\} \\ \mathsf{kill}(n) &= \{(n', v) \mid v' \in \mathsf{def}(n) \wedge v' \in \mathsf{def}(n') \wedge v \in \mathrm{SR}(v')\} \end{aligned}$$

Unions cause a slight problem because the fields of a union share the same space: an access to a field may access some other fields of the union, too. Again, this is handled through the scalar replacement: Let $\mathrm{UR}(v)$ return the set of variables which share a union with variable v.

$$\begin{aligned} \mathsf{gen}(n) &= \{(n, v) \mid v' \in \mathsf{def}(n) \wedge v \in \mathrm{SR}(v') \cup \mathrm{UR}(v')\} \\ \mathsf{kill}(n) &= \{(n', v) \mid v' \in \mathsf{def}(n) \wedge v' \in \mathsf{def}(n') \wedge v \in \mathrm{SR}(v')\} \end{aligned}$$

Because it is compiler dependent which other fields of a union share the space with an accessed field, a definition must not kill other fields of the union.

Pointer Usage

A real problem is the presence of pointers: They cannot be tackled by a simple solution like scalar replacement. A special data flow analysis is needed which also handles the combination with structures, unions and arrays. Pointer analysis is too complex to be discussed here and we assume a function $PT(v, n)$ that returns—for an access to a variable v—the set of variables v can point to at n. Again, this function is integrated into the $\mathsf{kill}(n)$ and $\mathsf{gen}(n)$ functions using the scalar replacement. An extensive amount of literature exists for pointer and alias analysis and among hundreds of works, [HP00, Hin01] can be used for an introduction. For the analysis system which will be discussed later, a flow-insensitive but context-sensitive alias analysis of Burke et al [BCCH95] has been implemented.

The equations are extended to obey pointer usage:

$$\begin{aligned} E_1 \to \&E_2 &: (d_1, r_1, a_1) = (d_1, r_1, \{(n, \&v) \mid (n, v) \in a_2\} \\ E_1 \to *E_2 &: (d_1, r_1, a_1) = (d_1, r_1, \{(n, *v) \mid (n, v) \in a_2\} \\ E_1 \to E_2\texttt{->}f &: (d_1, r_1, a_1) = (d_1, r_1, \{(n, *v.f) \mid (n, v) \in a_2\} \end{aligned}$$

Other Forms of Assignments

ANSI C knows a series of special forms of assignments that are trivial to analyze:

$$\begin{aligned} E_1 \to E_2\texttt{+=}E_3 &: (d_1, r_1, a_1) = (d_2 \cup d_3 \cup a_2, r_2 \cup r_3 \cup a_3, a_2) \\ E_1 \to \texttt{++}E_2 &: (d_1, r_1, a_1) = (d_2 \cup a_2, r_2 \cup a_2, a_2) \end{aligned}$$

where `+=` stands for any modifying assignment and `++` stands for any pre- or post-modifying operator.

2.2.3 Control Flow in Expressions

For some expressions, ANSI C defines an execution order. It even allows some expressions to be evaluated or not. Examples for such expressions are:

Comma operator. Expressions separated by the comma operator are evaluated in left-to-right order and the rightmost expression is used as result. All expressions to the left of a comma are evaluated (*r*-value expressions) and the last expression can be used as a *r*-value or even as an *l*-value, e.g. `(a,b,c).f` or `(x,y)=(a,b)` is possible.

Question-mark operator. The question-mark operator is like an 'if' for expressions. Depending on the outcome of the predicate to the left of the question operator `?`, one of the two expressions to the right is used as *r*- or *l*-value, e.g. `a=x?y:z` or even `x?y:z=a`.

Shortcut operator. Predicates using logical shortcut operators `||` or `&&` allow to skip the evaluation of the expression to the right of the operator if the result of the complete expression is clear after evaluating the expression to the left. For example in `p || q` the expression `q` must not be evaluated if `p` has been evaluated to `true`.

For compiling purposes, such expressions are normally transformed to statements first, so that a control flow graph can be used and all expressions are free of control flow. Again, if transformations are not allowed, these expressions need special attention. First of all, definitions (and uses) have to be distinguished between *must* happen and *may* happen: a definition to the left of a shortcut operator must happen, if the complete expression is evaluated, but a definition to the right may or may not happen. Second, all equations must be extended to compute def, ref and acc in may- and must-versions. The may-versions are notated as $d^?, r^?, a^?$ and the must-versions as $d^!, r^!, a^!$. Some of the equations are shown in figure 2.4 on the next page.

Because these three classes of operations define control flow inside expressions, the *killing* of uses and definitions must be obeyed. The presented equations ignore killing so far. Instead of changing def, ref and acc, the equations are extended to already compute kill and gen. For presentation purposes, only some extended equations are shown in figure 2.5 on page 24 and aliasing and scalar replacement is ignored.

2.2.4 Syntax-directed Data Flow Analysis

A different approach to the problem of reaching definitions is to ignore the control flow graph and to solve it based on the abstract syntax tree. The key insight is that in well-structured programs the control flow is obvious. The following equations show how the transfer functions are combined:

$$\begin{aligned} [\![\texttt{if E then } S_1 \texttt{ else } S_2]\!] &= ([\![S_1]\!] \cup [\![S_2]\!])[\![E]\!] \\ [\![S_1]\!]; [\![S_2]\!] &= [\![S_2]\!][\![S_1]\!] \\ [\![\epsilon]\!] &= [\![]\!]_{id} \end{aligned}$$

Under the condition that $[\![S]\!][\![S]\!] = [\![S]\!]$ holds (for reaching definitions), the following equation is valid:

$$[\![\texttt{while E do } S_1]\!] = [\![E]\!] \cup ([\![E]\!][\![S_1]\!][\![E]\!])$$

This might raise the impression that a syntax-directed approach is easier to implement than an iterative framework. However, this is a wrong impression—looking at the syntax-directed computation of gen and kill in the previous sections, you get a glimpse of an implementation's complexity. The problem is not the theoretical complexity, but the sheer amount of implementation work. Despite that, the implementation of the analysis tool presented in the next chapters includes a traditional as well as a syntax-directed computation of reaching definitions for evaluation and debugging purposes.

$$
\begin{array}{l}
E_1 \to s_2 : \\
\quad (d_1^!, r_1^!, a_1^!) = (\emptyset, \emptyset, \{(n, \nu_s)\}) \\
\quad (d_1^?, r_1^?, a_1^?) = (\emptyset, \emptyset, \emptyset)
\end{array}
$$

$$
\begin{array}{l}
E_1 \to c : \\
\quad (d_1^!, r_1^!, a_1^!) = (\emptyset, \emptyset, \emptyset) \\
\quad (d_1^?, r_1^?, a_1^?) = (\emptyset, \emptyset, \emptyset)
\end{array}
$$

$$
\begin{array}{l}
E_1 \to E_2 + E_3 : \\
\quad (d_1^!, r_1^!, a_1^!) = (d_2^! \cup d_3^!, r_2^! \cup r_3^! \cup a_2^! \cup a_3^!, \emptyset) \\
\quad (d_1^?, r_1^?, a_1^?) = (d_2^? \cup d_3^?, r_2^? \cup r_3^? \cup a_2^? \cup a_3^?, \emptyset)
\end{array}
$$

$$
\begin{array}{l}
E_1 \to E_2 = E_3 : \\
\quad (d_1^!, r_1^!, a_1^!) = (d_2^! \cup d_3^! \cup a_2^!, r_2^! \cup r_3^! \cup a_3^!, \emptyset) \\
\quad (d_1^?, r_1^?, a_1^?) = (d_2^? \cup d_3^? \cup a_2^?, r_2^? \cup r_3^? \cup a_3^?, \emptyset)
\end{array}
$$

$$
\begin{array}{l}
E_1 \to E_2 ? E_3 : E_4 : \\
\quad (d_1^!, r_1^!, a_1^!) = (d_2^!, r_2^! \cup a_2^!, \emptyset) \\
\qquad\quad d_1^? = d_2^? \cup d_3^? \cup d_4^? \cup d_3^! \cup d_4^! \\
\qquad\quad r_1^? = r_2^? \cup r_3^? \cup r_4^? \cup a_2^? \cup r_3^! \cup r_4^! \\
\qquad\quad a_1^? = a_3^? \cup a_4^? \cup a_3^! \cup a_4^!
\end{array}
$$

$$
\begin{array}{l}
E_1 \to E_2 \,||\, E_3 : \\
\quad (d_1^!, r_1^!, a_1^!) = (d_2^!, r_2^! \cup a_2^!, \emptyset) \\
\qquad\quad d_1^? = d_2^? \cup d_3^? \cup d_3^! \\
\qquad\quad r_1^? = r_2^? \cup r_3^? \cup a_2^? \cup a_3^? \cup r_3^! \cup a_3^! \\
\qquad\quad a_1^? = \emptyset
\end{array}
$$

$$
\begin{array}{l}
E_1 \to E_2, \ldots, E_n : \\
\quad d_1^! = \bigcup_{2 \leqslant i \leqslant n} d_i^! \\
\quad r_1^! = \bigcup_{2 \leqslant i \leqslant n} r_i^! \cup \bigcup_{2 \leqslant i < n} a_i^! \\
\quad a_1^! = a_n^! \\
\quad d_1^? = \bigcup_{2 \leqslant i \leqslant n} d_i^? \\
\quad r_1^? = \bigcup_{2 \leqslant i \leqslant n} r_i^? \cup \bigcup_{2 \leqslant i < n} a_i^? \\
\quad a_1^? = a_n^?
\end{array}
$$

Figure 2.4: Computation of may- and must-versions

$E_1 \to s_2$:

$$\begin{array}{lcl} (d_1^!, r_1^!, a_1^!, g_1^!, k_1^!) & = & (\emptyset, \emptyset, \{(n, \nu_s)\}, \emptyset, \emptyset) \\ (d_1^?, r_1^?, a_1^?, g_1^?, k_1^?) & = & (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \end{array}$$

$E_1 \to E_2 + E_3$:

$$\begin{array}{lcl} g_1^! & = & g_2^! \cup g_3^! \\ k_1^! & = & k_2^! \cup k_3^! \\ g_1^? & = & g_2^? \cup g_3^? \\ k_1^? & = & k_2^? \cup k_3^? \end{array}$$

$E_1 \to E_2 = E_3$:

$$\begin{array}{lcl} g_1^! & = & g_2^! \cup g_3^! \cup a_2^! \\ k_1^! & = & k_2^! \cup k_3^! \cup \{\nu | (n, \nu) \in a_2^!\} \\ g_1^? & = & g_2^? \cup g_3^? \cup a_2^? \\ k_1^? & = & k_2^? \cup k_3^? \cup \{\nu | (n, \nu) \in a_2^?\} \end{array}$$

$E_1 \to E_2 ? E_3 : E_4$:

$$\begin{array}{lcl} g_1^! & = & \{(n, \nu) \mid (n, \nu) \in g_2^! \cup a_2^! \wedge \nu \notin k_3^! \cap k_4^!\} \\ k_1^! & = & k_2^! \cup (k_3^! \cap k_4^!)) \\ g_1^? & = & \{(n, \nu) \mid (n, \nu) \in g_2^? \cup a_2^? \wedge \nu \notin k_3^! \cap k_4^!\} \cup g_3^? \cup g_4^? \cup g_3^! \cup g_4^! \\ k_1^? & = & k_2^? \cup k_3^? \cup k_4^? \cup (k_3^! - k_4^!) \cup (k_4^! - k_3^!)) \end{array}$$

Figure 2.5: Computation of may- and must-versions of gen and kill

The syntax-directed approach is even possible with well-structured jumps like break, continue or return statements. This is based on a simple trick: every statement (which may contain other statements) is assumed to be left through one of four possible exits:

1. The statement is entered, computation proceeds and it is left normally; no jump is encountered.
2. The statement is entered and left by a return statement (after some computation).
3. The statement is left by a break statement, or
4. by a continue statement.

Now, for every statement four matching transfer functions are defined describing the data flow on each of the four possible executions.

2.3 The Program Dependence Graph

A *program dependence graph* [FOW87] is a transformation of a CFG, where the control flow edges have been removed and two other kinds of edges have been inserted: *control dependence* and *data dependence* edges. These edges represent the effects of control flow and reaching definitions more directly: control dependence exists, if one statement is controlling the execution of a different statement. Data dependence exists, if a definition at a node reaches a different node and is used there.

2.3.1 Control Dependence

Node m is called a *post-dominator* of Node n, if any path from n to the EXIT node n^e must go through m. A node n is called a *pre-dominator* of m if any path from the START node n^s to m must go through n. In typical programs, statements in loop bodies are pre-dominated by the loop entry and post-dominated by the loop exit.

Definition 2.6 (Control Dependence)
A node m is called *(direct) control dependent* on node n, if

1. there exists a path p from n to m in the CFG ($n \rightharpoonup^{\star} m$),
2. m is a post-dominator for every node in p except n, and
3. m is not a post-dominator for n.

This basically means that at node n at least two outgoing edges exist: all paths to the EXIT node along one edge pass through m and all paths along the other edge don't.

Depending on the definition whether a node post- or pre-dominates itself, a node can be control dependent on itself. Some authors do not allow this, some do. If a node post-dominates itself, the predicate of a while loop is control dependent on itself. On the other hand, if a node never post-dominates itself, the control dependences of while loops and if-statements are identical.

There are two ways to compute control dependence: the traditional and the syntax-directed approach, which will both be presented next.

Syntax-directed Computation of Control Dependence

Within well-structured programs that just contain `if`-statements and loops without jump statements like `goto`, `break`, `continue` or `return`, the control dependence can easily be computed during traversal of the abstract syntax tree:

- Every statement is directly control dependent on its enclosing `if`- or loop-statement predicate.
- If nodes post-dominate themselves, every loop-statement predicate is control dependent on itself.

An important observation is that when nodes never post-dominate themselves, the control dependence subgraph of a well-structured program is a tree.

Traditional Computation of Control Dependence

The traditional approach to compute control dependence is to first compute the post-dominator tree with the fast Lengauer-Tarjan algorithm [LT79]. The second step is to traverse all control flow edges: For every $m \rightharpoonup n$ the ancestors of n in the post-dominator tree are traversed backwards to the parent of m, marking all visited nodes as control dependent on m (see [FOW87] for a detailed description).

2.3.2 Data Dependence

Two nodes are data dependent on each other, if a definition at one node might be used at the other node:

Definition 2.7 (Data Dependence)
A node m is called *data dependent* on node n, if

1. there is a path p from n to m in the CFG ($n \rightharpoonup^* m$),
2. there is a variable v, with $v \in \mathsf{def}(n)$ and $v \in \mathsf{ref}(m)$, and
3. for all nodes $k \neq n$ of path p, $v \notin \mathsf{def}(k)$ holds.

This is very much similar to the problem of reaching definitions: if a definition of a variable v at node n is a reaching definition at m and m uses variable v, then m is data dependent on n. Thus, the computation of data dependence is straightforward: compute the reaching definitions first and then check every node if it uses a variable of a reaching definition.

2.3.3 Multiple Side Effects

Up to now, the possibility of multiple side effects in expressions has been ignored. Because assignments are expressions, an expression may contain multiple assignments. In ANSI C it is common to make use of multiple assignments: pre- and post-modifying operators are very comfortable and typically used as in `x = a[i++]`. But what happens inside a single expression if a variable has a side effect and is accessed a second time? The ANSI C standard [Int90] allows an undefined behavior under such circumstances. This is adequate for compilers, but not for reverse engineering, program understanding or debugging purposes. There are two ways to deal with this:

1. Follow the ANSI C standard and ignore data dependence inside expressions. This might be awkward for users especially in conjunction with procedure calls (an example will be given in section 4.1).

2. Allow data dependence inside expressions. However, these dependences cannot be 'back-patched' while computing kill and gen: an expression like `x = x+1` contains a use and a definition of `x`, but has no inside data dependence. Therefore the equations for (d, r, a, g, k) must be extended to compute the generated data dependences.

The implemented analysis system uses the second approach. However, the extensions to the equations will provide no new insights and thus are not presented here.

2.3.4 From Dependences to Dependence Graphs

Once the control and data dependence have been computed, it is a simple task to build a dependence graph: The *program dependence graph (PDG)* consists of the nodes of the CFG, control dependence edges $n \overset{cd}{\prec} m$ for nodes m which are control dependent on nodes n, and data dependence edges $n \overset{dd}{\prec} m$ for nodes m which are data dependent on nodes n.

Example 2.2: Figure 2.6 on the next page shows the PDG for the example in figure 2.1 on page 14. The nodes are exactly those of the corresponding control flow graph, except for the absent EXIT node.

Without modifications, the control dependence subgraphs have no single root because the top most statements will not be control dependent on anything. On the other hand, it is desirable that there exists a single root, which should be the START node. This is usually achieved by inserting an irrelevant control flow edge from the START to the EXIT node. The effect is that no other node than the EXIT node post-dominates the START node and the START node will be the root in the control dependence subgraph. Normally, the EXIT node will be omitted from the program dependence graph, as it has no in- or outgoing dependence edges. The irrelevant edge is ignored during data flow analysis.

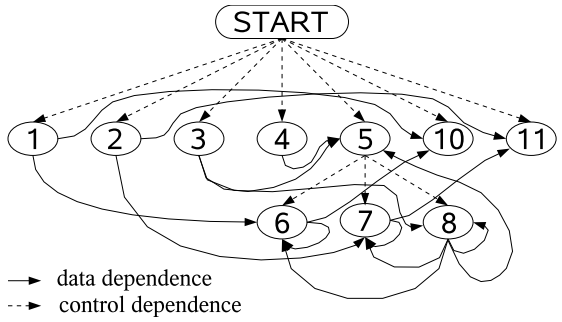


Figure 2.6: The program dependence graph for figure 2.1

This special handling of the START node can be seen as the introduction of a *region* node that joins nodes of a region. In [FOW87] such nodes are inserted at several places. It is observed that control dependence is nonintuitive, as a predicate node has more than one outgoing control dependence edge marked with `true` and mixed with more than one marked with `false`. The purpose of region nodes is to summarize the set of outgoing control dependence edges with the same attribute and group together all nodes with incoming control dependence edges coming from the same node with the same attribute. The result is that predicate nodes have only two successors like in the control flow graph. An algorithmic solution which inserts such region nodes is given in [FOW87]. Instead, it is now shown how such region nodes can be inserted by a modified construction of the control flow graph.

Remember the insertion of an irrelevant edge to make START a region node. The insertion of irrelevant edges can also be used for insertion of other region nodes. In well structured programs, the regions which should be summarized can be identified clearly as single-entry-single-exit blocks. Between the predicate node and the entry of such a block a control flow edge attributed with `true` or `false` exists. The first step is to insert a region node before the entry node, an unattributed control flow edge from the region to the entry node and redirect the attributed control flow edge to the region node instead. The region node will use the identity transfer function $[\![]\!]_{\mathrm{id}}$ and therefore this modification has no influence on the results of the data flow analysis. The second modification is to insert an irrelevant edge from the region node to the successor of the block's exit node. Because the program is well structured, the node has only one successor.

Example 2.3: Figure 2.7 on the next page shows a small program and its (traditional) CFG. First, the modifications will insert an irrelevant edge between START and EXIT. Next, two new region nodes are inserted for the two single-entry-single-exit regions consisting of nodes 3/4 and 6/7. They are also connected to the predicate node and the last node of their region (nodes 4 resp. 7). The resulting CFG is shown in Figure 2.8 on page 30, together with the result-

```
1    a = read()
2    if (a>0) {
3      b = a + 1
4      c = a * 2
5    } else {
6      b = a - 1
7      c = a / 2
8    }
9    write(b)
10   write(c)
```

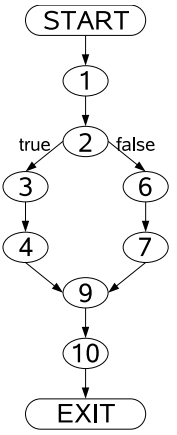


Figure 2.7: Example program and its CFG

ing PDG. The nodes of the two branches are now cleanly separated under the two region nodes.

2.4 Related Work

Broad discussions of intraprocedural data flow analysis are contained in [Hec77, MJ81, ASU85, Muc97, NNH99]. The data flow analysis frameworks have been introduced and proved in [Kil73, KU77].

Dependence graphs have long been an intermediate representation in compiler optimizations [KKP+81], mostly representing variations of data dependence. Control dependence is the main contribution of program dependence graphs, which have been introduced in [OO84, FOW87], targeted as an intermediate representation for optimization and as an representation for software development environments. Horwitz et al [HPR88] show that program dependence graphs capture a program's behavior. Cartwright and Felleisen [CF89] show that a PDG has the same semantics as the program it represents. Selke [Sel89] also worked on semantics of dependence graphs.

A syntax-directed approach to construct program dependence graphs is presented in [HMR93, HR96]. This approach basically constructs the control dependence subgraph like presented in section 2.3.1. Because the control dependence subgraph is ordered, it is used instead of the control flow graph for iterative computation of the data dependences. In presence of unstructured control flow, the control dependence subgraph is adjusted and additional information is inserted.

Program dependence graphs also have many applications in software engineering [HR92]. A main application there is program slicing, which will be discussed in the next chapter.

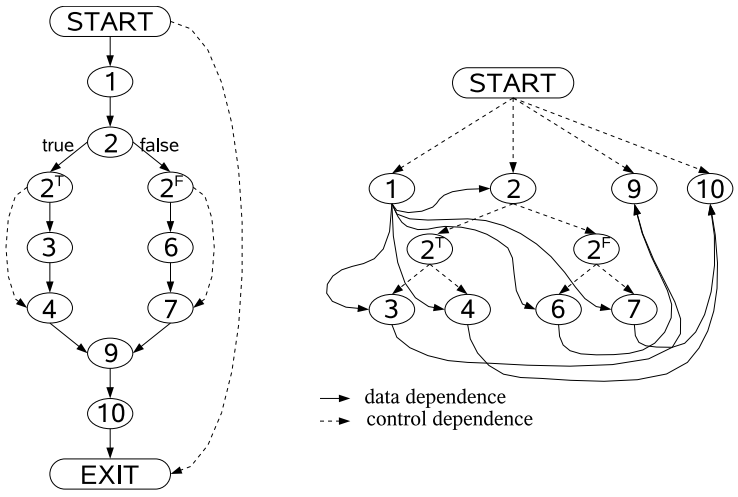


Figure 2.8: Modified CFG and its PDG

Chapter 3

Slicing

Mark Weiser showed in [Wei82] that programmers mentally build abstractions of programs while debugging. These abstractions, called *slices*, are informally defined as statements that may have an influence on a statement under consideration. In [Wei84] he gave a formal definition of slices and an algorithm to compute them. Since Weiser's publishings many variations and applications of his original definitions arose. In the following, his original approach (called Weiser-style slicing) and another widespread approach based on program dependence graphs will be presented.

3.1 Weiser-style Slicing

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

Definition 3.1 (Weiser-style Slice)
A slice $S(c)$ of a program P on a slicing criterion c is any executable program P', where

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

The most trivial (but irrelevant) slice of a program P is always the program P itself. Slices of interest are as small as possible:

Definition 3.2 (Statement-Minimal Slice)
A slice $S(c)$ of a program P with respect to a criterion c is a *statement-minimal slice*, iff no other slice of P with respect to c with fewer statements exists.

In general, it is undecidable if a slice $S(c)$ is statement minimal [Wei84]. Weiser presented an approximation based on identifying relevant variables and statements. Implemented as an iterative data flow analysis, the directly relevant variables and statements with respect to criterion c are computed:

$$\begin{aligned} R_c^0(n) &= \bigcup_{n \rightharpoonup m} \{v \in R_c^0(m) \mid v \notin \mathsf{def}(n)\} \\ &\cup \bigcup_{n \rightharpoonup m} \{v \in \mathsf{ref}(n) \mid \mathsf{def}(n) \cap R_c^0(m) \neq \emptyset\} \\ S_c^0 &= \bigcup_{n \rightharpoonup m} \{n \mid \mathsf{def}(n) \cap R_c^0(m) \neq \emptyset\} \end{aligned}$$

A variable v is relevant at a node n, $v \in R_c^0(n)$, iff it is relevant at a successor m of n and not defined (killed) at n, or v is referenced (used) at n and a variable w exists that is defined at n and relevant at a successor m. A statement n is relevant, $s \in S_c^0$, iff a variable v which is defined at n exists and is relevant at a successor m of n. R_c^0 can be written as a transfer function:

$$[\![n]\!]_{R_c^0}(x) = (x - \mathsf{def}(n)) \cup \{v \in \mathsf{ref}(n) \mid x \cap \mathsf{def}(n) \neq \emptyset\}$$

For a criterion $c = (s, V)$, the framework is initialized with $R_c^0(s) = V$ and $\forall n \neq s : R_c^0(n) = \emptyset$. The computed minimal fixed point of S_c^0 contains basically the reaching definitions for variables from V at s together with the transitive data dependent nodes. The next step is to trace control dependence: the *range of influence* $\mathrm{INFL}(n)$ contains the statements that are control dependent on n. The results from the first step are now expanded iteratively until a fixed point is reached:

$$\begin{aligned} B_c^k &= \{m \mid S_c^k \cap \mathrm{INFL}(m) \neq \emptyset\} \\ R_c^{k+1}(n) &= R_c^k(n) \cup \bigcup_{m \in B_c^k} R^0_{(m, \mathsf{ref}(m))}(n) \\ S_c^{k+1} &= B_c^k \cup \bigcup_{n \rightharpoonup m} \{n \mid \mathsf{def}(n) \cap R_c^{k+1}(m) \neq \emptyset\} \end{aligned}$$

The *indirectly* relevant variables R_c^k and statements S_c^k are expanded by the set B_c^k of all statements on which the relevant statements are control dependent:

- The relevant variables $R_c^k(n)$ are expanded by variables directly relevant to the statements in B_c^k: Basically, new criteria $(m, \mathsf{ref}(m))$ are constructed, consisting of the statements $m \in B_c^k$ and the referenced variables at m. The directly relevant variables are computed for the new criteria and added to the indirectly relevant variables.
- The relevant statements S_c^k are computed from the statements of B_c^k and all statements having relevant variables.

The final slice $S(c)$ is the minimal fixed point for S_c^k.

3.2 Slicing Program Dependence Graphs

Ottenstein and Ottenstein [OO84] were the first who suggested the use of program dependence graphs to compute Weiser's slices. Slicing on the PDG of a sequential program is a simple graph reachability problem, because control and data dependence is transitive.

Definition 3.3
A node m is called *transitive dependent* on node n, if

1. there is a path $p = \langle n_1, \ldots, n_k \rangle$ with $n = n_1$ and $m = n_k$ where every n_{i+1} is control or data dependent on n_i, and
2. p is a witness in the CFG.

Note that the composition of control and data dependence is always transitive: A dependence between x and y and a dependence between y and z imply a path between x and z resulting from the definition of control and data dependence.

Observation 3.1 A path in a PDG implies a path in the CFG:

$$m \multimap^\star n \text{ in the PDG} \Longrightarrow m \multimap^\star n \text{ in the CFG}$$

Definition 3.4 (Slice)
The (*backward*) *slice* $S(n)$ of a PDG at node n consists of all nodes on which n (transitively) depends:

$$S(n) = \{m \mid m \multimap^\star n\}$$

The node n is called the *slicing criterion*.

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

The definition of a slice in program dependence graphs is implemented easily through a graph reachability algorithm: A (backward) slice to a node (the *slicing criterion*) is the set of nodes from which the criterion node is reachable (algorithm 3.1 on the next page).

Example 3.1: Figure 3.1 on the following page shows the example program from the last chapter (figures 2.1 on page 14 and 2.6 on page 28). The shown slice is $S(11) = \{2, 3, 4, 5, 7, 8, 11\}$: the computation of the sum has no influence on the computation of the product.

Algorithm 3.1 Slicing in PDGs

Input: A PDG $G = (N, E)$
A slicing criterion $s \in N$
Output: The slice $S \subseteq N$

$W = \{s\}$
$S = \{s\}$, *mark* s *as visited*
while $W \neq \emptyset$, *worklist is not empty* **do**
  $W = W/\{w\}$, *remove one element from worklist*
  foreach $v \rightharpoonup w \in E$ **do**
    if $v \notin S$, *reached node is not yet marked* **then**
      $W = W \cup \{v\}$
      $S = S \cup \{v\}$, *mark reached node as visited*
return S, *the set of all visited nodes*

```
1  sum = 0
2  mul = 1
3  a = 1
4  b = read()
5  while (a <= b) {
6    sum = sum + a
7    mul = mul * a
8    a = a + 1
9  }
10 write(sum)
11 write(mul)
```

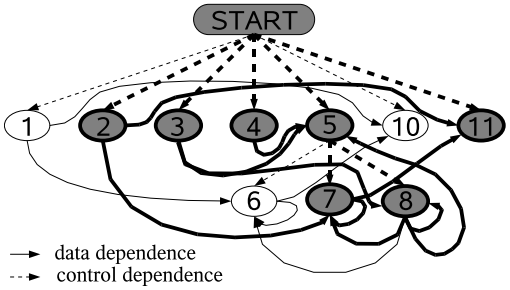


Figure 3.1: Sliced example program and its PDG

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

Definition 3.5 (Forward Slice)
The *forward slice* $S^F(n)$ of a PDG at node n consists of all nodes that (transitively) depend on n:

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

Algorithm 3.1 on the preceding page can be used to compute forward slices, if "$v \rightharpoonup w \in E$" is replaced by "$w \rightharpoonup v \in E$".

3.3 Precise, Minimal, and Executable Slices

A slice computed by graph reachability or in Weiser-style is only an approximation of a (statement) minimal slice. The presented approaches do not necessarily compute good approximations, as they sometimes cannot identify irrelevant statements.

Example 3.2: Slices computed for statement 7 in the following program will usually contain the complete program, despite that only statement 1 has an influence on statement 7 (and the statement-minimal slice would only contain lines 1 and 7).

```
1    read(a)
2    read(b)
3    c = 2 * b
4    a = a - b
5    a = a + c
6    a = a - b
7    write(a)
```

Some of such constructs can be identified, if techniques from optimization like constant-propagation are applied before computing program dependence graphs and slices. More complex constructs can be identified by symbolic execution techniques. However, even with the most sophisticated techniques, the minimal slice stays undecidable in general.

As long as procedures are ignored, both presented approaches generate *executable* slices, i.e. the computed slice is an executable subset of the original program. If this requirement is dropped, such that the computed slice only contains relevant statements but must not produce the same behavior when executed, statement-minimal slices will even be smaller.

Example 3.3:

```
1    read(x)
2    read(y)
3    y = y + x
4    x = y - x
5    y = y - x
6    write(x)
7    write(y)
```

In this example[1], the values of x and y are swapped (assuming x and y are integers). A minimal slice for line 7 will only contain lines 1 and 7. An executable minimal slice must contain all statements except line 6 to behave correctly.

Both examples rely on "tricks" with expressions. Even when such tricks are not used, the computation of precise slices remains complex. Therefore Weiser has considered *data flow minimal* slices [Wei79], where unrealizable paths and exact semantics of expressions are ignored. For that purpose, *program schemas* are used [LPP70]: A schema of a program is generated by replacing all expressions with symbolic expressions in the form $f(v_1, \ldots, v_n)$, where f is a function or predicate.

Definition 3.6 (Data Flow Minimal Slices)
Let S be a given schema and T a schema obtained by deleting statements from S. T is *data flow minimal*, if under all interpretations i, T(i) is a slice of S(i) and it is not possible to delete more statements.

As yet, it is not clear if such data flow minimal slices are computable; no known algorithm computes a data flow minimal slice.

3.4 Unstructured Control Flow

The original slicing algorithms assumed a language without jump statements. Such statement cause a small problem: because the (correct) control flow graph only contains a single outgoing edge, no other node can be control dependent on the jump statement. Therefore, a (backward) slice will never contain the jump statements (jump statements have no outgoing data dependence). The exclusion of jump statements makes executable slices incorrect and non-executable slices uneasy to comprehend. Various approaches handle this problem differently: Ball and Horwitz [BH93a] augment the CFG by adding *non-executable* edges between the jump statement and the immediate following statement. This makes both the statements at the jump target and the one following the jump statement control dependent on the jump. During data dependence computation, the non-executable edges are ignored. The resulting

[1] Kindly provided by Mark Harman. Such code was used to swap register contents in the days when memory access was very expensive.

program dependence graph can be sliced with the normal slicing algorithm, except that labels (the jump targets) are included in the slice if a jump to that label is in the slice.

3.5 Related Work

Slicing based on PDGs was introduced in [OO84] and proved to compute correct slices in [RY89]. The relation of program slicing and program specialization has been presented in [RT96]. A foundation of slicing and dependence on term rewriting systems has been presented by Field and Tip [FT98].

A characterization of slices similar to data flow minimal slices has been used in [MOS01]: *optimal slices*. However, a definition of optimal slices is not given.

A parallel version of Weiser's slicing algorithm is presented in [DHS95]; it is based on conversion into networks of concurrent processes.

Bergeretti and Carré [BC85] presented an algorithm which is neither Weiser-style nor dependence graph based. Their algorithm uses the concept of information flow relations. Hausler [Hau89] and Venkatesh [Ven91] independently developed denotational approaches for slicing.

Not only imperative programs can be sliced: In [SH96] tree-like structures are sliced. More specific approaches are slicing of hierarchical state machines [HW97], software architecture descriptions [Zha98], web applications [RT01], class hierarchies [TCFR96], knowledge-based systems [VA00], logical programs [Vas99], Prolog [SD96], hardware description languages [INIY96, CFR+99] and hardware-software codesign [Rus02]. Problems of implementing Weiser-style slicing for C are discussed in [JZR91].

More related work for slicing of object-oriented programs will be presented in chapter 7.

3.5.1 Unstructured Control Flow

The algorithm of Ball and Horwitz [BH93a] is basically the same approach as Choi and Ferrante's first algorithm in [CF94]. Their second algorithm computes a normal slice first (which does not include the jump statements) and then adds goto statements until the slice is correct. This algorithm produces slices that are more precise than slices from the first algorithm, but the additional gotos may not be part of the original program and the computed slices do not match the definition of a slice. Agrawal [Agr94] presents an algorithm which also adds jump statements to a normal slice to make it correct. However, the added statements are always part of the original program. Harman and Danicic present another algorithm [HD98], which is an extension of Agrawal's.

Kumar and Horwitz [KH02] present an improved algorithm, which is based on the augmented control flow graph, and a modified definition of control dependence using both the augmented and the non-augmented control flow

graph. They also present a modified definition of a slice based on *semantic effects* (similar to the *semantic dependence* of [PC90]), which basically inverts the definition of a slice: A statement `x` of program `P` has a semantic effect on a statement `y`, iff a program `P'` exists, created by modifying or removing `x` from `P`, and some input `I` exists such that `P` and `P'` halt on `I` and produce different values for some variables used at `y`. However, such a slice may be a superset of a Weiser's slice because a statement like "`x = x;`" has a semantic effect according to their definition.

3.5.2 Other Forms of Slicing

Section 10.5.1 contains a discussion of *dynamic slicing*: There a slice is computed for a specific execution defined by the input variables' values. In *amorphous program slicing* [HD97], a slice is generated by any simplifying transformation, not only by statement deletion. Binkley [Bin99] has shown how to compute amorphous slices with dependence graphs.

Errors in programs are normally ignored during slicing, [HSD96] presents an approach to handle errors explicitly for slicing.

Chapter 4

The Fine-Grained PDG

Traditional dependence graphs contain back-references to intermediate representations that were used to construct them, either high-level representations like the abstract syntax tree or low-level representations like quadruples (low-level representations can also be embedded in the nodes of a dependence graph). Without back-references, advanced applications of dependence graphs like the ones presented in chapter 11 or even a visualization of slices in the source code are not possible.

Our variant of the program dependence graph is a specialization of the traditional one [HRB90], which has been adapted to the following goals:

- The PDG should be as similar to the AST and the source code as possible to enable an instinctive mapping and understanding of the (visualized) graph.
- The PDG should be a complete (intermediate) representation, without references to other representations except positions in source code.

Because of these goals, transformations like SSA [CFR+91] (static single assignment form) were not applicable and the possibility of multiple side effects in expressions had to be dealt with directly (see section 2.3.3). Therefore a fine-grained representation is used and kept similar to both the AST and the traditional PDG: On the level of statements and expressions, the AST nodes are almost mapped one to one onto PDG nodes.

In this chapter it will be shown how such a fine-grained PDG is represented and constructed.

4.1 A Fine-Grained Representation

To carry the needed information, the nodes have three main attributes: They may be attributed with a class, an operator and a value. The class specifies the kind of node: statement, expression, procedure call etc. The operator specifies

it further, e.g. binary expression, constant etc. The value carries the exact operator, like "+" or "−", constant values or identifier names. Other attributes are the enclosing procedure and a mapping to the source text.

To handle the specialized nodes we must specialize the edges, too. In most cases the execution order of expression components is undefined in ANSI C [Int90] which results in representation problems. For example, consider the statement `z=f()+g()`: It is undefined if `f()` is evaluated and called before `g()` or vice versa. It is only defined that both have been evaluated before the returned values will be added. This behavior can not be represented in a normal CFG because it does not allow non-deterministic execution orders. For optimization purposes, the ANSI C standard allows the compiler to choose an arbitrary execution order. If such an expression contains conflicting side-effects (a location is accessed at least twice and at least one access is a write), the standard declares the behavior as completely undefined. Most compilers accept such programs silently, however, software maintenance tools should handle this behavior, either reporting or handling it intelligently.

Example 4.1: Consider the following example, where procedures `f` and `g` have conflicting assignments to global variable `a`. In the example three statements are marked as slicing criteria for backward slices (lines 4, 9 and 16). A software maintenance tool should either reject this program because the behavior of line 15 is undefined or should compute slices that allow both possible execution orders of `f()` and `g()`. Therefore, line 5 should be included in slices A and C, but not in B, and line 10 should be included in slices A and B, but not in C.

```
1    int a, b, c;
2
3    int f() {
4      b = a;          /* slice criterion B */
5      a = 1;          /* must be included in A and C */
6      return 1;
7    }
8    int g() {
9      c = a;          /* slice criterion C */
10     a = 2;          /* must be included in A and B */
11     return 1;
12   }
13   int main () {
14     int z;
15     z = f() + g();
16    return a;       /* slice criterion A */
17   }
```

A short experiment with this example shows that tools that just employ traditional optimization techniques accept this program silently and assume an arbitrary execution order. CodeSurfer[1] [AT01] and Icaria are two available

[1] As of version 1.8, users can specify an order of evaluation, either left-to-right or right-to-left.

slicing tools (presented in section 14.2). Both assume that `g()` executes after `f()`, and their backward slices at A do not include line 4. Even worse, Icaria computes empty slices for criteria B and C. Strictly speaking, both tools are correct, because the behavior is undefined. However, this might confuse the user and the fine-grained approach does a better job.

The representation of an expression `z=x+y` in the fine-grained PDG is that the nodes of `x` and `y` are control dependent on the "+" node. This control dependence is called *expression (control) dependence* because it is local to expressions. Expression dependence allows arbitrary execution order. To be able to compute slices for any node in the PDG the data flow between the expression components had to be represented in it. Here, the value of the addition is dependent on the value of `x` and `y`. If simple data dependence edges would have been added between the nodes of `+` and `x` and the nodes of `+` and `y`, cycles would have been induced between the nodes. One way to solve this would be to split the node of `+` into a "before evaluation of subcomponents" node and an "after evaluation of subcomponents" node. A different approach has been taken that introduces another specialized edge: the *value dependence edge*, which is like a data dependence edge between expression components. The resolution of the arising cycles has been delegated to the slicer and the other tools. Another specialized edge was needed to represent the assignments of values to variables: The *reference dependence edges* are similar to the value dependence edges, except that they do not induce cycles.

Definition 4.1
Let n and m be two nodes of the same expression.

1. The expression node n is *expression dependent* on expression node m if n and m are always executed together and m is executed after n.
2. The expression node n is *value dependent* on expression node m if the value computed at m is needed at node n.
3. The expression node n is *reference dependent* on (assignment) expression node m if the value computed at m is stored into a variable at n.

Example 4.2: Figure 4.1 on the following page shows an excerpt of the PDG for lines 4 and 5 of the following code.

```
1    a = 1;
2    b = 2;
3    c = 3;
4    x = a * (y = b + c);
5    z = x + y;
```

The nodes are all expression nodes with different operators. For example, node 20 is an expression node with operator kind "binary" and value "+". Note that assignments are expressions, e.g. node 13 is an expression node with an operator "assign" and no value.

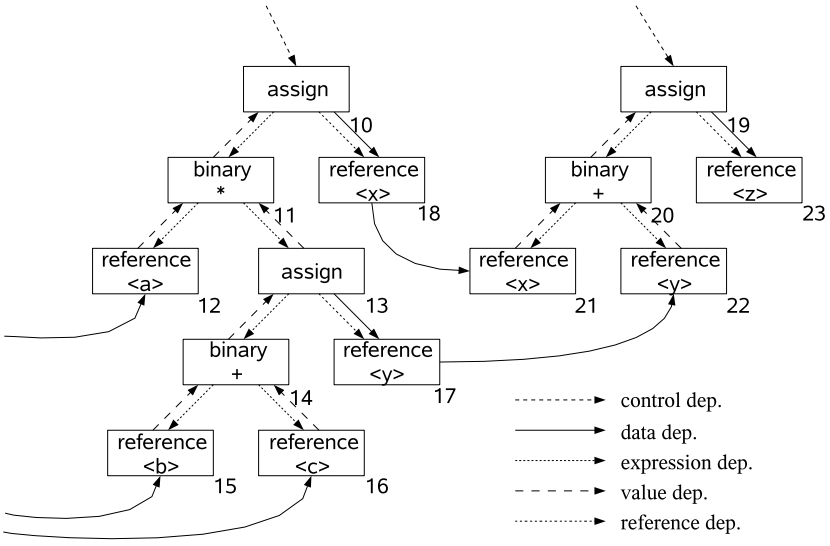


Figure 4.1: Excerpt of a fine-grained PDG

4.2 Data Types

The use of composite data types and pointers introduces new problems, which will be discussed next. Not only the analysis of data types is relevant, but also the representation in the fine-grained PDG, too. Agrawal et al. [ADS91] use abstract memory locations and define the following situations for two expression e_1 and e_2, where e_1 is a definition and e_2 is a usage:

Complete intersection: The locations corresponding to e_1 are a superset of the locations corresponding to e_2.

Maybe intersection: It cannot be determined statically whether or not the locations corresponding to e_1 and e_2 coincidence.

Partial intersection: The locations corresponding to e_1 are a subset of the locations corresponding to e_2.

Examples for these three situations will be presented in the next three sections, where the three main classes of data types are discussed.

4.2.1 Structures

Structures are aggregates of fields. The selection of such a field introduces complete or partial intersection between the selected field and the complete variable. Maybe intersection is impossible with normal structures, it is introduced only by unions.

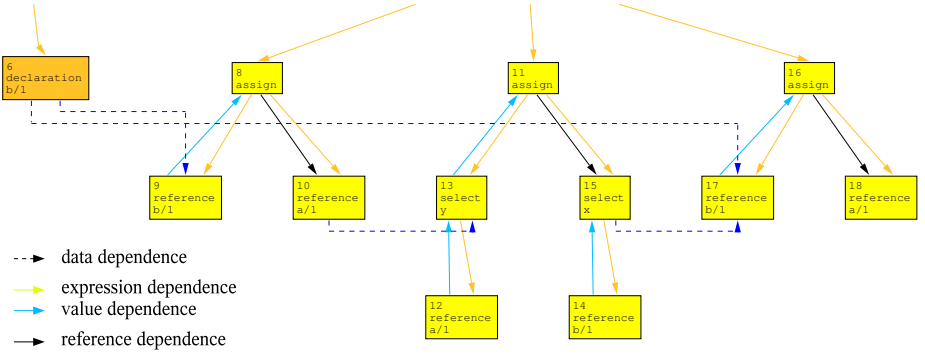


Figure 4.2: Partial PDG for a use of structures

Example 4.3:

```
1    struct s {
2        int x, y;
3    } a, b;
4    a = b;
5    b.x = a.y;
6    a = b;
```

A partial PDG for this fragment is shown in figure 4.2. The usage of `b` in the first assignment `a = b` is a complete intersection and therefore is represented like a simple variable in node 9 with a data dependence between nodes 6 and 9. The selection of field `y` of variable `a` is modeled through an expression node with an operator "select" and a value of `y` (nodes 12 and 13). The use of `a.y` at node 13 has a complete intersection with the definition of `a` at node 10 (not vice versa), therefore a data dependence between nodes 10 and 13 exists (note that node 12 has no data dependence). The definition of `b.x` at node 15 is a partial intersection with the use of `b` at node 17. Therefore node 17 is data dependent on node 15 (partial intersection) and the earlier definition of `b` at node 6 (the declaration), where a complete intersection exists.

All this does not need special handling for the fine-grained PDG, as the data flow analysis already handles the different types of intersection by scalar replacement of aggregates (section 2.2.2). However, the scalar replacement is only virtual and does not change the original program, which causes a loss of precision.

Example 4.4: Figure 4.3 on the following page presents an example for scalar replacement: The first column shows a small program and the second column shows the precise slice for statement 8 as criterion. The third column shows the slice as computed by the presented slicer. Because statement 7 is a usage of variable `b`, both assignments to fields of `b` are included in the slice, despite that

	Program	Slice	VALSOFT	CodeSurfer	Icaria
1	`int x;`				
2	`t a, b;`				
3	`a.x = 1;`	`a.x = 1;`	`a.x = 1;`	`a.x = 1;`	`a.x = 1;`
4	`b.x = 2;`	`b.x = 2;`	`b.x = 2;`	`b.x = 2;`	`b.x = 2;`
5	`a.y = 3;`	`a.y = 3;`	`a.y = 3;`	`a.y = 3;`	`a.y = 3;`
6	`b.y = 4;`	`b.y = 4;`	`b.y = 4;`	`b.y = 4;`	`b.y = 4;`
7	`a = b;`	`a = b;`	`a = b;`	`a.y = b.y;` `a.x = b.x;`	`a = b;`
8	`x = a.y;`	`x = a.y;`	`x = a.y;`	`x = a.y;`	`x = a.y;`

Figure 4.3: Slices as computed by different slicers

statement 4 has no influence on statement 8. If scalar replacement is applied before analyzing the program, the slice will be precise (see column four). This technique is used by CodeSurfer [AT01].

If the structure `s` of the earlier example had been a union (fields `x` and `y` are mapped to the same location), the PDG in figure 4.2 on the page before would not be different. However, in general, unions must be handled differently—again, the data flow analysis already takes care about that.

4.2.2 Arrays

The use of arrays may also introduce complete, maybe and partial intersection.

Example 4.5: The three different types of intersection are shown at the following code:

```
1    int i, x;
2    int a[20];
3    a[1] = 0;
4    a[i] = 1;
5    x = a[1];
```

Figure 4.4 on the facing page shows the matching fragment from the fine-grained PDG. Access to array element is represented by an "array" node, which has two subtrees, one for the accessed array and one for the index expression.

In this example, the value of `x` may be 0 or 1, which is dependent on the value of `i`. The statement `a[i] = 1` might kill the definition in the previous statement `a[1] = 0` because `i` might be `1`.

In general, it is undecidable if two accesses to two array elements of the same array access the same element. Therefore a maybe intersection is assumed. Usually, a maybe intersection is assumed to be a non-killing definition of all elements in an array. In this example this would generate three data dependences: between line 2 and 5 (complete intersection), between line 3 and 5

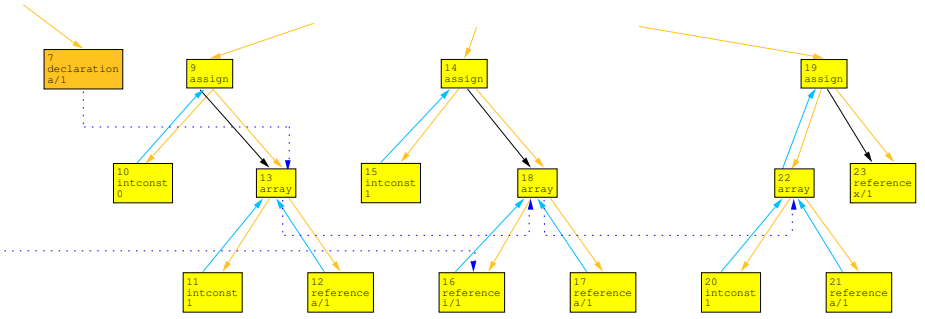


Figure 4.4: Partial PDG for a use of arrays

and between line 4 and 5 (both maybe intersections). However, with this representation, the information that statement 4 might kill the definition at statement 3 is lost. This is undesirable for the main application of the fine-grained program dependence graph—the generation of path conditions presented in chapter 13.

Therefore a different approach[2] is taken as shown in figure 4.4. An assignment to an array element is considered a *killing modification*: an assignment to an array element uses the (complete) array before the specified element is defined. Therefore the definitions at nodes 13 and 18 (lines 3 and 4) are "uses", too and data dependence edges $7 \rightharpoonup 13$ (from line 3 to 4) and $13 \rightharpoonup 18$ (from line 4 to 5) between definitions have been inserted. There is no direct dependence between node 13 and 22, only a transitive dependence. The handling of array elements has already been done during data flow analysis, and the desired representation does not need special handling.

4.2.3 Pointers

The use of pointers introduces aliasing. The problem of determining potential aliases is undecidable in general and a conservative approximation is used. The potential aliases are already computed during data flow analysis through alias or points-to analysis (see section 2.2.2). Again, only the representation is important here.

Special expression nodes have been introduced for that purpose: the "refer" operator represents the creation of an address and the "derefer" operator represents the dereferencing of a pointer. The data dependence edges are inserted according to the points-to set at those points of the program where pointers are used or defined.

[2]This has no effect on slicing. It is only needed for reasons explained in section 13.2.1 on page 198.

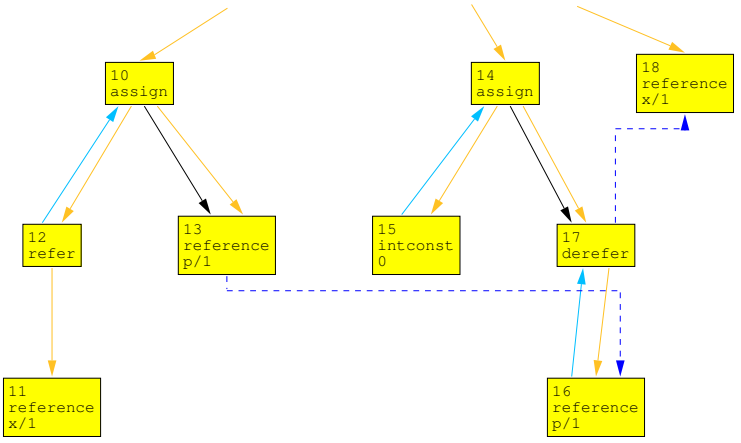


Figure 4.5: Partial PDG for a use of pointers

Example 4.6:

```
1    int x, *p;
2    p = &x;
3    *p = 0;
4    x;
```

A partial PDG of this code is shown in figure 4.5. At node 18 (line 4) the value of variable `x` is used, which is dependent on node 17, where an alias of it becomes defined (line 3). The alias has been introduced at node 13 (line 2), which adopts a data dependence edge $13 \rightharpoonup 16$. Note that the address operation from statement 2 does not introduce any dependence from `x` to `p` (absence of a value dependence edge $11 \rightharpoonup 12$).

Complete pointer analysis must include special analysis of structures and arrays, too. Therefore, structures and arrays are analyzed in two different ways: Once for the pointer analysis and a second time during computation of data dependence. One could be tempted to leave out the special analysis of structures and arrays, just do the complete pointer analysis—however, this may cause imprecision. Icaria is a slicer using this approach, computing an imprecise slice for the example in figure 4.3 on page 44.

4.3 Slicing the Fine-Grained PDG

The slicing algorithm is a direct adaption of algorithm 3.1 on page 34 with special handling of the cycles induced by value dependence edges. The algorithm is based on an implicit splitting of nodes that are sources of expression dependence edges and targets of value dependence edges. (The subexpressions

that are represented by the target nodes of expression dependence edges are evaluated before the value is used at the target node of the value dependence edges.) The slicing algorithm does not traverse value dependence edges if the current node has been reached by an expression dependence edge and vice versa. Different from the earlier algorithm 3.1 on page 34 the slicing algorithm for fine-grained PDGs 4.1 on the next page works with marking edges instead of nodes.

Example 4.7: Lets do a forward slice from node 12 in figure 4.1 on page 42 and see what impact a change to variable `a` has. The algorithm will first include node 11 into the slice. The algorithm will not follow the expression dependence edge from node 11 to 13, because it has reached node 11 through a value dependence edge. The algorithm will continue and add the following nodes to the slice: 10, 18, 21, 20, 19 and 23. Again, node 22 will not be reached. Now, let node 22 be the slicing criterion for a backward slice: the algorithm will include the nodes 17, 13, 11 and 10 probably first. It will also include nodes 14, 15 and 16. But it will not include node 12, because node 11 has been entered via a value dependence edge.

The algorithm must also handle the problem that a node can be reached a second time via a different edge, invalidating the restrictions of an earlier visit. Consider a backward slice for node 20 as an example. The algorithm might first reach node 22 and computes the backward slice from there as shown before, excluding node 12. Now, the algorithm also reaches node 21 which leads to node 18. There, node 10 must be visited a second time, because it had been entered via an expression dependence edge in the first place and is now entered via a reference dependence edge. For the same reason, nodes 11 and 13 are visited again via a value dependence edge and node 12 is now reached for the first time.

4.4 Discussion

The presented approach of fine-grained program dependence enabled slicing and other analyses with one single intermediate representation. The following chapters (especially chapters 11–13) will show some applications that therefore don't need another intermediate representation. The usage of a single intermediate representation is efficient: A second intermediate representation would have caused additional efforts in building it, linking between the representations and keeping both representations synchronized while manipulating one representation.

The direct handling of multiple side effects in conjunction with undefined execution order was only possible with this approach—traditional intermediate representations must revert to arbitrarily chosen execution orders. So far, present slicers (except the one based on this work) may assume behavior different to the actual behavior of executed programs.

Algorithm 4.1 Slicing Fine-Grained PDGs

Input: A PDG $G = (N, E)$
A slicing criterion $s \in N$
Output: The slice $S \subseteq N$

$W = \{e \mid e = n \rightharpoonup s \in E\}$
$M = W$, *mark all* e *as visited*
$S = \{s\}$
while $W \neq \emptyset$, *worklist is not empty* **do**
  $W = W/\{e = m \rightharpoonup n\}$, *remove one element from worklist*
  $S = S \cup \{m\}$
  foreach $e' = m' \rightharpoonup m \in E$ **do**
    if $e \notin M$, *edge is not yet marked* **then**
      if e is an expression dependence edge **then**
        if e' is not a value dependence edge **then**
          $M = M \cup \{e'\}$
          $W = W \cup \{e'\}$
      elsif e is a value dependence edge **then**
        if e' is not an expression dependence edge **then**
          $M = M \cup \{e'\}$
          $W = W \cup \{e'\}$
      else
        $M = M \cup \{e'\}$
        $W = W \cup \{e'\}$
return S, *the set of all visited nodes*

All these advantages come not without disadvantages. There are two main obstacles: First, some algorithms are more complex when using the fine-grained instead of the traditional program dependence graphs. For example, algorithm 4.1 on the preceding page must handle cycles due to expression and value dependence edges. Second, the fine-grained representation is very detailed, causing some implementation overhead.

4.5 Related Work

The original definition of program dependence graphs already was prepared for a fine-grained representation [OO84, FOW87]. The imprecision problem of multiple side-effects in statements has also been identified in [LC94b], and a fine-grained PDG based on parse trees is used. The approach in [Ste98] does not construct a PDG at all, but instead inserts dependence edges directly into the AST, which is then used for slicing. The imprecision problem of multiple side-effects is solved in statement based slicer by transforming the program into a single-side-effect form before analysis. Another fine-grained but highly specialized representation are value dependence graphs (VDG) [WCES94], which also can be used for slicing [Ern94].

The effect of complex data structures on program slicing have been discussed for pointers [ADS91, LB93, LH99b, FTAM99] and for aliasing of parameters [Bin93b].

Libraries and system calls are usually analyzed through *stubs* or *models*, i.e. replaced by source code with the desired behavior in respect to the analysis. Further problems with I/O are discussed in [SHD97].

Chapter 5

Slicing Concurrent Programs

Today, even small programs use concurrent execution and languages like Ada or Java have required features built-in. The analysis of programs where some statements may explicitly be executed concurrently is not new. The *static* analysis of these programs is complicated, because the execution order of concurrently executed statements is *dynamic*. Testing and debugging of concurrent programs have increased complexity: They may produce different behavior even with the same input. The nondeterministic behavior of a program is hard to understand and finding harmful nondeterministic behavior is even harder. Therefore, supporting tools are required. Unfortunately, most tools for sequential programs are not applicable to threaded programs as they cannot cope with the nondeterministic execution order of statements. One simple way to circumvent these problems is to simulate these programs through *sequentialized* or *serialized* programs [UHS97]. These are "product" programs, in which every possible execution order of statements is modeled through a path where the statements are executed sequentially. This may lead to exponential code explosion, which is unacceptable for analysis. Therefore, special representations of concurrent programs have been developed.

In the following a new notation of threaded programs is introduced by extending the control flow graph (CFG) and program dependence graph (PDG) to their *threaded* counterparts tCFG and tPDG. Based on tPDGs a more precise slicing algorithm is presented and it is shown how to extend the basic model of concurrency to allow synchronization and communication.

5.1 The Threaded CFG

A *thread* is a part of a program that must be executed on a single processor. Threads may be executed concurrently on different processors or interleaved on a single processor. In the model used it is assumed that threads are created through `cobegin/coend` statements and that they are properly synchronized on statement level (statements are assumed to be atomic). Let the set of threads

```
1    x = ...;
2    i = 1;
     cobegin {
        if (x>0) {
3         x = -x;
4         i = i+1;
        } else {
5         i = i+1;
        }
      }{
6       i = i+1;
7       z = y;
      } coend;
8     ... = i;
```

Figure 5.1: A threaded program

be $\Theta = \{\theta_0, \theta_1, \ldots, \theta_k\}$. For simplicity we consider the main program to be thread θ_0.

Example 5.1: A sample program with two threads is shown in Figure 5.1. Thread θ_1 is the block of statements 3, 4 and 5, the thread θ_2 is the block with 6 and 7. 1, 2 and 8 are part of the main program θ_0.

A *threaded* CFG (tCFG) extends the CFG with two special nodes COSTART and COEXIT that represent the `cobegin` and `coend` statements. The enclosed threads are handled like complete procedures and will be represented by whole CFGs, which are embedded in the surrounding CFG. The START and EXIT nodes of these CFGs are connected to the COSTART and COEXIT nodes with special *parallel* flow edges. The edges are distinguished through $n \overset{cf}{\rightharpoonup} m$ for a sequential control flow edge between nodes n and m and $n \overset{pf}{\rightharpoonup} m$ for a parallel flow edge. Until further notice, $n \rightharpoonup^+ m$ means that there is a non empty path from n to m via control flow ($\overset{cf}{\rightharpoonup}$) and parallel flow ($\overset{pf}{\rightharpoonup}$) edges.

Example 5.2: Figure 5.2 on the next page shows the tCFG for the example program of Figure 5.1.

Definition 5.1 (Threads and Concurrent Execution)

1. $\theta(n)$ is a function that returns its innermost enclosing thread for every node n. In the example we have $\theta(2) = \theta_0$, $\theta(4) = \theta_1$ and $\theta(6) = \theta_2$. This function is statically decidable and can be generated during parsing and constructing the tCFG.

2. $\Pi(t)$ is a function that returns for every thread t the set of threads that can potentially execute concurrently with t, e. g. $\Pi(\theta_1) = \{\theta_2\}$ or $\Pi(\theta_0) = \emptyset$.

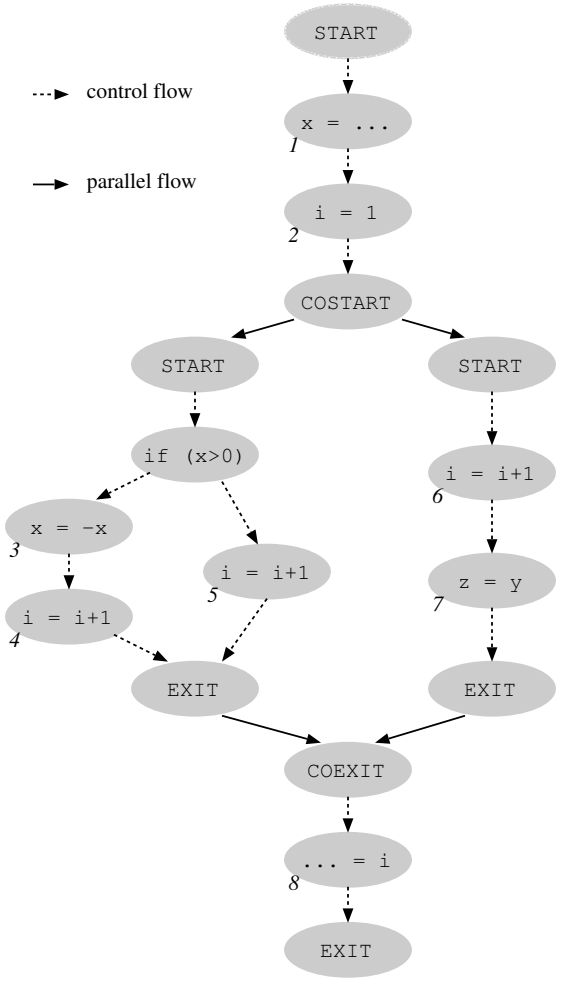


Figure 5.2: A threaded CFG

As the concurrent execution relation is symmetric, $t' \in \Pi(t) \iff t \in \Pi(t')$ holds. Note that the relation is neither reflexive nor transitive.

For concurrent programs without synchronization, Π can be computed easily. In presence of synchronization it is undecidable, but good approximations can be computed efficiently [MH89, NA98, NAC99].

The definition of witnesses in CFGs may also be applied to tCFGs. But this does not take the possible interleaving of nodes into account and we have to extend the definition:

Definition 5.2 (Threaded Witness)
A sequence $l = \langle n_1, \ldots, n_k \rangle$ of nodes is a *threaded witness* in a tCFG, iff

$$\forall 1 \leqslant i \leqslant k : \forall 1 \leqslant j < i : \theta(n_j) \in \Pi(\theta(n_i)) \vee n_j \overset{cf,pf}{\rightharpoonup}{}^{+} n_i$$

Basically this means that all nodes in a thread must be reachable from its predecessors if they cannot execute in parallel.[1]

Intuitively, a threaded witness can be interpreted as a witness in the sequentialized CFG. Every ordinary (not threaded) witness in the tCFG is automatically a threaded witness.

Example 5.3: In Figure 5.2 on the page before, $\langle 1,4,6 \rangle$ and $\langle 1,2,8 \rangle$ are threaded witnesses but $\langle 5,6,4 \rangle$ or $\langle 1,4,5 \rangle$ are not. The sequence $\langle 1,2,8 \rangle$ is also an ordinary witness, the sequence $\langle 1,4,6 \rangle$ is not.

Having a threaded witness $l = \langle n_1, \ldots, n_k \rangle$ and a node n, it can be decided whether $l' = \langle n_1, \ldots, n_k, n \rangle$ is a threaded witness without checking the threaded witness properties of l':

Theorem 5.1 (Appending/Prepending to a Threaded Witness)
Let the sequence $l = \langle n_1, \ldots, n_k \rangle$ be a threaded witness.

1. **Appending:** $l' = \langle n_1, \ldots, n_k, n \rangle$ is a threaded witness, iff

$$\forall 1 \leqslant i \leqslant k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^{+} n$$

2. **Prepending:** $l' = \langle n, n_1, \ldots, n_k \rangle$ is a threaded witness, iff

$$\forall 1 \leqslant i \leqslant k : \theta(n_i) \in \Pi(\theta(n)) \vee n \rightharpoonup^{+} n_i$$

Proof 5.1
Both variants follow directly from definition 5.2. □

[1] This definition of a threaded witness is more precise than the previous definition in [Kri98]: with the previous definition, $\langle 5,8,6 \rangle$ was a threaded witness.

Definition 5.3
Let sequence $l = \langle n_1, \ldots, n_k \rangle$ be a threaded witness, $t \in \Theta$. $L(l,t)$ and $F(l,t)$ are defined as follows:

$$L(l,t) = \begin{cases} n_i & \exists i : \theta(n_i) \notin \Pi(t) \wedge \forall i < j \leqslant k : \theta(n_j) \in \Pi(t) \\ \bot & \text{otherwise} \end{cases} \tag{5.1}$$

$$F(l,t) = \begin{cases} n_i & \exists i : \theta(n_i) \notin \Pi(t) \wedge \forall 1 \leqslant j < i : \theta(n_j) \in \Pi(t) \\ \bot & \text{otherwise} \end{cases} \tag{5.2}$$

The result is basically the last (or first) node of l relevant for the execution of thread t (if such a node exists).

Theorem 5.2 (Simplified Appending/Prepending to a Threaded Witness)
Let $l = \langle n_1, \ldots, n_k \rangle$ be a threaded witness.

1. $l' = \langle n_1, \ldots, n_k, n \rangle$ is a threaded witness, iff
$$L(l, \theta(n)) = \bot \vee L(l, \theta(n)) \rightharpoonup^+ n$$
2. $l' = \langle n, n_1, \ldots, n_k \rangle$ is a threaded witness, iff
$$F(l, \theta(n)) = \bot \vee n \rightharpoonup^+ F(l, \theta(n))$$

Proof 5.2
Only the "forward" direction (appending) is proved by contradiction. Basically the proof for the "backward" direction (prepending) is the same. Assume the opposite:

$$\begin{gathered} L(l, \theta(n)) \neq \bot \wedge L(l, \theta(n)) \not\rightharpoonup^\star n \\ \iff \exists 1 \leqslant i \leqslant k : n_i = L(l, \theta(n)) \wedge n_i \not\rightharpoonup^\star n \end{gathered}$$

From definition 5.3 follows $\theta(n_i) \notin \Pi(\theta(n))$. Altogether:

$$\exists 1 \leqslant i \leqslant k : \theta(n_i) \notin \Pi(\theta(n)) \wedge n_i \not\rightharpoonup^\star n$$

However, this is a contradiction to theorem 5.1 because l and l' are threaded witnesses and therefore theorem 5.2 must hold. □

Having a threaded witness $l = \langle n_1, \ldots, n_k \rangle$ and an edge $n_k \rightharpoonup n$, can it be decided if $l' = \langle n_1, \ldots, n_k, n \rangle$ is a threaded witness without checking the threaded witness properties of l'?

Theorem 5.3
Let $l = \langle n_1, \ldots, n_k \rangle$ be a threaded witness.

1. If an edge $n_k \rightharpoonup n$ exists, then $l' = \langle n_1, \ldots, n_k, n \rangle$ is a threaded witness.
2. If an edge $n \rightharpoonup n_1$ exists, then $l' = \langle n, n_1, \ldots, n_k \rangle$ is a threaded witness.

Proof 5.3
Again, only the "forward" direction is proved.

There exist three possibilities for $n_k \rightharpoonup n$: traditional control flow edges and parallel flow edges starting and ending threads.

1. From $n_k \overset{cf}{\rightharpoonup} n$ follows $\theta(n_k) = \theta(n)$ and with theorem 5.1:

$$\begin{array}{rl} & (\forall 1 \leqslant i < k : \theta(n_i) \in \Pi(\theta(n_k)) \vee n_i \rightharpoonup^+ n_k) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & (\forall 1 \leqslant i < k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^+ n_k) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & (\forall 1 \leqslant i < k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^+ n) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & \forall 1 \leqslant i \leqslant k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^+ n \end{array}$$

   which is theorem 5.1 itself.

2. Assume $n_k \overset{pf}{\rightharpoonup} n$:

   (a) n_k is a START node, therefore $\Pi(\theta(n_k)) \subset \Pi(\theta(n))$ and with theorem 5.1:

$$\begin{array}{rl} & (\forall 1 \leqslant i < k : \theta(n_i) \in \Pi(\theta(n_k)) \vee n_i \rightharpoonup^+ n_k) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & (\forall 1 \leqslant i < k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^+ n_k) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & (\forall 1 \leqslant i < k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^+ n) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & \forall 1 \leqslant i \leqslant k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^+ n \end{array}$$

   which is theorem 5.1.

   (b) n is a COEXIT node, therefore $\Pi(\theta(n_k)) \supset \Pi(\theta(n))$. Let $T = \Pi(\theta(n_k)) - \Pi(\theta(n))$. T can only contain the other threads that end at n_k and all nodes contained in these threads must reach n_k:

$$\begin{array}{rl} & (\forall t \in T : \forall n', \theta(n') \in \Pi(t) : n' \rightharpoonup^+ n_k) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & \forall t \in T : \forall n', \theta(n') \in \Pi(t) : n' \rightharpoonup^+ n \\ \Longrightarrow & \forall t \in \Pi(\theta(n_k)) \wedge t \notin \Pi(\theta(n)) : \forall n', \theta(n') \in \Pi(t) : n' \rightharpoonup^+ n \\ \Longrightarrow & \forall 1 \leqslant i < k : (\theta(n_i) \in \Pi(\theta(n_k)) \wedge \theta(n_i) \notin \Pi(\theta(n))) \longrightarrow n_i \rightharpoonup^+ n \end{array}$$

   From theorem 5.1 follows:

$$\begin{array}{rl} & (\forall 1 \leqslant i < k : \theta(n_i) \in \Pi(\theta(n_k)) \vee n_i \rightharpoonup^+ n_k) \wedge n_k \rightharpoonup^+ n \\ \Longrightarrow & \forall 1 \leqslant i \leqslant k : \theta(n_i) \in \Pi(\theta(n_k)) \vee n_i \rightharpoonup^+ n \end{array}$$

   with the previous result:

$$\Longrightarrow \quad \forall 1 \leqslant i \leqslant k : \theta(n_i) \in \Pi(\theta(n)) \vee n_i \rightharpoonup^+ n$$

   which is theorem 5.1 again.

□

Theorem 5.2 and 5.3 will later be used for an efficient algorithm that does not have to check all paths for the threaded witness property.

5.2 The Threaded PDG

As threaded programs have a special representation in the control flow graph, they also need special representation in the program dependence graph to enable precise slicing. Threading not only influences control and data dependence, but also necessitates the special treatment of *interference*, when a variable is accessed by threads running in parallel.

5.2.1 Control Dependence

The extension of the CFG to the tCFG influences both domination and control dependence. Post-dominance is defined based on paths in the CFG and must be adapted to tCFGs. Based on intuition, the following observations can be made:

- A COEXIT node n post-dominates its COSTART node and all nodes of the threads ending at n.
- A node n belonging to a thread and post-dominating the thread's START node also post-dominates the thread's COSTART node.

However, it follows that nodes belonging to a thread cannot be control dependent on the thread's START or COSTART node.

Example 5.4: In figure 5.2 on page 53, nodes 1, 2, 6 and 7 would all be control dependent just on the START node. More intuitive would be that node 6 and 7 are control dependent on the thread's START node, which itself is control dependent on the COSTART node.

Therefore, post-dominance in the tCFG is defined by viewing parallel flow edges as standard control flow edges, which makes the START nodes control dependent on their COSTART node. Also, "irrelevant" control flow edges are inserted between START and EXIT of threads, just like the control flow edge between the START and EXIT node of traditional control flow graphs.

Example 5.5: Figure 5.3 on the next page shows the prepared tCFG of figure 5.2 on page 53.

5.2.2 Data Dependence

Data dependence in traditional CFGs is based on reaching definitions. However, this is inadequate for tCFGs, because reaching definitions include definitions of concurrently executing threads. For example, in figure 5.2 on page 53 not only the definitions of statements 1 and 2 reach statement 6 but also all definitions of the other thread at 3, 4 and 5. For slicing purposes it is desirable to separate reaching definitions from concurrent threads, which makes the data dependence in non-concurrent threads computable by almost standard techniques for sequential programs. Data dependence between concurrent threads will be discussed in the following section and the rest of this section focuses on data dependence between non-concurrent threads.

Data dependence in non-concurrent threads can be divided into two classes:

1. Data dependence from a definition at a node n that reaches a node m in the *same* thread ($\theta(n) = \theta(m)$).
2. Data dependence from a definition at a node n that reaches a node m in a *different* thread ($\theta(n) \neq \theta(m)$).

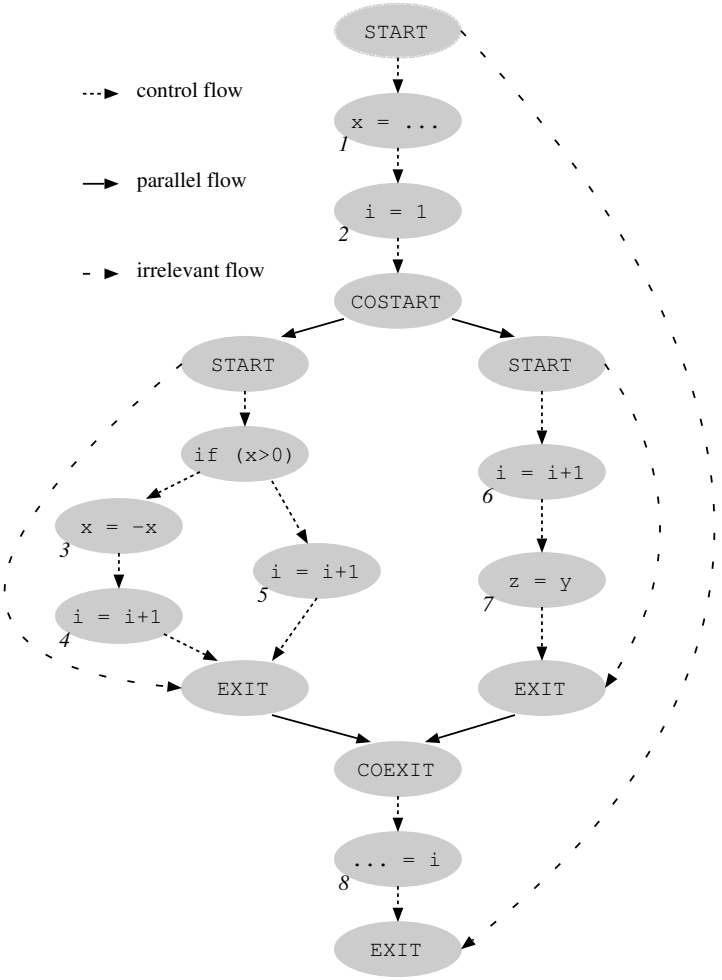


Figure 5.3: A tCFG prepared for control dependence

Even when we ignore data dependence between concurrent threads, the non concurrent threads are relevant because of possible *killing* definitions.

Example 5.6: Consider again the program fragment of figure 5.1 on page 52: Assume that the definition of x at statement 1 is a definition of z. This definition never reaches statement 8, as it *must* be killed at statement 7.

A definition reaching a COSTART node cannot reach the corresponding COEXIT node if it *must* be killed by any of the threads between these two nodes. Traditional data flow frameworks [Kil73, KU77] cannot handle this and specialized frameworks have been developed [GS93, KSV96]. These frameworks rely on the special handling of *killing* [GS93] or *destruction* [KSV96]. The usage of such a framework is assumed in the following. If a traditional framework is used instead, the result is still correct but imprecise because the killing information is not sharp.

5.2.3 Interference Dependence

When a variable is defined in one thread and referenced in another concurrently executing thread, *interference* occurs, which must be explicitly represented.

Example 5.7: In Figure 5.1 on page 52 we have an interference for the variable i between θ_1 and θ_2. The value of i at statement 6 may be the value computed at 2, 4 or 5. The value of i at statement 8 may be the value computed at 4, 5 or 6. However, if the statements 4, 5 and 6 are properly synchronized, the value of i will always be 3.

Definition 5.4
A node m is called *interference dependent* on node n, if

1. there is a variable v, such that $v \in \mathsf{def}(n)$ and $v \in \mathsf{ref}(m)$, and
2. $\theta(n) \in \Pi(\theta(m))$ (n and m may potentially be executed in parallel).

The dependences introduced by interference cannot be handled with normal data dependence because normal dependence is transitive but interference dependence is not: The transitivity of the data and control dependence results from their definitions, where a sequential path between the dependent nodes is required. The composition of paths in the CFG always results in a path again. Interference dependence is not transitive: If a statement x is interference dependent on a statement y, which is interference dependent on z, then x is only dependent on z iff there is a possible execution where these three statements are executed one after another: The sequence $\langle x, y, z\rangle$ of the three statements has to be a threaded witness in the tCFG.

Example 5.8: In Figure 5.4 on the following page statement 4 is interference dependent on statement 6, which in turn is interference dependent on statement 5. However, there is no possible execution where 4 is executed after 5 and

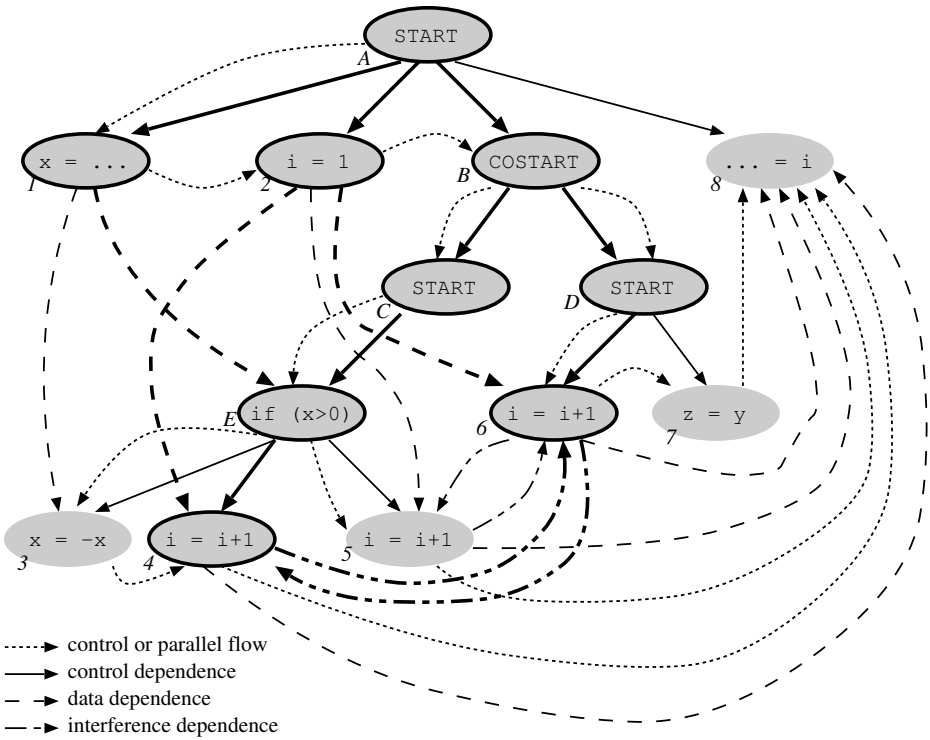


Figure 5.4: A threaded PDG

thus 4 cannot be interference dependent on 5. Thus, $\langle 5, 6, 4\rangle$ is not a threaded witness.

A simple version would implement Π according to [MH89] or [NA98, NAC99]. An interference dependence edge $n \overset{\text{id}}{\dashrightarrow} m$ will be inserted for all (n, m) if there is a variable v that is defined at n, referenced at m and $\theta(n) \in \Pi(\theta(m))$ holds.

5.2.4 Threaded Program Dependence Graph

A *threaded program dependence graph* (tPDG) consists of the nodes and the edges of the tCFG with the addition of control, data and interference dependence edges. In contrast to the standard PDG, where the control flow edges have been removed, the control and parallel flow edges will be needed for reasons that are explained later. As usual, the EXIT and COEXIT nodes can be removed provided the control and parallel flow edges are adapted accordingly.

Example 5.9: The tPDG for the running example is shown in Figure 5.4.

```
1     i = 1;
      cobegin {
        while (z>0) {
          cobegin {
2           x = i;
          }{
3           y = x;
          } coend;
        }
      }{
4       z = y;
      } coend;
5     x = z;
```

Figure 5.5: A program with nested threads

More complicated structures like loops or nested threads may be handled in the same way. If threads are embedded in loops, multiple instances of the same thread may exist. Such a thread may both execute concurrently and non-concurrently with respect to a different thread.

Example 5.10: An example is shown in Figure 5.5. In the tPDG in Figure 5.6 on the next page there is both a data and an interference dependence edge between statement 2 and 3. Both statements and their threads may be executed concurrently (thus the interference dependence). The statements and their threads may also be executed sequentially through different iterations of the enclosing loop.

5.3 Slicing the tPDG

Slicing on the PDG of sequential programs is a simple graph reachability problem because control and data dependence are transitive. As interference dependence is not transitive, this definition of a slice for PDGs is not valid for tPDGs and hence the standard algorithms are not adequate.[2]

The basic idea of our approach stems from a simple observation: Because every path in the PDG is a witness in the corresponding CFG, every node n that is reachable from a node m in the PDG is also reachable from m in the corresponding CFG. This does not hold for the threaded variants. The definition of a slice in the tPDG establishes a similar property because it demands that the tPDG contains a threaded witness between every node in the slice and the slicing criterion.

[2]The "classical" definition of a slice is any subset of a program that does not change the behavior with respect to the criterion: a program is a correct slice of itself. Therefore, if interference is modeled with normal data dependence, the resulting slices are correct but imprecise.

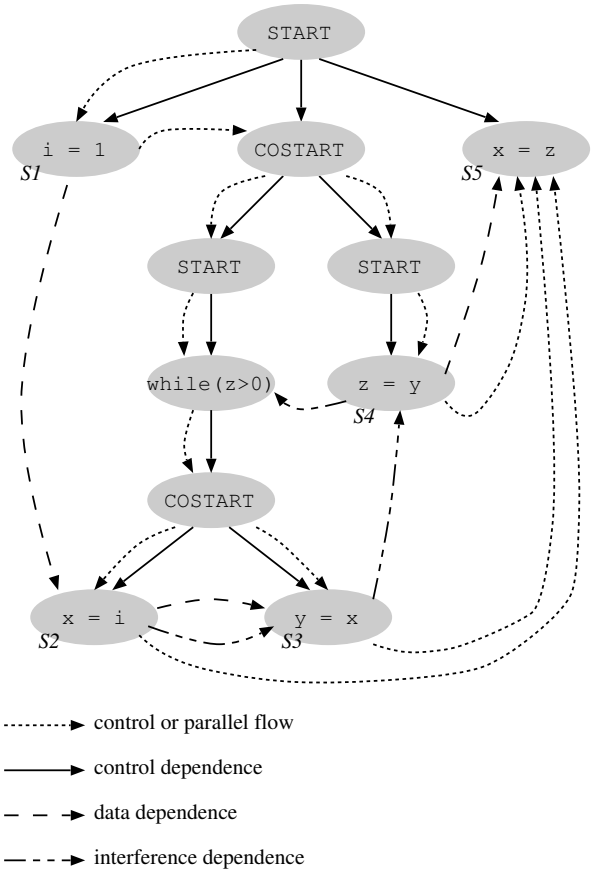


Figure 5.6: The tPDG of Figure 5.5

Definition 5.5 (Threaded Realizable Paths)
A path $P = \langle n_1, \ldots, n_k \rangle$ in a tPDG is called a *threaded realizable path*, iff

1. the path contains no control flow edge:

$$n_1 \xrightarrow{d_1} \ldots \xrightarrow{d_{k-1}} n_k \wedge \forall_{1 \leqslant i < k} d_i \neq cf$$

   and

2. a threaded witness W exists corresponding to the path P: $W = P$.

If such a path exists, n_1 is said to reach n_k via a threaded realizable path: $n_1 \rightharpoonup_R^\star n_k$.

Definition 5.6
The (*backward*) *slice* $S_\theta(n)$ of a tPDG at a node n consists of all nodes m on which n transitively depends via a threaded realizable path:

$$S_\theta(n) = \{m \mid m \rightharpoonup_R^\star n\}$$

Example 5.11: A slice from the statement 4 of the example program in Figure 5.1 on page 52 is shown in Figure 5.4 on page 60 as framed nodes. The responsible edges are drawn in a thicker style. Note that there are interference edges between statement 6 and 5, which do not force the inclusion of statement 5 into the slice because 4 is not reachable from 5 in the tCFG. The standard slicing algorithm would include statement 5 into the slice, which is inaccurate, albeit correct.

The algorithm to slice sequential programs is a simple reachability algorithm. However, it is not easy to transform the definition of a threaded slice into an algorithm because the calculation of threaded witnesses would be too costly. Therefore we present a different slicing algorithm 5.1 on the following page based on theorem 5.2 and 5.3. Its basic idea is the coding of possible execution states of all threads in tuples $(t_0, t_1, \ldots, t_{|\Theta|-1})$, where the t_i are nodes in the tPDG. The value t_i represents the fact that it is still possible to reach node t_i in thread θ_i (a value of $\perp$ does not restrict the state of execution). This is used to keep track of the node n where a thread has been left by following an interference edge. If another interference edge is followed back into the thread at node m, the reachability of n from m can be checked, which assures that paths over interference edges are always threaded witnesses in the tCFG. This is the reason why the control and parallel flow edges have to be kept in the tPDG.

We denote the extraction of the ith element t_i for thread θ_i in a tuple $T = (t_0, t_1, \ldots, t_n)$ with $T[\theta_i]$. The substitution of the ith element t_i for thread θ_i in a tuple $T = (t_0, t_1, \ldots, t_n)$ with a value x will be denoted as $T[\theta_i] = x$.

The algorithm keeps a worklist of pairs of nodes and state tuples that have to be examined. Every edge reaching the current node is examined and handled based on its type. In case of a control or data dependence edge, a new pair consisting of the source node and the modified state tuple is inserted into the worklist. According to theorem 5.3 this is done without checking the threaded witness property. In the other case it is an interference dependence edge. It may only be considered if the state node relevant to the source node thread is reachable from the source node in the tCFG (all examined paths are still threaded witnesses). Then, the new pair with the updated state tuple is inserted into the worklist. The resulting slice is the set of nodes constructed from the first elements of the inserted pairs.

Example 5.12: In the following an application of the algorithm to calculate a backward slice for node 4 is demonstrated. The worklist w is initialized with the element $(4, (4, 4, \perp))$. This element is removed immediately from the worklist and all edges reaching 4 are examined. The edge $E \xrightarrow{cd} 4$ does not cross

Algorithm 5.1 Slicing in tPDGs

Input: The slicing criterion s, a node of the tPDG
Output: The slice S, a set of nodes of the tPDG

Initialize the worklist with an initial state tuple:
$T = (t_0, \ldots, t_{|\Theta|}), t_i = \begin{cases} s & \theta(s) \notin \Pi(\theta_i) \\ \bot & \text{otherwise} \end{cases}$
worklist $w = \{s, T\}$
slice $S = \{s\}$
repeat
  remove the next element $c = (m, T)$ from w
  Examine all reaching edges:
  foreach edge $e = n \xrightarrow{cd,dd} m$ **do**
    Concatenation of e *results in a threaded witness (theorem 5.3)*
    Update T *for definition 5.3:*
    foreach $t \notin \Pi(\theta(n))$:
      $T[t] = n$
    $c' = (n, T)$
    if c' has not been already calculated **then**
      mark c' as calculated
      $w = w \cup \{c'\}$
      $S = S \cup \{n\}$
  foreach edge $e = n \xrightarrow{id} m$ **do**
    $n' = T[\theta(n)]$
    if $n' = \bot$ **or** $n \xrightarrow{cf,pf}{}^{+} n'$ **then**
      Concatenation of e *results in a threaded witness (theorem 5.2)*
      Update T *for definition 5.3:*
      foreach $t \notin \Pi(\theta(n))$ **do**
        $T[t] = n$
        $c' = (n, T)$
      if c' has not been already calculated **then**
        mark c' as calculated
        $w = w \cup \{c'\}$
        $S = S \cup \{n\}$
until worklist w is empty
return S

threads and the state of the thread $\theta(4) = \theta(E)$ is updated before the created element $(E, (E, E, \bot))$ is inserted into the worklist. The edge $2 \xrightarrow{dd} 4$ crosses threads and creates a new element $(2, (2, 2, 2))$. The edge $6 \xrightarrow{id} 4$ creates $(6, (6, 4, 6))$, because the state of $\theta(6)$ is $\bot$. Let us step forward in the calculation and assume the worklist is $\{(6, (6, 4, 6)), (C, (C, C, \bot)), \ldots\}$. There are four edges reaching 6:

1. $2 \xrightarrow{dd} 6$ crosses threads and creates the element $(2, (2, 2, 2))$. As this element has already been visited it is not inserted into the worklist again.
2. $D \xrightarrow{cd} 6$ does not cross threads and inserts the element $(D, (D, 4, D))$ into the worklist.
3. $5 \xrightarrow{id} 6$: as $(6, 4, 6)[\theta(5)] = 4$ and the condition $5 \xrightarrow{cf,pf}{}^{+} 4$ is not fulfilled, this edge has to be ignored.
4. $4 \xrightarrow{id} 6$: $4 \xrightarrow{cf,pf}{}^{+} 4$ cannot be fulfilled and the edge has to be ignored.

In the third step, the edge has to be ignored because it would destroy the property that every node in the slice is part of a threaded witness. The condition that is not satisfiable in step four may be relaxed if we drop our assumption that the program is properly synchronized on statement level. The remaining calculations are presented in Figure 5.7 on the next page.

If we assume that the analyzed program has no threads, $\Theta = \{\theta_0\}$, then this algorithm is similar to the sequential slicing algorithm. In that case, the second iteration over all interference dependence edges will not be executed and the worklist will only contain tuples of the form $(n, (n))$, where n is a node of the PDG. Hence the standard slicing algorithm on PDGs is a special case of our algorithm, with the same time and space complexity for the non-threaded case.

In the threaded case the reachability $n \xrightarrow{cf,pf}{}^{+} m$ has to be calculated iteratively. This determines the worst case for time complexity in the number of interference edges: the traversal of these edges might force another visit of all nodes that may reach the source of the edge. Therefore, the worst case is exponential in the number of interference dependence edges. The number of interference edges is potentially quadratic, however, we believe that the number of interference dependence edges will be very small in every program, as interference is error prone, hard to understand and to debug. The required calculation time will be much less than the time required to analyze serialized programs.

5.4 Extensions

For simplicity, additional features of concurrent programs like synchronization have been ignored so far. Most models of concurrent execution include some methods of synchronization. Two such methods are synchronized blocks and send/receive communication, which will be discussed in the following.

$w : \{(4,(4,4,\bot))\}$
$E \overset{cd}{\multimap} 4 \Rightarrow (E,(E,E,\bot))$
$2 \overset{dd}{\multimap} 4 \Rightarrow (2,(2,2,2))$
$6 \overset{id}{\multimap} 4 \Rightarrow (6,(6,4,6))$

$w : \{(E,(E,E,\bot)),(2,(2,2,2)),(6,(6,4,6))\}$
$C \overset{cd}{\multimap} E \Rightarrow (C,(C,C,\bot))$
$1 \overset{dd}{\multimap} E \Rightarrow (1,(1,1,1))$

$w : \{(2,(2,2,2)),(6,(6,4,6)),(C,(C,C,\bot)),(1,(1,1,1))\}$
$A \overset{cd}{\multimap} 2 \Rightarrow (A,(A,A,A))$

$w : \{(6,(6,4,6)),(C,(C,C,\bot)),(1,(1,1,1)),(A,(A,A,A))\}$
$2 \overset{dd}{\multimap} 6 \Rightarrow (2,(2,2,2))$ already visited
$D \overset{cd}{\multimap} 6 \Rightarrow (D,(D,4,D))$
$5 \overset{id}{\multimap} 6 \Rightarrow 5 \overset{cf,pf}{\multimap}{}^{+} 4$ is not fulfilled $(T[\theta(5)] = 4)$
$4 \overset{id}{\multimap} 6 \Rightarrow 4 \overset{cf,pf}{\multimap}{}^{+} 4$ is not fulfilled $(T[\theta(4)] = 4)$

$w : \{(C,(C,C,\bot)),(1,(1,1,1)),(A,(A,A,A)),(D,(D,4,D))\}$
$B \overset{cd}{\multimap} C \Rightarrow (B,(B,B,B))$

$w : \{(1,(1,1,1)),(A,(A,A,A)),(D,(D,4,D)),(B,(B,B,B))\}$
$A \overset{cd}{\multimap} 1 \Rightarrow (A,(A,A,A))$ already in worklist

$w : \{(A,(A,A,A)),(D,(D,4,D)),(B,(B,B,B))\}$
no edge reaching A exists

$w : \{(D,(D,4,D)),(B,(B,B,B))\}$
$B \overset{cd}{\multimap} D \Rightarrow (B,(B,B,B))$ already in worklist

$w : \{(B,(B,B,B))\}$
$A \overset{cd}{\multimap} B \Rightarrow (A,(A,A,A))$ already visited

$$\Rightarrow \quad S_\theta(4) = \{4,E,2,6,C,1,A,D,B\}$$

Figure 5.7: Calculation of $S_\theta(4)$

5.4.1 Synchronized Blocks

Synchronized blocks are blocks of statements that are executed atomically: Interference cannot arise inside such a block. An example for an instance are monitors:

Example 5.13:

```
     ...
5    synchronized {
6      x = y;
7      if (z > 0)
8        x = x + z;
9    }
     ...
```

In this example, interference cannot happen to the variables x, y and z while the synchronized block is executing: The usage of x in line 8 can only be data dependent on line 6. However, both definitions in line 6 and line 8 can interfere with any usage or definition of the same variable in other threads.

One possibility is to ignore the synchronization statement and treat synchronized blocks as normal blocks. This is a conservative approximation and will only add unrealizable interference. The precise solution is to compute the set of definitions that reach the end of the synchronized block and the set of usages that reach the entry.

Definition 5.7
A node m is *interference dependent* on node n, if

1. there is a variable v, such that $v \in \mathsf{def}(n)$ and $v \in \mathsf{ref}(m)$,
2. n is not embedded in a synchronized block or the definition at n reaches the exit of the synchronized block,
3. m is not embedded in a synchronized block or the usage at m reaches the entry of the synchronized block, and
4. $\theta(n) \in \Pi(\theta(m))$ (n and m may potentially be executed in parallel) or the synchronized blocks of n and m would potentially execute in parallel without synchronization.

5.4.2 Communication via Send/Receive

If two threads exchange information via send and receive communication, the execution of the receiving thread may block until the sending thread has sent some information. This has three effects on control and data dependence:

1. The exchange of information creates a data dependence between the sending and receiving statement. To distinguish it from normal data dependence, such dependence may be called *communication* dependence. In order to omit time travel, communication dependence must be treated like interference dependence during slicing.

2. Because the execution at a receive may be blocked until some other thread sends some data, the computation of Π becomes more complex. However, a conservative approximation is to ignore such blocking, as the Π function will still return an (imprecise) superset of the realizable relations.

3. A receiving statement that may block (together with its successors) can be seen as control dependent on the sending statement. This control dependence can be computed simply by inserting the communication dependence into the control flow graph and treat it as a control flow edge.

Example 5.14: Consider the example of figure 5.3 on page 58 again and assume that statements 4 and 5 are send statements that send information to statement 6, the latter is assumed to be a receive statement. Therefore the control flow graph in figure 5.8 on the next page contains two communication dependence edges. If control dependence is computed with this extended control flow graph, statements 6 and 7 will be control dependent on statement 4 and 5, which can be interpreted as statements 6 and 7 will only be executed if one of the statements 4 or 5 has executed. The control dependence graph for this control flow graph is shown in figure 5.9 on page 70.

5.5 Related Work

An earlier, less precise version of the presented work has been published in [Kri98].

There are many variations of the program dependence graph for threaded programs like parallel program graphs [SS93, Che97, Che93, DGS92, CXZY02]. Most approaches to static or dynamic slicing of threaded programs are based on such dependence graphs.

Dynamic slicing of threaded or *concurrent* programs has been approached by different authors [MC88, CMN91, DGS92, KF92, KK95, GM00] and is surveyed in [Tip95]. Probably the first approach for *static* slicing of threaded programs was the work of Cheng [Che93, ZCU96, Che97]. He introduced some dependences, which are needed for a variant of the PDG, the *program dependence net* (PDN). His *selection* dependence is a special kind of control dependence and his *synchronization* dependence is basically control dependence resulting from the previously presented communication dependence. Cheng's *communication dependence* is a combination of data dependence and the presented communication dependence. Although the tPDG is not mappable to his PDN and vice

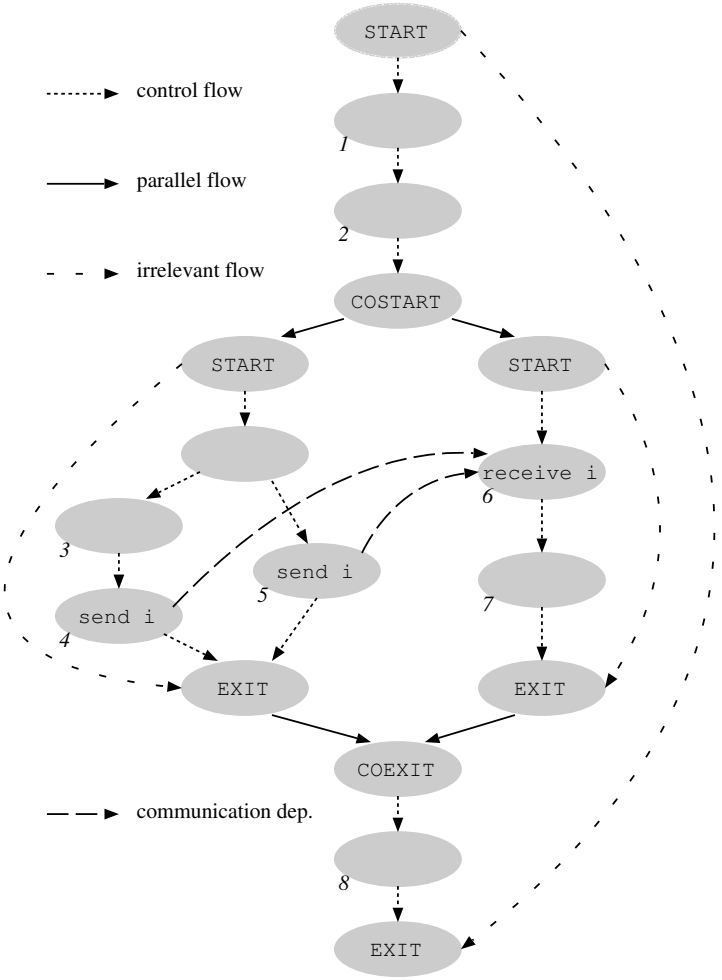


Figure 5.8: A tCFG with communication dependence

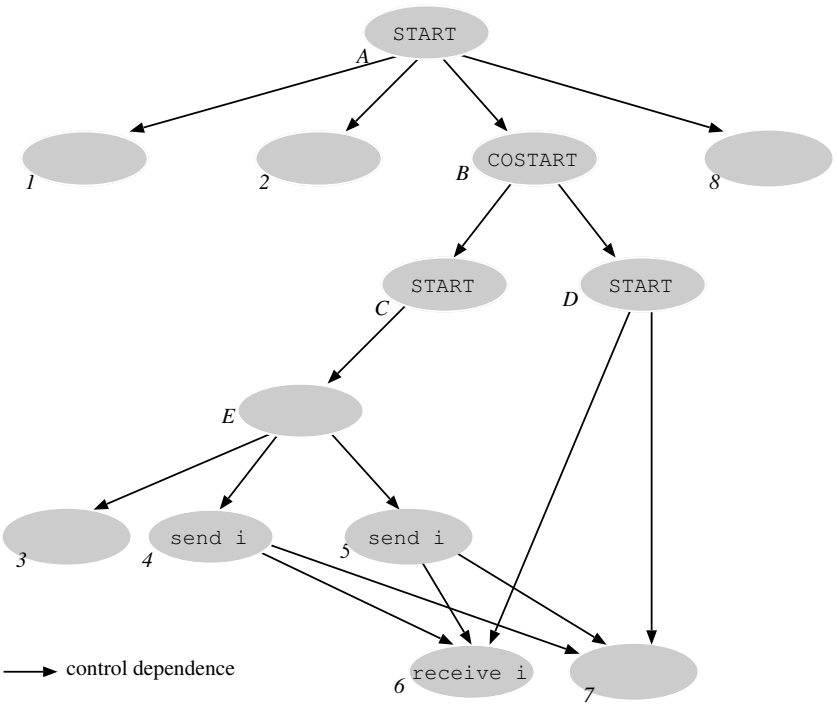


Figure 5.9: Control dependence for tCFG from figure 5.8

versa, both graphs are similar in the number of nodes and edges. Cheng defines slices simply based on graph reachability. The resulting slices are not precise, as they do not take into account that dependences between concurrently executed statements are not transitive. Therefore, the integration of his technique of slicing threaded programs into slicing threaded object oriented programs [ZCU96, ZCU98, Zha99a, Zha99b] has the same problem. After the first publication of [Kri98] more work on precise static slicing of concurrent programs has been done: [NR00] improves the earlier version of our work in [Kri98]. Their improvements are not needed in this chapter's approach, where cleaner definition avoid the earlier drawbacks. [CX01a] is a different approach but is also based on dependence graphs.

There is a series of works that use static slicing of concurrent programs but treat interference transitively and accept the imprecision: [HCD+99, DCH+99] present the semantics of a simple multi-threaded language that contains synchronization statements similar to the JVM. For this language, they introduce and define additional types of dependence: divergence dependence, synchronization dependence and ready dependence. [MT98, MT00] applies Cheng's approach to slice Promela for model checking purposes.

Data flow analysis frameworks exist also for concurrent programs: [KSV96] uses essentially the same `cobegin/coend` model of concurrency and its parallel flow graph is similar to the tCFG. To represent valid node sequences the restriction of G^**-well-formedness* is used, which is similar to the threaded witness property. Seidl [SS00] presents a framework for the problems of strong copy constant propagation and (ordinary) liveness of variables in concurrent programs and proves that these problems have the same complexity in both sequential and concurrent cases. Slicing is a harder problem than reaching definitions. Proofs for lower bounds can be found in [MOS01, Ram00]. Both show that precise slicing is undecidable in the interprocedural case. Despite the undecidability results, chapter 8 will present a high-precision approximation for interprocedural slicing of concurrent programs.

Part II

Interprocedural Analysis

Chapter 6

Interprocedural Data Flow Analysis

The first part of this thesis presented the fundamentals of analyzing and slicing programs without procedures. Of course, real world programs without procedures don't exist. In the second part of this thesis the work of the first part is therefore extended to programs with procedures. Interprocedural analysis is not a simple extension to intraprocedural analysis—it will be shown that analysis in the right calling context is the main problem.

A main focus for this part are variants of *interprocedurally realizable paths*. For procedure-less programs, it was assumed that every path through the control flow and program dependence graph is realizable. Now paths are interprocedural realizable, if every called procedure returns to the call site it was called from. Interprocedural data flow analysis is usually defined in terms of interprocedurally realizable paths and the next chapter will define program slicing based on those paths.

This chapter will give a short introduction to interprocedural data flow analysis first by revisiting the problem of reaching definitions. The main section will discuss the problem of interprocedurally realizable paths, which are needed in the following section about data flow analysis. The last section will introduce the interprocedural version of the program dependence graph: the *interprocedural program dependence graph (IPDG)*.

6.1 Interprocedural Reaching Definitions

The problem of reaching definitions was clear and simple in the intraprocedural case. However, in the interprocedural case it is more complicated: a definition in one procedure might reach a statement in a different procedure by various means.

```
1  int a, b, c;
2
3  void q () {
4    int z = 1;
5    a = 2;
6    b = 3;
7    p(4, z);
8    z = a;
9    c = 5;
10   p(6, c);
11 }
```

```
12 void p (int x, int& y) {
13   static int d = 6;
14   a = c;
15   if (x) {
16     d = 7;
17     p(8, x);
18   } else {
19     b = 9;
20   }
21   y = 0;
22 }
```

Figure 6.1: Simple example for reaching definitions

Example 6.1: Consider the program in figure 6.1, which contains multiple definitions: `a`, `b`, `c` and `d` are global variables, `z` is a local variable, `x` a call-by-value parameter and `y` a call-by-reference parameter.

- The definition of global `a` in line 5 reaches lines 6–7, but not lines 8–11, as it is killed by line 14 through the call in line 7. It also reaches lines 13–14 but not 15–22. The definition of global `c` in line 9 reach lines 13–22 through the call in line 10.

- More complex are the definitions of global `b`: the definition in line 6 cannot reach lines 8–10 or 21, as line 19 kills it—any call of `p` must execute line 19 to terminate the recursion. Also, the definition in line 19 reaches line 13–19, as it might reach the call in line 10 by procedure `p` returning from the call in line 7.

- The variable `d` is global and only visible inside procedure `p`: the definition in line 16 may reach lines 13–16 because of the call in line 17. Through procedure `p` returning from the call in line 7, both definitions (line 13 and 16) may reach lines 8–10 and therefore also line 13–16 and 18–22.

- Locals like `z` are (usually) only visible in procedures they are defined in. Call-by-value parameters are like locals, with a definition at the procedure entry: `x` is defined in line 12.

- Call-by-reference introduces a simple form of aliasing and make otherwise invisible variables available in called procedures.

For the moment, call-by-reference and aliasing are ignored and only scalar local or global variables and call-by-value parameters will be discussed.

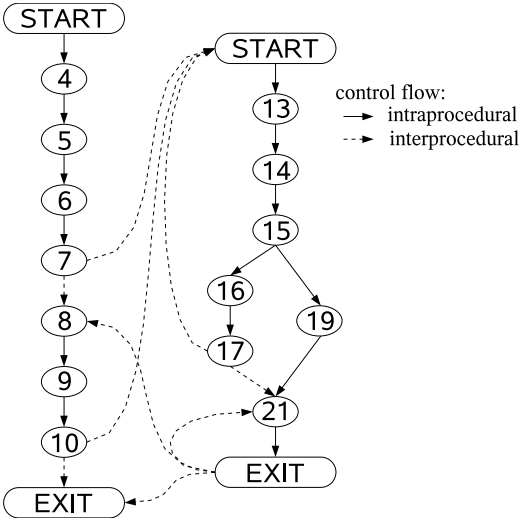


Figure 6.2: Interprocedural control flow graph

6.2 Interprocedural Realizable Paths

In the intraprocedural case all paths in the CFG were assumed to be executable and therefore realizable. In the interprocedural case this is more complicated:

The individual procedures of a program are represented in control flow graphs $G_p = (N_p, E_p, n_p^s, n_p^e)$ for each procedure p. An *interprocedural control flow graph* (ICFG) is a directed graph $G = (N^\star, E^\star, n_0^s, n_0^e)$, where $N^\star = \bigcup_p N_p$ and $E^\star = E^C \cup \bigcup_p E_p$. One procedure q is the program's main procedure, its START and EXIT nodes are the main START and EXIT nodes: $n_0^s = n_q^s$ and $n_0^e = n_q^e$. The calls are represented by *call* and *return edges* in E^C: A call edge $e \in E^C$ is going from a *call node* $n \in N_p$ to the START node n_q^s of the called procedure q. A return edge $e \in E^C$ is going from the EXIT node n_q^e of the called procedure q back to the immediate successor of the call node $n \in N_p$.[1]

Example 6.2: Figure 6.2 shows the ICFG for the reaching definition example. Note that there are control flow edges between call nodes and their immediate successors.

If any path through the ICFG is assumed to be a realizable path, data flow analysis will become imprecise, as clearly unrealizable paths can be traversed: Consider the definition of global `c` in line/node 9, which reaches the called procedure via the call edge at line/node 10. All paths through `p` are free of definitions for `c` and the definition gets propagated along the return edges: via

[1]There are two common variants: First, the immediate successor of a call node is an explicitly defined return node. Second, the return edge is going from the EXIT node to the call node itself.

the return edge for the call in line/node 8 the definition may reach line/node 9. However, this is clearly an *interprocedurally unrealizable* path, because the path does not return to the matching successor in line/node 10.

One way to describe interprocedurally realizable paths is via context-free language reachability: The intraprocedural control flow graph can be seen as a finite automaton and the intraprocedurally realizable paths are words of its accepted language. Therefore, reachability in the control flow graph is an instance of regular language reachability. The problem in interprocedural reachability is the proper matching of call edges to return edges. This can be achieved by defining a context-free language on top of the ICFG: Edges from E_p are marked with the empty word ϵ and edges from E^C are marked according to their source and target nodes:

- Call edges between a call node m and a START node n_p^s are marked with "$(_m^{n_p^s}$".
- Return edges between an EXIT node n_p^e and a (return) node n are marked with "$)_m^{n_p^s}$", where m is the predecessor of n (the corresponding call node m) and n_p^s is the START node of p.
- Edges between a call node n and its successor (the return node) are marked with $\bot$.

Let Σ be the set of all edge labels in an ICFG G. Every path in G induces a word over Σ by concatenating the labels of the edges on the path. A path is an interprocedural *matched* path if it is a word of the context-free language defined by:

$$\begin{array}{lcll} M & \rightarrow & MM & \\ & | & (_m^{n_p^s} M)_m^{n_p^s} & \forall (_m^{n_p^s} \in \Sigma \\ & | & \epsilon & \end{array}$$

This grammar assures the proper matching of calls and returns by simulating an abstract call stack. Notice the absence of $\bot$ from the grammar: this assures that paths must pass through called procedures. On the other hand, it is sometimes desirable to allow paths to bypass calls—in that case edges between call and return are labeled with the empty word ϵ instead.

Interprocedural matched paths require their start and end node to be in the same procedure. Interprocedurally realizable paths with start and end node in different procedures have only partially matching calls and returns: dependent if the end node is lower or higher in the abstract call stack, the paths are right-balanced or left-balanced. A path is an interprocedural *right-balanced* path if it is a word of the context free language defined by:

$$\begin{array}{lcll} R & \rightarrow & MR & \\ & | & (_m^{n_p^s} R & \forall (_m^{n_p^s} \in \Sigma \\ & | & \epsilon & \end{array}$$

Here, every $)_m^{n_p^s}$ is properly matched to a $(_m^{n_p^s}$ to the left, but the converse need not hold. A path is an interprocedural *left-balanced* path if it is a word of the context free language defined by:

$$\begin{array}{rcll} L & \rightarrow & LM & \\ & | & L)_m^{n_p^s} & \forall (_m^{n_p^s} \in \Sigma \\ & | & \epsilon & \end{array}$$

An *interprocedurally realizable path* is an interprocedurally right- or left-balanced path.

$$\begin{array}{rcl} I & \rightarrow & L \\ & | & R \end{array}$$

Definition 6.1 (Interprocedural Reachability)
A node n is *interprocedurally reachable* from node m, iff an interprocedurally realizable path from m to n in the ICFG exists, written as $m \rightharpoonup_R^* n$.

The concept of a witness (see 2.1 on page 15) must also be transfered to the interprocedural case:

Definition 6.2 (Interprocedural Witness)
A sequence $\langle n_1, \ldots, n_k \rangle$ of nodes is called an *interprocedurally (realizable) witness*, iff n_k is interprocedurally reachable from n_1 via an interprocedurally realizable path $p = \langle m_1, \ldots, m_l \rangle$ with:

1. $m_1 = n_1$, $m_l = n_k$, and
2. $\forall 1 \leqslant i < k : \exists x, y : x < y \wedge m_x = n_i \wedge m_y = n_{i+1}$.

6.3 Analyzing Interprocedural Programs

Ignoring parameters, the *interprocedural meet-over-all-paths (IMOP)* solution of a data flow problem can be defined just by using the definition of interprocedurally realizable paths. For example, the interprocedural reaching definition problem is:

$$RD_{IMOP}(n) = \bigcup_{p = \langle n_0^s, \ldots, n \rangle} [\![p]\!](\emptyset)$$

where all p are interprocedurally realizable paths. As in the intraprocedural case, the computation of the IMOP solution is impossible in general. Therefore only the *interprocedural minimal-fixed-point (IMFP)* solution is computed. However, complete paths are no longer analyzed, it is impossible to check for interprocedurally realizable paths and different approaches must be applied:

- Procedures can be *inlined*: calls get replaced by the called procedure and the resulting program can be analyzed like an intraprocedural one. However, this is not possible in the presence of recursion and even without, the size of the inlined programs may grow exponentially.

- The *effects* of procedures are computed first and represented in a transfer function that maps flow information at a call site from the call to the return. Thus the call statements are ordinary statements with transfer functions and intraprocedural techniques can be applied.
- The *calling context* of a procedure is encoded explicitly and the procedure is analyzed for each calling context separately. Again, in the presence of recursion the set of calling contexts may be infinite, depending on the encoding of the calling context.

The inlining approach can only be used for non realistic small programs without recursion. Therefore, only the two remaining approaches are relevant:

6.3.1 Effect Calculation

This approach, also called the *functional* approach [SP81], computes a function for every procedure that maps the data flow information at the entry of a procedure to the information that holds at the exit. The computed function can be used in the transfer functions at the call statements and intraprocedural data flow analysis can then be used in a second pass. The first pass is basically a data flow analysis where the data flow information are functions and the transfer functions are function compositions. For some data flow problems the resulting data flow information is infinite function compositions and therefore not computable. For a large class of data flow problems these computed functions reduce to simple mappings where the composition can be computed instantly.

6.3.2 Context Encoding

One way to encode the calling context is to capture the "history" of calls that lead to a node n in form of a *call string* [SP81]. This is basically an abstraction of the call stack of the machine executing the program. With this approach the lattice of the data flow facts is replaced by a lattice whose elements combine calling context and the former data flow facts. The transfer functions are extended to handle the additional calling context. In the presence of recursion the number of such call strings is infinite, which disables the use of this approach with recursive functions. To overcome this problem, the length of the call strings can be limited to a certain length k. Calling context that would have a call string longer than k are shortened such that the "oldest" elements are removed first (called *call-string suffix approximation* [SP81]). This approach may be imprecise if there exist calling contexts that have the same k "newest" elements in their k limited call strings: these calling contexts will not be distinguished in the data flow analysis.

The complexity of the (full) call-string approach is not different from the one using inlining: Let procedure p have the set C_p of possible call-strings. With inlining, p is replicated $|C_p|$ times. On the other hand with the (full) call-string approach, p is analyzed $|C_p|$ times.

The ideas of both approaches can be combined: The calling context $c \in \mathcal{C}$ is encoded through the data flow facts that hold at the entry to a procedure $p \in P$. From that the data flow facts c' that hold at the exit of the procedure are computed and stored in a mapping $\mathcal{C} \times P \to \mathcal{C}$. At every call node n of a procedure p the data flow facts c are then bound to data flow facts $c' = \mathsf{bind}(c)$ that hold at the entry node of p. If the effect of p for c' has already been computed, it can be reused from the mapping which contains the data flow facts c'' holding at the exit of p. After back-binding the effect to the call site, the effect $c''' = \mathsf{bind}^{-1}(c'')$ holds at the exit of the call node n.

6.4 The Interprocedural Program Dependence Graph

An interprocedural version of the program dependence graph can be constructed by just using the intraprocedural version with interprocedural data dependence computed from interprocedural reaching definitions. However, this has a disadvantage: A path in such a dependence graph no longer has a corresponding interprocedurally realizable path in the ICFG.

Example 6.3: In the earlier example (figure 6.1 on page 76) the definition of `c` at node 9 is an interprocedurally reaching definition at the use of `c` in node 14, which creates a data dependence between nodes 9 and 14. The definition of `a` at node 14 reaches the usage in node 8 and creates a data dependence between 14 and 8 (see figure 6.3 on the next page). The resulting path on both data dependence edges $9 \overset{\mathrm{dd}}{\multimap} 14 \overset{\mathrm{dd}}{\multimap} 8$ has no corresponding interprocedurally realizable path.

Data dependence based on interprocedurally realizable paths is called *interprocedural data dependence*:

Definition 6.3 (Interprocedural Data Dependence)
A node m is called *interprocedurally data dependent* on node n, if

1. there is an interprocedurally realizable path p from n to m in the ICFG ($p = n \rightharpoonup_R^* m$),
2. there is a variable v, with $v \in \mathsf{def}(n)$ and $v \in \mathsf{ref}(m)$, and
3. for all nodes $k \neq n$ of path p, $v \notin \mathsf{def}(k)$ holds.

As transitive interprocedural data dependence may result in unrealizable paths, it is not adequate for interprocedural slicing. Therefore, the interprocedural version of the PDG, the *interprocedural program dependence graph* (IPDG), models each procedure as a single PDG and connects these with special edges.

6.4.1 Control Dependence

Each procedure is assumed to be a single-entry-single-exit region. It is also assumed that all procedures terminate. Because of these assumptions, the execution of the successor of a call node is never controlled by a node in the called

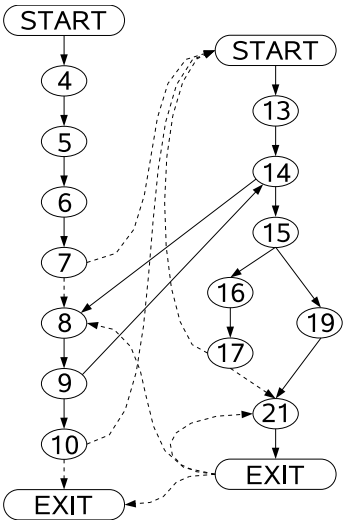


Figure 6.3: ICFG with data dependence

procedure. Therefore control dependence is computed only intraprocedural, where the edges between call nodes and their successors are assumed to be normal control flow edges.

6.4.2 Data Dependence

For representation in the PDGs of the procedures, data dependence is computed only intraprocedural:

Definition 6.4
Let $G = (N^\star, E^\star, n_0^s, n_0^e)$ be an ICFG. A node $m \in N^\star$ is *data dependent* on node $n \in N^\star$, if

1. there is an interprocedurally matched path p from n to m in the ICFG,
2. there is a variable v, with $v \in \mathsf{def}(n)$ and $v \in \mathsf{ref}(m)$, and
3. for all nodes $k \neq n$ of path p, $v \notin \mathsf{def}(k)$ holds.

The difference to interprocedural data dependence is the restriction on interprocedurally matched paths. This variant of data dependence can be computed with a slightly modified version of interprocedural reaching definitions RD_{IMFP}.

Without global variables (and call-by-reference and aliasing) the analysis would be even simpler, as called procedures would have no effects in the calling procedure and the intraprocedural computation of RD_{MFP} would be sufficient. Therefore an approach that eliminates global variables is used, where

global variables are substituted by local variables in every procedure. At call sites the global variables are modeled as call-by-value-result parameters, which is correct without call-by-reference parameters and aliasing. With a simple flow-insensitive data flow analysis the introduction of call-by-value-result parameters can be effectively reduced to the (global) variables that may be used or modified inside a procedure:

- $\mathsf{GMOD}(p)$: the set of all variables that might be modified if procedure p is called.
- $\mathsf{GREF}(p)$: the set of all variables that might be referenced if procedure p is called.

To compute the effect at the calling node n of procedure q calling procedure p ($p \in \mathsf{calls}(q)$), the GMOD and GREF sets of p must be back-binded, i.e. without call-by-reference parameters and aliasing this is just removing local variables:

$$\begin{aligned} \mathsf{bind}^{-1}(S, p) &= S - \mathsf{locals}(p) \\ \mathsf{GMOD}(n) &= \mathsf{bind}^{-1}(\mathsf{GMOD}(p)) \\ \mathsf{GREF}(n) &= \mathsf{bind}^{-1}(\mathsf{GREF}(p)) \end{aligned}$$

To compute the GMOD and GREF sets for a procedure p, first the set of all directly modified or referenced variables in p, $\mathsf{IMOD}(p)$ and $\mathsf{IREF}(p)$, are computed. This can already be done during parsing or while constructing the CFGs. Second, the effects of all calls in p ($= \mathsf{calls}(p)$) must be included:

$$\begin{aligned} \mathsf{GMOD}(q) &= \mathsf{IMOD}(q) \cup \bigcup_{p \in \mathsf{calls}(q)} \mathsf{bind}^{-1}(\mathsf{GMOD}(p), p) \\ \mathsf{GREF}(q) &= \mathsf{IREF}(q) \cup \bigcup_{p \in \mathsf{calls}(q)} \mathsf{bind}^{-1}(\mathsf{GREF}(p), p) \end{aligned}$$

Again, these equations are recursively defined and the minimal fixed point solution is computed iteratively with empty GMOD and GREF sets as initializations.

Any variable v that may be modified by a call to procedure p ($v \in \mathsf{GMOD}(p)$) is assumed to be used and modified at the call site itself. Under this assumption the data dependence computation is done via the intraprocedural MFP framework for $\mathrm{RD}_{\mathrm{MFP}}$ with special def and ref sets for the transfer function at the call nodes:

$$\begin{aligned} \mathsf{def}(n) &= \mathsf{GMOD}(n) \\ \mathsf{ref}(n) &= \mathsf{GMOD}(n) \cup \mathsf{GREF}(n) \end{aligned}$$

The extension of the PDG for *interprocedural programs* introduces more nodes and edges: For every procedure a *procedure dependence graph* is constructed, which is basically a PDG with *formal-in* and *-out* nodes for every formal parameter of the procedure. A procedure call is represented by a *call* node and *actual-in* and *-out* nodes for each actual parameter. The call node is connected to the

entry node by a *call* edge, the *actual-in* nodes are connected to their matching *formal-in* nodes via *parameter-in* edges and the *actual-out* nodes are connected to their matching *formal-out* nodes via *parameter-out* edges. Such a graph is called *Interprocedural Program Dependence Graph (IPDG)*. The *System Dependence Graph (SDG)* is an IPDG, where *summary edges* between actual-in and actual-out have been added representing transitive dependence due to calls [HRB90]. The concept of summary edges and the system dependence graph will be explained in the next chapter.

6.5 Related Work

There exists a lot of previous work for interprocedural analysis which cannot be discussed here; any good book on compilers can serve as an introduction, e.g. [ASU85, Muc97]. The two principle approaches for interprocedural analysis of section 6.3 have been introduced in [SP81]. The concept of interprocedurally realizable paths in section 6.2 has been taken from [Rep98].

Knoop [KS92, Kno98] presents an interprocedural data flow analysis framework that can be used to compute interprocedural data flow analysis problems efficiently. A complete system that generates interprocedural data flow analyses from specifications is presented in [Mar99, AM95]. It uses an intraprocedural approach that does not fit into the three approaches on page 79. A number of program analysis problems can be transformed to context-free language reachability problems, which can be seen as another framework [Rep98, RHS95].

Harrold et al [HRS98, SHR01] present *interprocedural control dependence* which occurs if a procedure can be left abnormally by an embedded halt statement. They extend the IPDG with corresponding edges and show how to compute slices within the extended graph [SHR99]. An efficient approach to compute interprocedural control dependence is presented in [EBP01].

Chapter 7

Interprocedural Slicing

To slice programs with procedures, it is not enough to perform interprocedural data flow analysis as presented in the last chapter. If the intraprocedural slicing algorithm 3.1 on page 34 from section 3.2 is used on IPDGs or SDGs, the resulting slices are not accurate, as the *calling context* is not preserved: The algorithm may traverse a parameter-in edge coming from a call site into a procedure, may traverse some edges there and may traverse a parameter-out edge going to a different call site. The sequence of traversed edges (the path) is an *unrealizable path*: It is impossible for an execution that a called procedure does not return to its call site. We consider an interprocedural slice to be *precise* if all nodes included in the slice are reachable from the criterion by a *realizable* path.

The next short section will transfer the notion of a realizable path in the ICFG to realizable paths in the IPDG and define slicing in those terms. The following section will explain summary edges and how they can be used for efficient computation of slices. A section about slicing without summary edges will follow. The last section will present an evaluation of the presented slicing algorithms.

7.1 Realizable Paths in the IPDG

Even when we performed precise interprocedural data flow analysis, the unconstrained traversal of the dependences will result in imprecise slices.

Example 7.1: Figure 7.1 on the following page shows a fragment of a program that contains two calls to procedure `p` in lines 3 and 6. Assume that the parameters are passed by call-by-reference. A backward slice for variable `y` in line 9 will include line 4, despite that it has no influence on `y` at line 9. The reason is that the call in line 3 is forcing line 11–13 into the slice. Line 13 forces the inclusion of line 4, because parameter `a` is bound to variable `x` at the call in line 6. A precise algorithm should obey the calling context and omit line 4 from the slice.

```
1    read x
2    read y
3    p(y,x)
4    x = x + 1
5    y = y + 1
6    p(x,y)
8    print x
9    print y

10  proc p(a,b):
11    b = b + 1
12    c = b / 2
13    a = a + c
14  end
```

Figure 7.1: A simple example with a procedure

There is a correspondence between realizable paths in the ICFG and paths in the IPDG. *Interprocedurally realizable paths in the IPDG* are defined exactly like interprocedural realizable paths in the ICFG: The edges are marked with bracket symbols and a grammar checks the interprocedural realizability. This is similar to section 6.2 and straightforward, thus not presented here again.

Definition 7.1 (Slice in an IPDG)
The (*backward*) *slice* $S(n)$ of an IPDG at node n consists of all nodes on which n (transitively) depends via an interprocedurally realizable path:

$$S(n) = \{m \mid m \rightarrow_R^\star n\}$$

A slice can also be defined for a criterion set of nodes C:

$$S(C) = \{m \mid m \rightarrow_R^\star n \wedge n \in C\}$$

These definition s cannot be used in an algorithm directly because it is impractical to check whether paths are interprocedurally realizable. The next section will present an efficient algorithm for slicing that complies with the definitions.

7.2 Slicing with Summary Edges

Accurate slices can be calculated with a modified algorithm on SDGs. The benefit of SDGs is the presence of *summary* edges that represent transitive dependence due to calls. Summary edges cam be used to identify actual-out nodes that are reachable from actual-in nodes by an interprocedurally realizable path through the called procedure without analyzing it. The idea of the slicing algorithm using summary edges [HRB90, RHSR94] is first to slice from the criterion only ascending into calling procedures, and then to slice from all visited nodes only descending into called procedures. The algorithm 7.1 on the next page is a variant of the original algorithm in [HRB90]. It is a two-phase technique which first computes a descending slice for the criterion, and computes an ascending slice from the result of the first phase. The original algorithm does not

Algorithm 7.1 Summary Information Slicing (in SDGs)

Input: $G = (N, E)$ the given SDG
$s \in N$ the given slicing criterion
Output: $S \subseteq N$ the slice for the criterion s

$W^{up} = \{s\}$
$W^{down} = \emptyset$
$S = \{s\}$
first pass, descending slice
while $W^{up} \neq \emptyset$ *worklist is not empty* **do**
  $W^{up} = W^{up} / \{n\}$ *remove one element from the worklist*
  foreach $m \rightharpoonup n \in E$ **do**
    if $m \notin S$ **then**
      if $m \rightharpoonup n$ is a parameter-out edge ($m \overset{po}{\rightharpoonup} n$) **then**
        $W^{down} = W^{down} \cup \{m\}$
        $S = S \cup \{m\}$
      else
        $W^{up} = W^{up} \cup \{m\}$
        $S = S \cup \{m\}$
second pass, ascending slice
while $W^{down} \neq \emptyset$ *worklist is not empty* **do**
  $W^{down} = W^{down} / \{n\}$ *remove one element from the worklist*
  foreach $m \rightharpoonup n \in E$ **do**
    if $m \notin S$ **then**
      if $m \rightharpoonup n$ is not a parameter-in or call edge ($m \overset{pi,cl}{\rightharpoonup} n$) **then**
        $W^{down} = W^{down} \cup \{m\}$
        $S = S \cup \{m\}$
return S *the set of all visited nodes*

traverse parameter-out edges in the first phase, ignores parameter-in and call edges in the second phase and uses the result of the first phase as slicing criterion for the second phase. This has the effect that many nodes will be visited twice. The new algorithm 7.1 is therefore improved in that every node is never visited more than once. This improvement is quite obvious and has been used in CodeSurfer from the beginning, though has not been published. This algorithm is correct, because the nodes inserted into the down-worklist are formal-out nodes that can only be reached by traversing parameter-out edges and will never be visited in the first phase via different edges. In the following, this algorithm is called *context-sensitive slicing (CSS)*. It computes a *non-truncated backward slice*, where non-truncated means that nodes in called procedures are included. With small modifications it can be made to compute other variants of slicing:

Forward slicing. All edges are traversed in the opposite direction.

Truncated slicing. A truncated (backward) slice does not contain nodes from called procedures; it does not ascend into them. To compute it, the second pass is left out (because it computes exactly those nodes).

Same-level slicing. A same-level (backward) slice does not contain nodes from calling procedures; it does not descend into them. To compute it, the first pass has to ignore parameter-in and call edges.

All variants can be combined: for example, a same-level truncated slice only contains nodes of the procedure containing the criterion node. This is different from computing a non-truncated non-same-level slice and removing nodes from other procedures: Due to recursion some nodes of the procedure may only be reached by ascending or descending at call sites.

Generation of Summary Edges

The algorithm to compute summary edges in [HRB90] is based on attributed grammars. A more efficient algorithm was presented in [RHSR94], which will be used in the following and is shown in algorithm 7.2 on the next page. This algorithm starts on an IPDG without summary edges and checks for all pairs of formal-in and -out nodes, if a path in the dependence graph between the nodes exists and corresponds to a same-level realizable (matched) path in the control flow graph. If such a path exists, a summary edge between the corresponding actual-in and -out node is inserted. Because the insertion of summary edges will make more paths possible, the search iterates until a minimal fixed point is found. The algorithm only follows data dependence, control dependence and summary edges. Therefore any path consisting of these edges must correspond to a same-level realizable path. This algorithm is basically of cubic complexity [RHSR94].

7.3 Context-Sensitive Slicing

There are situations in which summary edges cannot be used: In presence of interference in concurrent programs dependence is no longer transitive (see chapter 5), which is a requirement for summary edges. As interference dependence crosses procedure boundaries, it cannot be summarized by summary edges. Under such circumstances the calling context has to be preserved explicitly during the traversal of the (threaded) IPDG.

The computation and usage of summary edges can be seen as an instance of *effect calculation*, one of the two approaches discussed in section 6.3. *Call strings*, the other approach, are now used for slicing where the calling context is encoded explicitly during analysis.

Let each call node and its actual-in and -out nodes in the IPDG G be given a unique call site index s_i. A sequence of call sites $c = s_{i_1} ... s_{i_n}$ is a call string.

Algorithm 7.2 Computing Summary Edges

Input: $G = (N, E)$, the given IPDG
Output: $G' = (N, E')$, the corresponding SDG

Initialization
$W = \emptyset$, *worklist*
$P = \emptyset$
foreach $n \in N$ which is a formal-out node **do**
  $W = W \cup \{(n, n)\}$
  $P = P \cup \{(n, n)\}$
Iteration
while $W \neq \emptyset$ *worklist is not empty* **do**
  $W = W/\{(n, m)\}$ *remove one element from the worklist*
  if n is a formal-in node **then**
    foreach $n' \xrightarrow{pi} n$ which is a parameter-in edge **do**
      foreach $m \xrightarrow{po} m'$ which is a parameter-out-edge **do**
        if n' and m' belong to the same call site **then**
          $E = E \cup n' \xrightarrow{su} m'$ *add a new summary edge*
          foreach $(m', x) \in P \wedge (n', x) \notin P$ **do**
            $P = P \cup \{(n', x)\}$
            $W = W \cup \{(n', x)\}$
  else
    foreach $n' \xrightarrow{dd,cd,su} n$ **do**
      if $(n', m) \notin P$ **then**
        $P = P \cup \{(n', m)\}$
        $W = W \cup \{(n', m)\}$
return G *the SDG*

During traversal of a parameter-out edge from a call site s going *down* into the called function, a new, longer call string $c' = sc$ is generated. If a parameter-in or call edge is traversed back *up* to a call site s', this call site must *match* the current leading element of the call string ($c = s'c'$). Using call strings a context-sensitive slicing method which is as precise as context-sensitive slicing based on summary edges can be defined: is precise in respect to realizable paths.

7.3.1 Explicitly Context-Sensitive Slicing

In Figure 7.3 on the facing page a general slicing algorithm obeying calling context via call strings is shown. Variants of the algorithm come from the definition of down, up, match and equals. The simplest definition is as follows[1]:

$$\begin{aligned}
\mathsf{down}(c,s) &\rightarrow \mathrm{cons}(s,c)\\
\mathsf{up}(c) &\rightarrow \begin{cases} \mathrm{cdr}(c) & | \; c \neq \epsilon \\ \epsilon & | \; c = \epsilon \end{cases}\\
\mathsf{match}(s,c) &\rightarrow c = \epsilon \vee s = \mathrm{car}(c)\\
\mathsf{equals}(c_1,c_2) &\rightarrow c_1 = c_2
\end{aligned}$$

In presence of recursion this simple approach fails, as neither the set of call strings nor the call strings themselves are finite. Agrawal and Guo presented an improved algorithm named ECS in [AG01a], where the call strings are cycle free. They define down as follows (up and equals are as above):

$$\mathsf{down}(c,s) \rightarrow \begin{cases} \mathrm{cons}(s,c) & | \; c = s_1 s_2 \ldots s_k \wedge \forall_{s_i} : s \neq s_i \\ s_i \ldots s_k & | \; c = s_1 s_2 \ldots s_k \wedge s = s_i \end{cases}$$

However, using this definition the resulting slices are not correct as they might leave out nodes (statements) of the slices. The incorrectness is based on the following observation: As soon as a call string c would form the string $c = s_x s_{y_1} .. s_{y_n} s_x s_{z_1} ... s_{z_m}$, it is replaced by $c' = s_x s_{z_1} ... s_{z_m}$ to remove the cycle. Now, the algorithm fails to propagate the effects with call string c' (which includes effects with call string c) back to the call site s_{y_1}, because the call string $c'' = s_{y_1} .. s_{y_n} s_x s_{z_1} ... s_{z_m}$ is not generated by $\mathsf{up}(c')$.

Example 7.2: A simple counter example based on this observation is given in Figure 7.2 on page 92, where two procedures are shown as an IPDG. Procedure 1 contains its entry node 1 and a call site A, composed from the call node A1, two actual-in nodes A2 and A3, and an actual-out node A4. The second procedure contains its entry node 2, two formal-in nodes 3 and 4, a formal-out node 6, a node if for an if-statement, and a call site B, composed from the call node B1, two actual-in nodes B2 and B3, and an actual-out node B4. The nodes

[1] equals will be replaced by an optimized version in section 7.3.4. The following algorithm are already prepared for that version—therefore the check "m has not been marked with a context c' for which $\mathsf{equals}(c, c')$ holds".

Algorithm 7.3 Explicitly Context-Sensitive Slicing

Input: $G = (N, E)$, the given IPDG
$s \in N$, the given slicing criterion
Output: $S \subseteq N$ the slice for the criterion s

Initialization
$W = \{(s, \epsilon)\}$, *worklist*
while $W \neq \emptyset$ *worklist is not empty* do
  $W = W/\{(n, c)\}$ *remove one element from the worklist*
  $S = S \cup \{n\}$
  foreach $m \rightharpoonup n \in E$ **do**
    if m has not been marked with a context c'
      for which $\mathsf{equals}(c, c')$ holds **then**
      if $m \rightharpoonup n$ is a parameter-in or call edge **then**
        Let s_m be the call site of m
        if $\mathsf{match}(s_m, c)$ **then**
          $c' = \mathsf{up}(c)$
          $W = W \cup \{(m, c')\}$
          mark m with c'
      elsif $m \rightharpoonup n$ is a parameter-out edge **then**
        Let s_n be the call site of n
        $c' = \mathsf{down}(c, s_n)$
        $W = W \cup \{(m, c')\}$
        mark m with c'
      else
        $W = W \cup \{(m, c)\}$
        mark m with c
return S,*the set of all visited nodes*

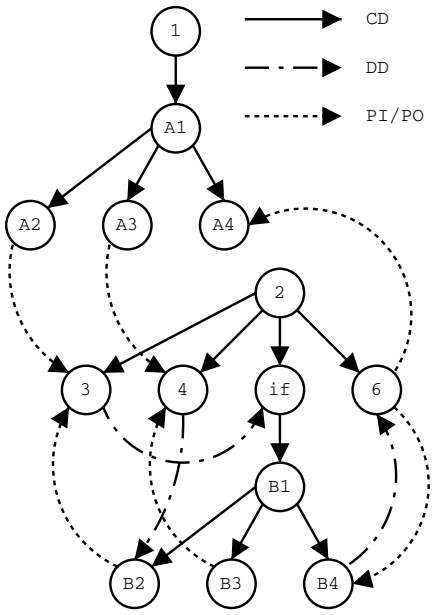


Figure 7.2: Counter example for Agrawal's ECS

inside the procedures are connected by control and data dependence edges. Also, procedure 1 calls procedure 2 at call site A, and procedure 2 calls itself at call site B. The actual-in/-out nodes are properly connected to their formal-in/-out nodes by parameter-in/-out edges; the call edges are not shown. Let us do a backward slice from A4: The initial worklist is $\{(\texttt{A4}, \epsilon)\}$. A4 is reachable from A1 and 6, and the worklist will contain $\{(\texttt{A1}, \epsilon), (6, A)\}$. Next, A1 is visited, which is reachable from 1, and this leaves the worklist as $\{(6, A)\}$. Node 6 is reached (transitively) from 2, 3, B1, B4, and if, which are all marked with the call string A. B4 is reached from 6 by recursion. At this point, the worklist is $\{(3, A), (6, BA)\}$ and node 6 has two marks: A and BA. Node 3 is reachable from A2, where the call string matches (at this point, node 3 is only marked with A). It is also reachable from B2, but the call string doesn't match. From node 6 the call string BA is now propagated to 2, 3, B1, B4, and if. Because node 3 is now marked with BA, A is propagated to B2, which is visited the first time. B2 is reachable from node 4 and transitively from A3. At this point, only B3 has not been visited yet, the nodes 1 and A1–A4 are marked with ϵ; 2, 3, if, 6, B1, and B4 are marked with A and BA, and 4 and B2 are only marked with A. Now, the worklist only contains $\{(\texttt{B4}, BA)\}$. Due to the recursive call, a new call string BBA would be generated and propagated to node 6. However, the cycle removal in down folds BBA into BA, which has previously been used at node 6. The ECS algorithm now terminates (the worklist is empty) but the generated slice doesn't contain B3 which is wrong: a propagation of BBA would visit B3.

7.3.2 Limited Context Slicing

One popular approach to circumvent the infinity problems of call strings is the limitation of their length (*call string suffix approximation* in [SP81]). For a given k, the call string is not allowed to be longer than k elements. If the call string is already k elements long before concatenation, the oldest element is removed:

$$\mathsf{down}(c, s) \to \begin{cases} \mathrm{cons}(s, c) & \mid c = s_1 s_2 \dots s_l \wedge l < k \\ s s_1 \dots s_{k-1} & \mid c = s_1 s_2 \dots s_l \wedge l = k \end{cases}$$

We can use the general algorithm of Figure 7.3 on page 91 with these modifications. This variant is called k-*limited context slicing (*k*LCS)*. This approach becomes quite imprecise (but is still correct) in presence of recursion: Consider a call string of maximal length k. If this call string is propagated into a recursive procedure, it may be propagated k times into the called recursive procedure. The result is a call string that only contains call sites in the recursive procedure. This call string may then be propagated k times back into the calling recursive procedure. Now, the resulting call string is empty and all procedures that call the recursive one are marked with the empty call string. This makes the analysis context-insensitive for procedures calling recursive procedures (directly or indirectly) and causes a reduced precision.

7.3.3 Folded Context Slicing

As seen, a way to make the call strings finite is to remove cycles and maintain correctness additionally. Instead of removing them explicitly a different approach is presented: First, the *calling context graph* is built. This is a graph where the nodes are the call sites and the edges (n, m) represent calls that lead from call site n into a procedure that contains call site m. In that graph, the strongly connected components are folded into one single node and the call sites of such a component are replaced by one single call site. During that process all recursive call sites are marked.

The definitions of down etc. are then adapted not to iterate through the cycles, making the contexts finite. Therefore they use the replacement $[s]$ of a call site s:

$$\begin{aligned} \mathsf{down}(c, s) &\to \begin{cases} c & \mid [s] = \mathrm{car}(c) \\ \mathrm{cons}([s], c) & \mid \text{otherwise} \end{cases} \\ \mathsf{up}(c) &\to \begin{cases} \mathrm{cdr}(c) & \mid c \neq \epsilon \\ \epsilon & \mid c = \epsilon \end{cases} \\ \mathsf{match}(s, c) &\to c = \epsilon \vee [s] = \mathrm{car}(c) \\ \mathsf{equals}(c_1, c_2) &\to c_1 = c_2 \end{aligned}$$

This definition of down does not create new contexts if the old and the new call site are in the same recursive cycle.

This variant will be called *unlimited folded context slicing (UFCS)*. If it is used within the algorithm of Figure 7.3 on page 91, the same incorrectness as in

Algorithm 7.4 Folded Context-Sensitive Slicing

Input: $G = (N, E)$ the given SDG
$s \in N$ the given slicing criterion
Output: $S \subseteq N$ the slice for the criterion s

Initialization
$W = \{(s, \epsilon)\}$, *worklist*
while $W \neq \emptyset$ *worklist is not empty* **do**
  $W = W/\{(n, c)\}$ *remove one element from the worklist*
  $S = S \cup \{n\}$
  foreach $m \rightharpoonup n \in E$ **do**
    if m has not been marked with a context c'
      for which $\mathsf{equals}(c, c')$ holds **then**
      if $m \rightharpoonup n$ is a parameter-in or call edge **then**
        Let s_m be the call site of m
        if $\mathsf{match}(s_m, c)$ **then**
          if s_m is marked as recursive
            and m has not been marked with a context c'
            for which $\mathsf{equals}(c, c')$ holds **then**
            $W = W \cup \{(m, c)\}$
            mark m with c
          $c'' = \mathsf{up}(c)$
          $W = W \cup \{(m, c'')\}$
          mark m with c''
      elsif $m \rightharpoonup n$ is a parameter-out edge **then**
        Let s_n be the call site of n
        if s_n is marked as recursive
          and $\mathrm{car}(c) = s_n$ **then**
          $W = W \cup \{(m, c)\}$
          mark m with c
        else
          $c' = \mathsf{down}(c, s_n)$
          $W = W \cup \{(m, c')\}$
          mark m with c'
      else
        $W = W \cup \{(m, c)\}$
        mark m with c
return S, *the set of all visited nodes*

Agrawal's algorithm is obtained. Figure 7.4 on the facing page presents a corrected algorithm, modified as follows:

1. If the algorithm descends into a called recursive function, it propagates *two* call strings: the call string generated by up and the actual call string (because of the possible recursion).

2. If the algorithm ascends into a calling recursive function, it propagates the actual call string if the actual call site is already present either in (the first element of) the call string or the call string generated by down.

This modified algorithm is general enough to also be used in an k-limited version which is defined by:

$$\mathsf{down}(c, s) \rightarrow \begin{cases} c & | \; [s] = \mathrm{car}(c) \\ \mathrm{cons}([s], c) & | \; c = [s_1][s_2]\ldots[s_l] \wedge [s] \neq [s_1] \wedge l < k \\ [s][s_1]\ldots[s_{k-1}] & | \; c = [s_1][s_2]\ldots[s_l] \wedge [s] \neq [s_1] \wedge l = k \end{cases}$$

This variants will be called k-*limited folded context slicing (kFCS)*. The evaluation will show that the unlimited variant is impractical due to combinatorial explosion of the set of call strings.

7.3.4 Optimizations

During early test-case evaluations it was observed that implementation decisions have a high impact on runtime. To prevent an evaluation where bad results stem from inefficient implementations, a considerable amount of time has been spent on experiments with different optimizations.

Subsuming Call Strings

To reduce the amount of generated call strings, the following observation has been used: Call strings that are prefixes of other call strings subsume the other call strings. Consider the context $c = c's$. If a node is marked with both call strings c and c', the propagation of c' reaches a superset of the nodes that are reached by the propagation of c. Therefore the definition of equals has to be replaced:

$$\mathsf{equals}(c_1, c_2) \rightarrow c_1 = s_1 s_2 \ldots s_k \wedge c_2 = s_1 s_2 \ldots s_k s_{k+1} \ldots s_{k+l}$$

With this definition, only the shorter call strings are propagated and the amount of propagated call strings is much smaller.

Worklist Ordering

One of the important optimizations was the handling of the worklists. Different combinations of appending (depth-first) and prepending (breadth-first) showed differences up to 300% in runtime of summary slicing. With limited

		LOC	proc	nodes	slices	edges	summ.	%	time
A	gnugo	3305	38	3875	281	10657	2064	16	0.03
B	ansitape	1744	76	6733	1082	18083	12746	41	0.15
C	assembler	3178	685	13393	2401	97908	114629	54	3.58
D	cdecl	3879	53	5992	697	17322	9089	34	0.08
E	ctags	2933	101	10042	1621	24854	20483	45	0.24
F	simulator	4476	283	9143	1019	22138	5022	18	0.06
G	rolo	5717	170	37839	6540	264922	170108	39	5.53
H	compiler	2402	49	15195	1017	45631	58240	56	0.80
I	football	2261	73	8850	818	30474	17605	37	0.35
J	agrep	3968	90	11922	1403	35713	12343	26	0.19
K	bison	8313	161	25485	3744	84794	29739	26	0.72
L	patch	7998	166	20484	3099	104266	83597	44	4.39
M	flex	7640	121	38508	5191	235687	144496	38	4.19
N	diff	13188	181	46990	10130	471395	612484	57	28.2

Figure 7.3: Details of the test-case programs

context slicing, a badly chosen combination can cause a combinatorial explosion of call strings (and thus runtime). To follow the most general call strings first and therefore subsume all less general call strings, elements with a more general (shorter) call string get prepended to the worklist and elements with a less general (longer) call string get appended. This is only done by looking at the type of the traversed edge; no sorting is involved.

We also tried a priority queue[2], sorted with respect to the length of the contexts. This caused no optimization but an increased runtime due to the overhead (in comparison to the best combination of prepending and appending to the unsorted worklist).

7.4 Evaluation

All slicing algorithms of the previous sections have been implemented to evaluate them completely. The details of the analyzed programs are shown in Figure 7.3. The programs stem from two different sources: ctags, patch and diff are the GNU programs. The rest are from the benchmark database of the PROLANGS Analysis Framework (PAF) [RLP+01]. The 'LOC' column shows lines-of-code (measured via `wc -l`), the 'proc' column the number of procedures (the number of entry nodes in the PDG) and the 'nodes' column shows the number of nodes in the IPDG. Like Agrawal and Guo, the formal-in nodes had been selected as slicing criteria to make the results comparable; the amount of resulting slicing criteria (the number of formal-in nodes) is shown in column

[2]Basically an array of $k + 1$ worklists was used, one worklist for every possible context length up to limit k. Thus insertion is a constant operation.

	CIS	CSS	1LCS	2LCS	3LCS	4LCS	5LCS	6LCS
A	1861	1798	100	100	100	100	100	100
B	2909	1645	7	74	74	74	74	100
C	6458	4286	48	48	48	100	100	100
D	1039	880	99	99	100	100	100	100
E	3207	2010	100	100	100	100	100	100
F	5455	3212	83	83	100	100	100	100
G	12819	7766	46	57	84	92	92	92
H	7474	6731	22	26	26	26	26	26
I	3081	2593	100	100	100	100	–	–
J	3521	3183	41	49	59	100	100	100
K	7215	1859	74	96	96	97	97	97
L	9680	7965	92	99	99	100	100	100
M	14558	6172	29	29	29	29	29	29
N	19641	9179	88	98	98	98	98	–
			1FCS	2FCS	3FCS	4FCS	5FCS	6FCS
A			100	100	100	100	100	100
B			7	74	74	74	74	100
C			48	48	48	100	100	100
D			99	99	100	100	100	100
E			100	100	100	100	100	100
F			83	83	100	100	100	100
G			46	57	84	92	92	**92**
H			22	**30**	**32**	**46**	**73**	**73**
I			100	100	100	100	100	100
J			41	49	59	100	100	100
K			74	96	96	**98**	**100**	**100**
L			92	99	99	100	100	100
M			*29*	**30**	**30**	**32**	**98**	**100**
N			88	98	**99**	**99**	**100**	**–**

Figure 7.4: Precision of kLCS and kFCS (avg. size)

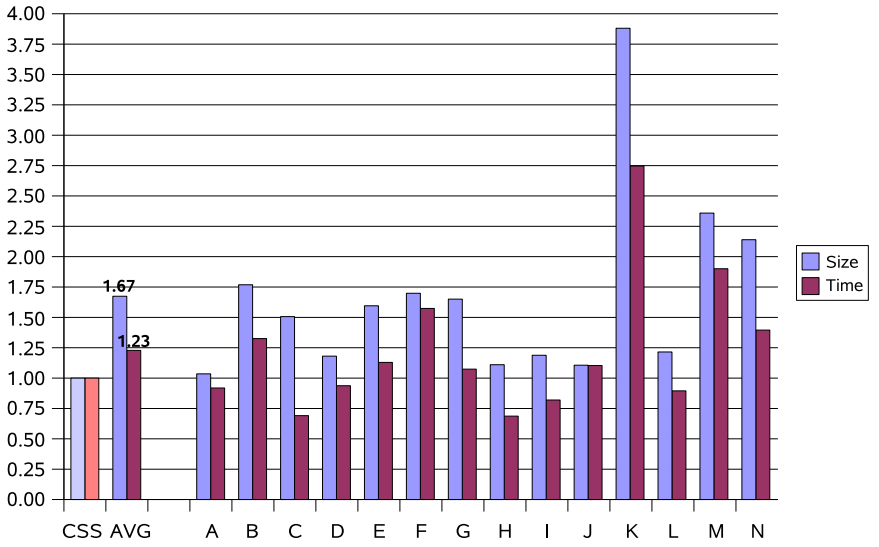


Figure 7.5: Context-insensitive vs. context-sensitive slicing

'slices'. The limited size of the test-case programs is due to the number of slices, which caused a quadratic runtime for one complete test.

A first evaluation shows how expensive the generation of summary edges is: As seen in column 'time' of Figure 7.3 on page 96, it is not that expensive: the time needed to construct the summary edges is always below one second and greatly depends on the number of summary edges. All programs contain very high numbers of summary edges (column 'summ.'), sometimes every second edge is a summary edge (column '%'). This causes a blowup of the SDG in comparison with the corresponding IPDG. For larger programs that are not shown here, SDGs were encountered where 90% of the edges are summary edges.

7.4.1 Precision

How precise are the new algorithms kLCS and kFCS? How is precision measured? Context-insensitive slicing (CIS) is the most simple (and imprecise) algorithm—considered to have 0% precision. Slicing with summary edges (CSS) is precise with respect to realizable paths and considered as 100% precise.

In Figure 7.4 on the page before, the precision of the four different algorithms is presented. The amount of memory and time in which all slices had to be calculated had been limited: The tests marked with "–" needed more than 300MB core memory or didn't finish in fewer than eight hours on 1GHz ma-

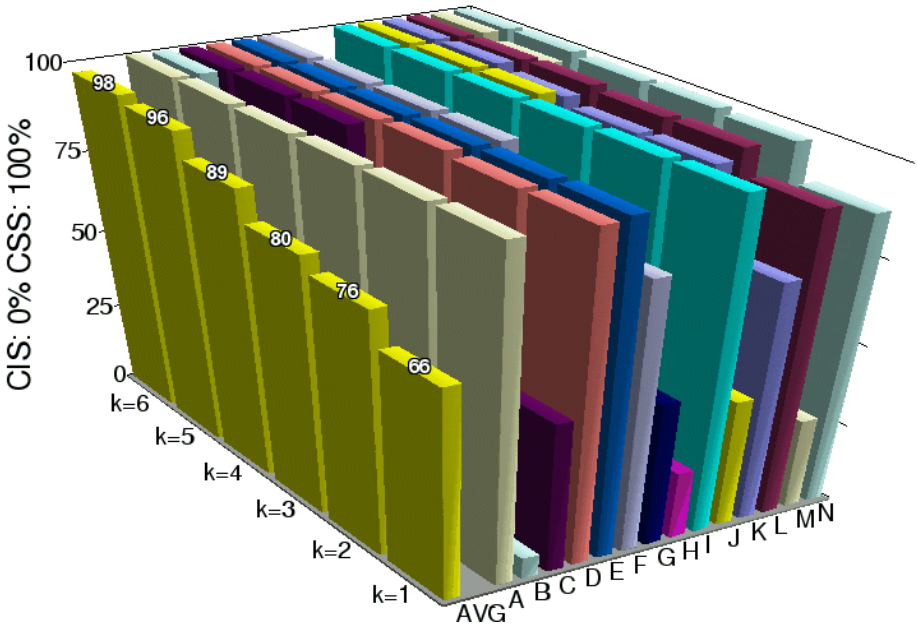


Figure 7.6: Precision of kLCS (avg. size)

chines with 512MB. The CIS column gives the average size of a CIS slice and the CSS column the average size of a CSS slice. There is already a big variation: The average CIS slice is 3%–300% larger than the average CSS slice. This is shown in Figure 7.5 on the facing page.

Many programs have a high precision for even small k in both kLCS and kFCS. This can be seen in Figure 7.6, where the precision for kLCS is shown in percentage. In some cases (shown in bold in Figure 7.4 on page 97) the precision of kFCS is higher than of kLCS. In only one case (flex, M) the precision of 1FCS is slightly less than 1LCS (less than 0.1%, shown in italics).

7.4.2 Speed

In Figure 7.7 on the next page the runtimes of the test-cases for the different slicing algorithms can be seen (in sec.). The given times are for the computation of the complete set of slicing criteria. To get the average time for a single slice one has to divide by the number given in the 'slices' column of Figure 7.3 on page 96. The average time is not shown because most numbers would be sub-second. In the first two columns of Figure 7.7 and also in Figure 7.5 it can be seen that the time needed to do CSS slices is less than the time for CIS slices if the CSS slices have a much higher precision: the quadratic complexity of CSS is more than compensated by the smaller amount of nodes that are visited. Only

	CIS	CSS	1LCS	2LCS	3LCS	4LCS	5LCS	6LCS
A	1,13	1,23	3,05	5,99	14,8	54,1	267	1357
B	7,37	5,56	17,8	18,8	29,7	73,6	240	845
C	81,5	118	158	206	228	171	177	143
D	1,65	1,76	3,70	5,18	8,49	11,5	10,6	8,30
E	11,1	9,83	17,2	24,0	38,3	148	352	574
F	12,4	7,88	27,4	97,4	97,4	121	176	206
G	480	447	768	1033	1174	1511	2007	2738
H	21,3	31,0	68,9	200	470	1461	5123	19140
I	6,01	7,33	14,3	29,0	75,6	621	–	–
J	12,8	11,6	30,3	39,1	52,0	39,7	59,5	120
K	83,5	30,4	108	162	323	373	335	284
L	119	133	227	313	382	402	398	424
M	287	151	586	918	1289	1672	1899	2259
N	1824	1307	1170	1580	2737	6675	20849	–
			1FCS	2FCS	3FCS	4FCS	5FCS	6FCS
A			2,65	4,36	8,65	11,0	11,4	12,5
B			17,8	18,0	26,1	49,8	82,5	77,0
C			160	207	230	174	178	144
D			3,75	5,13	8,48	11,6	10,8	8,18
E			17,3	24,3	41,6	137	343	579
F			27,4	87,5	98,2	121	178	208
G			749	964	1077	1355	1804	2600
H			48,0	200	487	2030	5220	8540
I			13,9	25,2	29,6	41,1	46,3	45,3
J			29,9	38,3	49,8	31,5	31,9	30,6
K			109	165	334	430	397	374
L			229	315	384	404	400	426
M			591	1088	2127	4571	2102	2420
N			1109	1341	1946	3182	9227	–

Figure 7.7: Runtimes of kLCS and kFCS (sec.)

when the precision of CSS is not much better, is the CIS algorithm faster (see A, C, D, H, I, L).

A typical problem of call string approaches is the combinatorial explosion of the generated call strings. This is also present in kLCS (figure 7.7, see A, B, E, H, I, N) and kFCS (see E, H, N). However, the increased precision and the resulting smaller number of visited nodes reduced runtimes for higher k (see C, D, K for kLCS and B, C, D, K for kFCS). Often kFCS is much faster than kLCS—it also is less likely to suffer from combinatorial explosion. In many other cases it is slightly slower than kLCS resulting from the overhead propagating *two* call strings. In some situations kFCS is much slower than kLCS. This is not related to a problem specific to kFCS but stems from the higher precision of kFCS in these cases (see Figure 7.4 for comparison).

A further-increased k (not shown here) causes many test-cases to suffer from combinatorial explosion up to a point where slicing is not any longer possible with kLCS or kFCS (e.g. test-case I for $k > 4$). The previously-described effect of higher speed for increased k is never compensating the combinatorial explosion and the experience of Agrawal and Guo [AG01a] who successfully used an unlimited call string slicing algorithm could not be shared. This experience is backed up by Atkinson and Griswold in [AG96], who reported the same for the Weiser style control flow graph approach.

Comparing the runtimes of kLCS and kFCS with CSS, it can be seen that even for $k = 1$ the runtimes of kLCS and kFCS are much higher than those of CSS. This leads to the conclusion that as long as summary edges are available, one should use CSS for slicing. This stays the same if we consider the overhead of calculating the summary edges: The generation of summary edges is part of the analysis to generate the PDG, which is expensive even without their generation. The overhead is only not affordable in situations where only one or two slices have to be done; in those situations the use of the Weiser-style control-flow-graph based slicing may probably be better anyway.

If summary edges are not available and the imprecision of context-insensitive algorithms is unacceptable, one must revert to k-limiting algorithms, where kFCS is preferable to kLCS.

7.4.3 Influence of Data Flow Analysis Precision

The measured precision of slicing is always dependent on the precision of the underlying data flow analysis. This section shows a simple experiment to evaluate the effect of the underlying data flow analysis. For that reason, the flow-sensitive computation of data dependence was replaced by two versions:

1. An experiment with an analysis where a definition of a variable is never killed by any other definition: If a path from a definition to a usage exists, the definition reaches the usage—independent of other definitions on that path. This is still flow-sensitive.

2. An experiment with a flow-insensitive analysis. Again, the reason is to check for effects of points-to analysis, which is itself flow-insensitive.

Program	A	B	C	A/B	A/C
agrep	47	50	56	6	19
ansitape	24	25	45	4	88
assembler	34	34	43	0	26
bison	11	13	17	18	55
cdecl	32	33	34	3	6
compiler	44	45	48	2	9
ctags	24	24	25	0	4
diff	33	36	50	9	52
flex	31	33	52	6	68
football	25	26	26	4	4
gnugo	59	62	70	5	19
lex315	26	45	47	73	81
loader	22	23	26	5	18
patch	47	49	49	4	4
plot2fig	23	24	24	4	4
rolo	25	25	30	0	20
simulator	47	49	54	4	15

Table 7.1: Precision

To conduct these experiments, SDGs have been constructed for several programs and exactly 1000 slices have been computed from 1000 evenly distributed nodes in every SDG. The results are presented in table 7.1. Column A shows the average size of the slice as percentage of the number of nodes in the SDG. Column B shows the same percentage for the non-killing analysis (experiment 1) and column C for the flow-insensitive analysis (experiment 2). Columns A/B and A/C show how much bigger the sizes in average are compared to the precise analysis (in percent). This shows that flow-sensitivity has a high impact on the precision of slices. On the other hand, killing has not that big effect: on average, the computed slices are only 5% larger if definitions are never killed.

7.5 Related Work

The system dependence graph has been introduced in [HRB90] as the interprocedural extension of the program dependence graph [FOW87]. There, the computation of summary edges was done with a *linkage grammar*. This has later been superseded by a more efficient algorithm [RHSR94] for computing summary edges. Another approach to building system dependence graphs is presented in [LJ00]: The summary edges are computed in reverse procedure invocation order, so that the transitive effects of called procedures are known at call sites. In the presence of recursion incomplete summary information is used and later expanded until a fixed point is reached. The same optimization is used in [FG97].

The original slicing algorithm from [HRB88, HRB90] is a two-phase approach: first, slice and descend into called procedures; second, slice from there and ascend into calling procedures. This algorithm is used in CodeSurfer with an improvement similar to the one in Section 7.2. In [Lak92] another improved algorithm is presented, where nodes are marked with three (unvisited, first, second) instead of two values (unvisited, visited). Similar to our approach, this enables an algorithm where every node and edge is visited only once. The implementation is evaluated and compared to the original algorithm, which shows that the improved algorithm has fewer edge visits. However, it has not been implemented optimally and some edges are still visited twice.

System-dependence-graph based slices are not executable because different call sites may have different parameters included in the slice. Binkley [Bin93a] shows how to extend slicing in system dependence graphs to produce executable slices.

System dependence graphs assume procedures to be single-entry-single-exit, which does not hold in presence of exceptions or interprocedural jumps. [HC98] obeys interprocedural control dependence from embedded halt statements.

An earlier approach to slicing recursive procedures precisely is [HDC88]: Starting with an intraprocedural slice, all called procedures present in the slice are inlined and the slice is redone. This is repeated until a fixed point is reached and the slice does not change anymore.

Extensive evaluations of different slicing algorithms have not been done yet—for control-flow-graph based algorithms some data reported by Atkinson and Griswold can be found in [BAG00, AG01b, AG96]. The only evaluation of program-dependence-based algorithms that the author is aware of has been conducted by Agrawal and Guo [AG01a] and just compares two algorithms, where one has flaws (as shown in this chapter).

The effect of the precision of the underlying data flow analysis and points-to analysis has been studied in a series of works [SH97, AG98, BL98, LH99a, OSH01a, MACE02].

Slicing for object-oriented programs is more complex than slicing for procedural languages because of virtual functions and the distinction of classes and objects. Malloy et al [MMKM94] have introduced the object-oriented program dependence graph which is used for slicing. Larsen and Harrold [LH96] have also extended the system dependence graph to a class dependence graph representing dependences in object-oriented programs. It has been improved by Tonella et al [TAFM97] and Liang and Harrold [LH98]. Other works for slicing object-oriented programs are [ZCU96, CW97, Zha99b, CX01b].

Chapter 8

Slicing Concurrent Interprocedural Programs

After suitable preparation through the previous chapters, this chapter will present an approach to slicing concurrent interprocedural programs. As seen before, the main problem for precision is context-sensitivity. In concurrent programs it is even harder: Müller-Olm has shown that precise context-sensitive slicing of concurrent programs is undecidable in general [MOS01]. He used a reduction onto two-counter-state-machines. Another important result has been proven by Ramalingam [Ram00]. He showed that context-sensitive analysis of concurrent programs with procedures and synchronization is undecidable in general. This applies not only to slicing but also to any data flow analysis. Therefore, conservative approximations have to be used to analyze concurrent programs. First of all, synchronization can be ignored. In that case, the results are still correct, but are imprecise because unrealizable paths are now allowed. The other simple approximation would be to do context-insensitive slicing, which is unacceptable due to much lower precision.

All approaches to slicing concurrent interprocedural programs rely on inlining to remove procedures. Therefore, no approach is able to slice recursive programs. This chapter will present an interprocedural version of the approach presented in chapter 5.

The basic idea is the adaption of the call string approach to concurrent programs. It is assumed that a concurrent program consists of separate threads that do not share code and communicate via shared variables. The context is now captured through one call string for every thread. The context is then a tuple of call strings that is propagated along the edges in IPDGs. The traversal of intraprocedural edges does not change the call string of the enclosing thread and the context can simply be propagated. During traversal of interprocedural edges the call string of the enclosing thread is used to check that a call always returns to the right call site. This may generate a single new context.

The traversal of interference edges is much more complicated: The call string of the newly reached thread is used to check that the reached node is reachable from a node with the old (saved) call string. To do that, every call string that the reached node can possibly have is checked. This can generate a set of new call strings that have to be propagated.

To avoid combinatorial explosion of call strings, a combined approach is pursued: using summary edges to compute the slice within threads. Additionally, call strings are generated and propagated along interference edges only if the slice crosses threads. With this approach many fewer contexts are propagated. This only outlines the idea of this chapter's approach, which will be formalized in the following sections.

The next section will present the underlying simple model of concurrent programs. It is followed by a section about threaded interprocedural control flow graphs and how to define reachability in them. Section three defines *threaded* interprocedurally realizable paths and the *threaded* interprocedural PDG. Section four defines slices based on these graphs and presents an algorithm to compute them. This chapter closes with a discussion about extensions to the concurrency model before drawing conclusions.

8.1 A Simple Model of Concurrency

For easier presentation, a different model of concurrency than the one of chapter 5 will be used. A concurrent program is assumed to consist of a set of threads $\Theta = \{\theta_1, \ldots, \theta_n\}$. Threads may be executed in parallel on different processors or interleaved on a single processor. All threads are started immediately after the program's start and the program exits after all threads have finished. The threads do not share any code, communication is done via global variables, and every statement is assumed to be atomic and synchronized properly. Every thread consists of a series of procedures that may call each other but may not call a procedure from a different thread. One of the procedures for every thread is the main procedure, which is called from the runtime system after the corresponding thread has started. The corresponding thread stops after the main procedure returns. Similar to chapter 5, synchronization is ignored for now.

8.2 The Threaded Interprocedural CFG

Every thread $t \in \Theta$ can be represented with its interprocedural control flow graph $G_t = (N_t^\star, E_t^\star, n_t^s, n_t^e)$. Because all threads are independent, no edges exist between the control flow graphs of two different threads. The *threaded interprocedural CFG (tICFG)* $G = (N^\Pi, E^\Pi, S^\Pi, X^\Pi)$ simply consists of the union of all ICFGs for the different threads:

$$N^\Pi = \bigcup_{t \in \Theta} N_t^\star$$

$$\begin{aligned} E^{\Pi} &= \bigcup_{t \in \Theta} E_t^{\star} \\ S^{\Pi} &= \{n_t^s \mid t \in \Theta\} \\ X^{\Pi} &= \{n_t^e \mid t \in \Theta\} \end{aligned}$$

Similar to definition 5.1 on page 52 the existence of function $\Pi(t)$ that returns the set of threads that may execute in parallel to thread t. In this underlying simple model, this is trivial: $\Pi(t) = \{t' \in \Theta \mid t' \neq t\}$. However, the following will not rely on that to make more complex models possible later on.

The main problem is context-sensitivity again. In this chapter, this problem is handled with explicit context through virtual inlining. It is assumed that the execution state of a thread is encoded by a context in the form of a (possibly infinite) call string (see section 6.3.2). The call string is represented as a stack of nodes, where the topmost node represents the currently executing statement. Then, all definitions are modified to use such contexts instead of nodes—this is called *virtual inlining*.

We first define when a context *reaches* another context:

Definition 8.1 (Context)
The execution state c of thread t is a *context* $c = n_0 \dots n_k$ representing an execution stack of nodes $n_i \in N_t^{\star}$ with the topmost node $T(c) = n_0$ (context c belongs to the current node n_0). The 'pop' function P is defined as $P(c) = n_1 \dots n_k$. Let $\theta(c) = \theta(T(c))$.

A context c *directly reaches* another context c': $c \rightharpoonup_R c'$, iff one of the following alternatives holds:

1. an edge $n \rightharpoonup n' \in E_t^{\star}$ exists and $n \rightharpoonup n' \notin E_t^C \wedge T(c) = n \wedge T(c') = n' \wedge P(c) = P(c')$ (a corresponding edge in the CFG of a procedure exists),
2. a call edge $n \rightharpoonup n' \in E_t^C$ exists and $T(c) = n \wedge T(c') = n' \wedge c = P(c')$ (a corresponding call edge exists), or
3. a return edge $n \rightharpoonup n' \in E_t^C$ exists and
   (a) $T(c) = n \wedge T(c') = n'$,
   (b) $P(P(c)) = P(c')$, and
   (c) $T(P(c)) \rightharpoonup n' \in (E_t^{\star} - E_t^C)$.

   (A corresponding return edge exists that returns to an immediate successor of the last call node—a return node.)

A context c *reaches* another context c': $c \rightharpoonup_R^+ c'$, iff a series of contexts $c_1, \dots, c_n$ exists, with $c = c_1 \wedge c' = c_n \wedge \forall_{1 \leqslant i < n} c_i \rightharpoonup_R c_{i+1}$. The set of possible contexts for ICFG $G_t = (N_t^{\star}, E_t^{\star}, n_t^s, n_t^e)$ is $C_t = \{c' \mid n_t^s \rightharpoonup_R^+ c' \vee c' = n_t^s\}$.

Note that $c \rightharpoonup_R^+ c'$ implies the existence of an interprocedurally realizable path from $T(c)$ to $T(c')$. Also, $\theta(c) \neq \theta(c') \Rightarrow c \not\rightharpoonup_R^+ c'$ because the ICFGs are disjunct.

The *threaded interprocedural witness* is now defined in terms of contexts, not nodes:

Definition 8.2 (Threaded Interprocedural Witness)
A sequence $l = \langle c_1, \ldots, c_k \rangle$ of contexts (execution stacks) is a *threaded interprocedural witness* in a tICFG, iff

$$\forall 1 \leqslant j < i \leqslant k : \theta(c_j) \in \Pi(\theta(c_i)) \vee c_j \rightharpoonup_R^+ c_i$$

Basically this means that all contexts in a thread must be reachable from its predecessors if they cannot execute in parallel.

Intuitively, a threaded interprocedural witness can be interpreted as a sequence of contexts that form a valid execution chronology.

The virtual inlining is also applied to the other definitions and theorems of chapter 5. For simplicity, only prepending is presented: Having a threaded witness $l = \langle c_1, \ldots, c_k \rangle$ and a context c, it can be decided whether $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded witness without checking the threaded witness properties of l':

Theorem 8.1 (Prepending to a Threaded Interprocedural Witness)
Let sequence $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness. Then $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded interprocedural witness, iff

$$\forall 1 \leqslant i \leqslant k : \theta(c_i) \in \Pi(\theta(c)) \vee c \rightharpoonup_R^+ c_i$$

Proof 8.1
Follows from definition 8.2.

Definition 8.3
Let $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness and thread $t \in \Theta$. $F(l, t)$ is defined as:

$$F(l, t) \quad = \quad \begin{cases} c_i & \exists i : \theta(c_i) \notin \Pi(t) \wedge \forall 1 \leqslant j < i : \theta(c_j) \in \Pi(t) \\ \bot & \text{otherwise} \end{cases}$$

The result is basically the first context of witness l relevant for the execution of thread t (if such a context exists).

Theorem 8.2 (Simplified Prepending)
Let sequence $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness. Then sequence $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded interprocedural witness, iff

$$F(l, \theta(c)) = \bot \vee c \rightharpoonup_R^+ F(l, \theta(c))$$

Proof 8.2
Proof by contradiction. Assume:

$$\begin{aligned} & F(l, \theta(c)) \neq \bot \wedge c \not\rightharpoonup_R^+ F(l, \theta(c)) \\ \Longleftrightarrow\ & \exists 1 \leqslant i \leqslant k : c_i = F(l, \theta(c)) \wedge c \not\rightharpoonup_R^+ c_i \end{aligned}$$

From definition 8.3 follows $\theta(c_i) \notin \Pi(\theta(c))$. Altogether:

$$\exists 1 \leqslant i \leqslant k : \theta(c_i) \notin \Pi(\theta(c)) \wedge c \not\rightharpoonup_R^+ c_i$$

However, this is a contradiction to theorem 8.1 because l and l' are threaded witnesses and therefore theorem 8.2 holds. □

Having a threaded witness $l = \langle c_1, \ldots, c_k \rangle$ and an edge $n \rightharpoonup n'$ with $T(c_1) = n'$ and $T(c) = n$, it can be decided whether $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded witness without checking the threaded witness properties of l':

Theorem 8.3 (Prepending an Edge)
Let $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness. If an edge $T(c) \rightharpoonup T(c_1)$ exists, then $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded interprocedural witness.

Proof 8.3
Three possibilities for $T(c) \rightharpoonup T(c_1)$ exist: traditional control flow, call or return edges. Let $n = T(c)$ and $n_1 = T(c_1)$. From $n \rightharpoonup n_1 \in E_t^*$ follows $\theta(n) = \theta(n_1)$ and $\theta(c) = \theta(c_1)$. Using theorem 8.1:

$$\begin{array}{ll} & \left(\forall 1 < i \leqslant k : \theta(c_i) \in \Pi(\theta(c_1)) \vee c_1 \rightharpoonup_R^+ c_i\right) \wedge c \rightharpoonup_R^+ c_1 \\ \Longrightarrow & \left(\forall 1 < i \leqslant k : \theta(c_i) \in \Pi(\theta(c)) \vee c_1 \rightharpoonup_R^+ c_i\right) \wedge c \rightharpoonup_R^+ c_1 \\ \Longrightarrow & \left(\forall 1 < i \leqslant k : \theta(c_i) \in \Pi(\theta(c)) \vee c \rightharpoonup_R^+ c_i\right) \wedge c \rightharpoonup_R^+ c_1 \\ \Longrightarrow & \forall 1 \leqslant i \leqslant k : \theta(c_i) \in \Pi(\theta(c)) \vee c \rightharpoonup_R^+ c_i \end{array}$$

which is theorem 8.1 itself. □

The concept of threaded interprocedural witnesses is needed to define slices in concurrent interprocedural programs based on the threaded interprocedural PDG, which is presented next.

8.3 The Threaded Interprocedural PDG

Control and data dependence is computed for the different threads, assuming that every thread is independent from all the others. This results in a set of non-connected interprocedural program dependence graphs—one IPDG for each thread. The next step is to compute the interference dependence between threads. This is done exactly as in section 5.2.3: If a variable is defined in one thread and referenced in another parallel executing thread, an interference dependence edge is generated between the corresponding nodes.

The *threaded interprocedural PDG (tIPDG)* is the union of the IPDGs for each thread, connected by the interference dependence edges. The control flow, call and return edges from the ICFG are also present in the tIPDG. The usual call edges in IPDGs have to be distinguished from the control flow call edges. Therefore, control flow edges will be denoted with $\rightharpoonup$ and dependence edges with $\rightharpoondown$. The different types of dependence edges are distinguished by a label d in $\overset{d}{\rightharpoondown}$, e.g. dd for a data dependence edge $\overset{dd}{\rightharpoondown}$.

Definition 8.4 (Threaded Interprocedural Realizable Paths)
A path $P = \langle n_1, \ldots, n_k \rangle$ in a tIPDG is a *threaded interprocedurally realizable path*, iff

1. the path contains no edge from the control flow graph (control flow edges, control flow call or return edges):

$$n_1 \overset{d_1}{\rightharpoondown} \ldots \overset{d_{k-1}}{\rightharpoondown} n_k,$$

2. every sub-path without an interference edge is an interprocedurally realizable path in the containing IPDG:

$$\forall 1 \leqslant i < k, i < j \leqslant k : (\forall i \leqslant m < j : d_m \neq id) \Rightarrow \langle n_i, \ldots, n_j \rangle \text{ is interprocedurally realizable,}$$

and

3. a threaded interprocedural witness W exists and corresponds to path P:

$$\exists W = \langle c_1, \ldots, c_k \rangle : P = \langle T(c_1), \ldots, T(c_k) \rangle$$

If a node n is reachable from n' via a threaded interprocedurally realizable path, it is denoted as $n' \rightarrow_R^\star n$.

Similar to the previous chapters, slicing is now defined in terms of threaded interprocedurally realizable paths:

Definition 8.5
The (*backward*) *slice* $S_\theta(n)$ of a tIPDG at a node n consists of all nodes m on which n depends transitively via a threaded interprocedurally realizable path:

$$S_\theta(n) = \{m \mid m \rightarrow_R^\star n\}$$

Definition 8.6
The *slice* $S_\theta(c)$ in a tIPDG for a context c consists of all contexts c' for which $T(c)$ depends transitively on $T(c')$ via a threaded interprocedurally realizable path:

$$S_\theta(c) = \{c' \mid \exists P = \langle T(c'), \ldots, T(c) \rangle : T(c') \rightarrow_R^\star T(c)\}$$

with the additional constraint that P's threaded interprocedural witness W is $W = \langle c', \ldots, c \rangle$. This definition stays the same for a slice $S(c)$ restricted to an IPDG.

These definitions of threaded interprocedural paths and slices are not computable because the set C of possible contexts is infinite when recursion exists in the analyzed program. However, the next section will show a way to make the set of possible contexts finite and thus make slices computable.

8.4 Slicing the tIPDG

For reachability the number of recursive calls is irrelevant and cycles in the ICFG can be folded into a single node representing the strongly connected region. There are two sources for cycles in the ICFG:

1. Loops in the program cause cycles only in the intraprocedural part of the CFG and have no 'stacking' influence on the number of possible contexts.

2. Recursive calls cause cycles over call and return edges. If such cycles are replaced by a single node, reachability is not realizable interprocedurally because call and return edges are not matched correctly.

The cycles are replaced by a two pass approach to keep the matching of call and return edges intact. The first pass finds and folds strongly connected components in the ICFGs consisting of control flow and call edges but ignores return edges. The second pass finds and folds cycles in the resulting graph consisting of control flow and return edges (now ignoring call edges). This replacement generates a function ρ that maps nodes from the original ICFG $G = (N^\star, E^\star, n_0^s, n_0^e)$ to the new graph $\overline{G} = (\overline{N}^\star, \overline{E}^\star, n_0^s, n_0^e)$:

$$\rho(n \in N^\star) = \begin{cases} n & \text{if } \forall n \xrightarrow{d_1} \dots \xrightarrow{d_k} n : \exists i,j : d_i = \text{call} \wedge d_j = \text{return} \\ n' \notin N^\star & \text{otherwise} \end{cases}$$

$$\rho(n) \neq n \;\Leftrightarrow\; \forall n \xrightarrow{d_1} n_1 \dots n_{k-1} \xrightarrow{d_k} n : \big(\forall i,j : d_i \neq \text{call} \wedge d_j \neq \text{return}\big) \Rightarrow \big(\forall i : \rho(n_i) = \rho(n)\big)$$

$$\overline{N}^\star = \{\rho(n) \mid n \in N^\star\}$$

Every interprocedurally realizable path in the resulting graph has a corresponding realizable path in the original graph. Due to unrealizable matchings of call and return edges, there are still cycles in the graph.[1]

Based on the newly created graph $\overline{G}$ the set of contexts and the 'reaches' relation between contexts are redefined: The execution state $c \in \overline{C}_t$ of thread t is a stack $c = n_0 \dots n_k$ of nodes $n_i \in \overline{N}_t^\star$ with the topmost node $T(c) = n_0$. The 'pop' function P is defined as before: $P(c) = n_1 \dots n_k$.

Definition 8.7
A context $c_1 \in \overline{C}_t$ *reaches directly* another context $c_2 \in \overline{C}_t$: $c_1 \rightharpoonup_R c_2$, iff one of the following alternatives holds

1. an edge $n_1 \rightharpoonup n_2 \in E_t^\star$ exists with $n_1 \rightharpoonup n_2 \notin E_t^C$ and where $T(c_1) = \rho(n_1) \wedge T(c_2) = \rho(n_2) \wedge P(c_1) = P(c_2)$
   (corresponding edge in the CFG of a procedure exists),

2. a call edge $n_1 \rightharpoonup n_2 \in E_t^C$ exists and

   (a) $T(c_1) = \rho(n_1) \wedge T(c_2) = \rho(n_2)$,

   (b) $T(c_1) \neq T(c_2) \rightarrow c_1 = P(c_2)$, and

   (c) $T(c_1) = T(c_2) \rightarrow c_1 = c_2$

   (corresponding call edge exists), or

[1] There are even cycles due to realizable paths, for example "`f(); f();`" creates a cycle from the entry to "`f`" to itself. However, those paths are not *matched* paths as defined in section 6.2.

3. a return edge $n_1 \rightharpoonup n_2 \in E_t^C$ exists and
   (a) $T(c_1) = \rho(n_1) \wedge T(c_2) = \rho(n_2)$,
   (b) $T(c_1) \neq T(c_2) \rightarrow$
   $P(P(c_1)) = P(c_2) \wedge \exists n_3 \rightharpoonup n_2 \in (E_t^\star - E_t^C) : \rho(n_3) = T(P(c_1))$,
   and
   (c) $T(c_1) = T(c_2) \rightarrow c_1 = c_2$

   (corresponding return edge exists, leading to a return node which matches the last call node.)

A context c *reaches* another context c': $c \rightharpoonup_R^+ c'$, iff a series of contexts $c_1, \ldots, c_n$ exists with $c = c_1 \wedge c' = c_n \wedge \forall_{1 \leqslant i < n} c_i \rightharpoonup_R c_{i+1}$. The set of possible contexts for ICFG $G_t = (N_t^\star, E_t^\star, n_t^s, n_t^e)$ is $\overline{C}_t = \{c' \mid \rho(n_t^s) \rightharpoonup^+ c' \vee c' = \rho(n_t^s)\}$.

With this definition $\overline{C}_t$ must be finite because traversing call edges does not 'stack' call nodes inside recursive cycles.

Observation 8.1 A context $c_1 = n_0 \ldots n_k$ reaches another context $c_2 = m_0 \ldots m_l$ ($c_1, c_2 \in C_t$) in terms of definition 8.1, iff $\overline{c}_1 = \overline{n}_0 \ldots \overline{n}_{k'}$ reaches $\overline{c}_2 = \overline{m}_0 \ldots \overline{m}_{l'}$ in terms of definition 8.7 ($\overline{c}_1, \overline{c}_2 \in \overline{C}_t$), where $\rho(n_0) \ldots \rho(n_k) = \overline{n}_0{}^{i_1} \ldots \overline{n}_{k'}{}^{i_{k'}} \wedge \rho(m_0) \ldots \rho(m_l) = \overline{m}_0{}^{j_1} \ldots \overline{m}_{l'}{}^{j_{l'}}$.

Based on this observation, definition 8.7 can now be used in the definitions and theorems about threaded interprocedural witnesses. Hence it is decidable and computable whether a path in a tIPDG is threaded interprocedurally realizable. Thus, slices $S_\theta(c)$ can be computed using definition 8.7, denoted as $\overline{S}_\theta(c)$ and $\overline{S}(c)$.

A naive implementation would enumerate the possible paths to the slicing criterion node and check them to be threaded interprocedurally realizable paths. This is way too expensive, so approaches from previous chapters are combined to create a more efficient approach for slicing based on tIPDGs, shown in algorithm 8.1 on page 114. There, the extraction of the ith element c_i in a tuple $\Gamma = (c_1, \ldots, c_n)$ is denoted by $\Gamma[i]$. The substitution of the ith element c_i in a tuple $\Gamma = (c_1, \ldots, c_n)$ with a value x will be denoted as $[x/_i]\Gamma$.

Its basic idea is the coding of possible execution states of all threads in tuples $(c_1, \ldots, c_{|\Theta|-1})$, where the c_i are contexts (in the tIPDG). The value c_i represents that it is still possible to reach context c_i in thread θ_i (a value of $\bot$ does not restrict the state of execution). This is used to keep track of the context c_i where thread θ_i has been left by following an interference edge. If another interference edge is followed back into the thread at node q, the reachability of c from the contexts c' of q ($q = T(c')$) can be checked. It assures that paths over interference edges are always threaded witnesses in the tICFG. This is the reason why the control flow edges have to be kept in the tIPDG. Reachability can be computed without checking threaded witness properties, because the simplified prepending theorem 8.2 allows checking based on the last reached context in a thread (the first element of a threaded witness relevant to the thread).

The algorithm keeps a worklist of pairs of contexts and state tuples that have to be examined. For computation of the slice $\overline{S}(c)$ inside a thread (not shown in algorithm 8.1 on the following page), every edge reaching the top node of the current context is examined and handled depending on its type. A new pair consisting of the new context and the modified state tuple is inserted into the worklist. According to theorem 8.3 this is done without checking the threaded witness property.

Interference edges are ignored while computing $\overline{S}(c)$, they are handled explicitly in algorithm 8.1 on the next page. An interference dependence edge may only be considered if the (old) context relevant to the source node thread is reachable from a context at the source node in the tICFG (all examined paths are still threaded witnesses). Then, the new pair with the updated state tuple is inserted into the worklist. The resulting slice is the set of nodes that is constructed out of the first elements of the inserted pairs.

Algorithm 8.1 on the following page contains two bottlenecks that are eased in the following: the computation of $\overline{C}_t(n)$ and $\overline{S}(c)$. The idea of subsuming call strings in section 7.3.4 is used to build subsuming contexts: With subsuming contexts, $\overline{C}_t(n)$ just contains one element $n\diamond$ that subsumes all contexts $nn_1 \dots n_k \in \overline{C}_t(n)$. Now, the third constraint of definition 8.7 has an alternative:

> A return edge $n_1 \rightharpoonup n_2 \in E_t^C$ exists and either constraint 3 of definition 8.7 holds or
>
> 1. $c_1 = \rho(n_1)\diamond \wedge T(c_2) = \rho(n_2)$ and
> 2. $T(c_1) = T(c_2) \rightarrow c_1 = c_2$
>
> (return edge exists that returns to a node matching $\diamond$ automatically).

In particular, traversal of a return edge $n_1 \rightharpoonup n_2$ from a context $\rho(n_1)\diamond$ leads to a context $\rho(n_2)\diamond$.

The second bottleneck is the computation of slice $\overline{S}(c)$. Section 7.4 showed that the computation based on explicit context may suffer from combinatorial explosion. Because the computation is restricted to an IPDG, summary edges can be generated and used for more efficient slicing. Instead of computing an expensive slice $S(c)$, a traditional slice using summary edges is computed. In this slice, all nodes are identified with at least one incoming interference dependence edge. For each of these nodes, a chop is computed between the node and the original slicing criterion. This chop is truncated non-same-level which can be computed efficiently (see section 10.2.6). Now, only along nodes in this chop the more expensive slice $S(c)$ is computed. This is much more efficient because a far smaller set of nodes is visited. In algorithm 8.2 on page 115, only the modifications to algorithm 8.1 on the following page implementing this improvement are shown.

Algorithm 8.1 Slicing Algorithm for tIPDGs, $\overline{S}_\theta$

Input: The tIPDG $G = (N^\Pi, E^\Pi, S^\Pi, X^\Pi)$
The slicing criterion $s \in N^\Pi$
Output: The slice S, a set of nodes of the tIPDG

Let $\overline{C}_t(n)$ be a function which returns the set of possible contexts for a node n in IPDG G_t.

Initialize the worklist with an initial state tuple:
$\Gamma = (n^e_{\theta_1}, \ldots, n^e_{\theta_{|\Theta|}})$, *every thread is at its end node*
$W = \{(c, \Gamma') \mid t = \theta(s) \wedge c \in \overline{C}_t(s) \wedge \Gamma' = [c/_t]\Gamma\}$
$M = W$, *Mark the contents of the worklist*

repeat
  Remove the next element $w = (c, \Gamma)$ from W:
  $W = W - \{w\}$
  $S = S \cup \{n \mid \rho(n) = T(c)\}$
  Compute a slice $\overline{S}(c)$ for c in the IPDG G_t.
  $I = \{c' \in \overline{S}(c) \mid \exists n \in N^\Pi : n \xrightarrow{id} T(c')\}$
  The top node has ⩾1 incoming interference dependence edges.
  foreach $i \in I$ **do**
    foreach edge $n \xrightarrow{id} n'$, $\rho(n') = T(i)$**do**
      $t = \theta(T(i))$
      $t' = \theta(n)$
      $\Gamma' = [i/_t]\Gamma$
      $\Gamma'' = [c/_{t'}]\Gamma'$
      $C' = \{c' \mid c' \in \overline{C}_{t'}(n) \wedge c' \rightharpoonup^+_R \Gamma[t']\}$
      foreach $w' \in \{(c'', \Gamma'') \mid c'' \in C'\}$ **do**
        if $w' \notin M$ **then**
          $M = M \cup \{w'\}$
          $W = W \cup \{w'\}$
until $W = \emptyset$
return S

Algorithm 8.2 Improved Slicing Algorithm for tIPDGs

$\vdots$
foreach $t \in \Theta$ **do**
  Generate summary edges and transform IPDG G_t into a SDG
$\vdots$
repeat
  Remove the next element $w = (c, \Gamma)$ from W:
  $W = W - \{w\}$.
  $N_c = \{n \mid \rho(n) = T(c)\}$
  $S = S \cup N_c$
  Compute a slice $S(n')$ for one $n' \in N_c$ in the SDG G_t.
  $I_N = \{n_1 \in S(n') \mid \exists n_2 \in N^{\Pi} : n_1 \xrightarrow{id} n_1\}$
  Nodes in the slice with $\geqslant$1 incoming interference dependence edges.
  foreach $i_N \in I_N$ **do**
    Compute a truncated non-same-level chop $C^{TN}(i_N, n')$
    Compute a slice $\overline{S}(c)$ only along nodes in $C^{TN}(i_N, n')$.
    foreach $i \in \overline{S}(c) \mid T(c) = \rho(i_N)$ **do**
      foreach incoming interference dependence edge $n \xrightarrow{id} T(i)$**do**
        $\vdots$
until $W = \emptyset$
return S

8.5 Extensions

The simple model of concurrency is similar to the Ada concurrency mechanism, except for synchronization. The extensions from chapter 5 to enable synchronization can also be applied here. For example, the send/receive-style communication can also be used to model the Ada-style rendezvous. To improve the precision of the interference dependence edges, the 'may-happen-in-parallel (MHP)' information in [NA98, NAC99] can be used.

The send/receive-style synchronization can also be used to simulate a cobegin/coend parallelism within the presented model: The branches of the cobegin/coend statement are transformed into single threads. At cobegin, synchronization statements that start the newly created threads are introduced, and at coend, synchronization statements are introduced to wait until the newly created threads have finished. This requires a modified Π-function—the earlier trivial definition of Π has never been exploited in a proof and thus, the presented theorems are not weakened.

To allow code sharing between threads, the duplication of the shared code is sufficient. Every thread will then have its own instance of the shared code.

The synchronization extensions can be used to represent a concurrency model where the different threads are allowed to be started and stopped from other threads. This is similar to the concurrency model of Java. However, in Java, threads are generated dynamically, which cannot be represented in the simple concurrency model. Therefore, data flow analysis is needed to compute a conservative approximation of the set of possible threads. The static set can be represented in the simple concurrency model with synchronization extensions, enabling more precise slicing of concurrent Java programs.

8.6 Conclusions and Related Work

All previous approaches known to the author to slice concurrent programs precisely rely on the inlining of called procedures to slice concurrent interprocedural programs ([NR00, CX01a], see chapter 5 for a discussion). The presented approach is the first that is able to slice concurrent *recursive* interprocedural programs accurately.

See section 5.5 for more related work.

The presented approach is precise up to threaded interprocedurally realizable paths. The undecidability result in [Ram00] does not apply to the simple model as it does not contain synchronization. Our approach is not optimal in terms of [MOS01]—their undecidability results apply to the model used here: It is possible that a thread kills a definition in a different thread. Within the presented approach we have ignored such killing. To explore how much precision is lost through this approach, the results from experiment 1 of section 7.4.3 for sequential execution can be used: There, a definition is never killed by another definition, which adds more data dependences to the IPDG. On average, the generated slices are only 5% larger than before. This cannot be worse in the concurrent case, and thus, the ignoring of killing for interference must have a similar effect. With this result we argue that ignoring the killing effects of concurrently executing threads has only a small influence on precision.

Part III

Applications

Chapter 9

Visualization of Dependence Graphs

The program dependence graph itself and the computed slices within the program dependence graph are results that should be presented to the user if not used in following analyses. As graphical presentations are often more intuitive than textual ones, a graphical visualization of PDGs is desirable. This chapter describes how a layout for the PDGs can be generated to enable an appealing presentation. However, experience shows that the graphical presentation is less helpful than expected and a textual presentation is superior. Therefore section 9.2 contains an approach to textually present slices in fine-grained PDGs in source code.

9.1 Graphical Visualization of PDGs

Layout of graphs is a widely explored research field with many general solutions available in many graph drawing tools. Some of these tools have been tested to lay out PDGs:

daVinci a visualization system for generating high-quality drawings of directed graphs [FW95].

VCG (Visualization of Compiler Graphs) is targeted at the visualization of graphs that typically occur as data structures in programs [San95].

dot is a widely-used tool to create hierarchical layouts of directed graphs [KN96, GNV88].

The experience with these tools has been disappointing. The resulting layouts were visually appealing but unusable, as it was not possible to comprehend the graph. The reason is that the viewer has no cognitive mapping back to the source code, which is the representation he is used to. The user expects a

representation that is either similar to the abstract syntax tree (as a presentation of the syntactical structure), or a control-flow-graph like presentation.

In a second experiment the layout was influenced as much as possible to generate a presentation that enables the viewer to map the graph structure to the syntactical structure based on the control-dependence subgraph. The control dependence subgraph is tree-like, and in structured programs it resembles the abstract syntax tree. The possibilities of influencing the layout were quite different in the evaluated tools, where *dot* had the greatest flexibility. However, it was not possible to manipulate the layout in a way that generated comprehensible presentations.

9.1.1 A Declarative Approach to Layout PDGs

As the general algorithmic approach to layout PDGs had failed, a declarative approach has been implemented. It is based on the following observations:

1. The control-dependence subgraph is similar to the structure of the abstract syntax tree.
2. Most edges in a PDG are data dependence edges. Usually, a node with a variable definition has more than one outgoing data dependence edge.

The first observation leads to the requirement to have a tree like layout of the control dependence subgraph with the additional requirement that the order of the nodes in a hierarchy level should be the same as the order of equivalent statements in the source code. The second observation leads to an approach where the data dependence edges should be added to the resulting layout without modifying it. As most data dependence edges would now cross large parts of the graph, a Manhattan layout is adequate. This enables an orthogonal layout of edges with fixed start and end points.

Layout of the Control Dependence Graph

Instead of a specialized tree layout, an available implementation of the Sugiyama algorithm [STT81] has been reused, consisting of three phases:

1. The nodes are arranged into a vertical hierarchy where the number of levels crossed by edges is minimized.
2. Nodes in a horizontal level of the hierarchy are ordered to minimize the number of edge crossings.
3. The coordinates of the nodes are calculated such that long edges are as straight as possible.

Because the control dependence graph is mainly a tree, phase one is simple and very fast. Phase two has been replaced completely as the order of nodes is defined by the statement order in the source code and is not allowed to change. In Phase three the original algorithm has been extended with a “rubber-band” improvement of [San95].

Layout of Data Dependence Edges

The layout of data dependence edges is basically a routing between fixed start and end points. As most edges in a PDG are data dependence edges, the routing must be fast and efficient. Based on the observations at the beginning of this section, the routing is done with the following principles:

1. The route of an edge is separated into three parts:
   - A vertical segment between the start node and the level above the end node,
   - a horizontal segment to the position of the end node, and
   - a vertical end segment to the end node.
2. Edges leaving the same node share the same first segment.

The layout of the three segments is done independently: The starting vertical segment is laid out straight if a node that would be crossed can be pushed aside. If this is not possible, the segment is split to circumvent the node. The horizontal segment is laid out with a sweep-line algorithm to minimize the space that routes passing a level take. The third segment is routed to its entry point into the end node.

Presentation of System Dependence Graphs

The presented approach to laying out PDGs has been implemented in a tool that visualizes system dependence graphs [Ehr96]. Starting from a graphical presentation of the call graph, the user can select procedures and visualize their PDGs. Through selection of nodes, slices can be calculated and are visualized through inverted nodes in the laid out PDGs.

Example 9.1: A visualization can be seen in Figure 9.1 on the next page, where a slice is visualized in the dependence graph and source code through highlighting.

Navigation

The user interface for the visualized graph contains extensive navigation aids:

- Nodes and edges can be searched for by their attributes.
- Edges can be followed forward or backward.
- A set of nodes can be expanded with all nodes reachable by traversing one edge.
- The visualization can be focused on each node of a node set by stepping through the set.

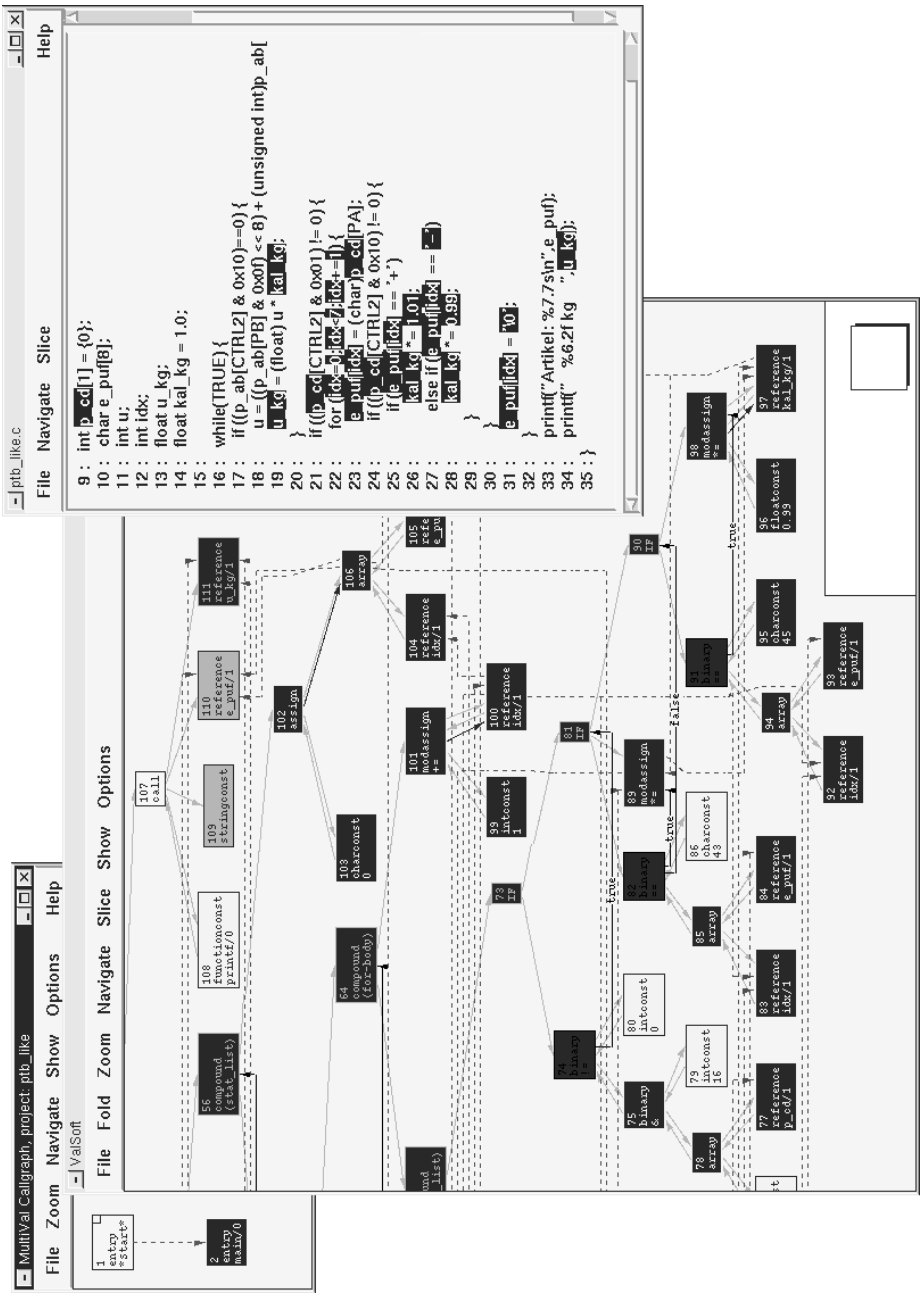


Figure 9.1: The graphical user interface

- Node sets can be saved and restored.
- Two node sets can be combined to a new node set by set operations.
- Node sets can be "filtered" through external tools; slicing and chopping is implemented that way.
- To compress the visualized graph, node sets can be folded to a single node.

9.1.2 Evaluation

Experience with the presented tool shows that the layout is very comprehensible for medium sized procedures and the user easily keeps a cognitive map from the structure of the graph to the source code and vice versa. This mapping is supported by the possibility of switching to a source code visualization of the current procedure and back: Sets of nodes marked in the graph can be highlighted in the source code and marked regions in the source code can be highlighted in the graph (see Figure 9.1 on the facing page for an example). Together with the navigation aids, it is easy to see what statements influence which other statements and how.

However, experience has shown that the graphical visualization is still too complex. For larger procedures the number of nodes and edges is too big and it takes very long to follow edges across multiple pages by scrolling.

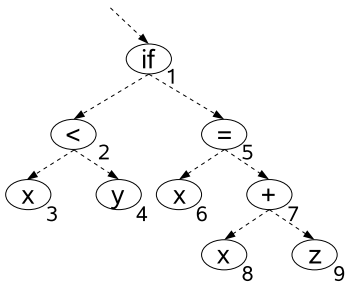
9.2 Textual Visualization of Slices

The graphical visualization presented in the previous section has been found to be overly complex for large programs and non-intuitive for visualization of slices. Therefore the graphical visualization has been extended with a visualization in source code. Because of the fine-grained structure, this causes a non-trivial projection of nodes onto source code. The technique presented in this section not only visualizes slices (and chops) in source code, but any set of nodes.

The source code is represented as a continuous sequence of characters, such that any piece of source code can be represented as an interval in that sequence. Such an interval is described by a file/row/column position for start and end. During parsing while constructing the abstract syntax tree, every node is attributed with an interval. During analysis, the nodes of the abstract syntax tree are transformed into nodes in the program dependence graph, which still have the source code interval attribute.

Example 9.2: Consider the following example:

```
if (x < y) {
  x = x + z;
}
```



1 : $[1-15]$
2 : $[3-7]$
3 : $[4-4]$
4 : $[6-6]$
5 : $[9-14]$
6 : $[9-9]$
7 : $[11-13]$
8 : $[11-11]$
9 : $[13-13]$

Figure 9.2: A small code fragment with position intervals

This program is represented as a sequence of characters, each character having a position, as shown in the following table (whitespace is ignored):

i	f	(	x	<	y	)	{	x	=	x	+	z	;	}
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

During transformation of this fragment in its program dependence graph the nodes are attributed with the position interval as shown in Figure 9.2.

If the visualization would just highlight the intervals of the nodes, the result would be catastrophic: any visualization of a set that includes node 1 will highlight the complete fragment. The problem here is the nesting of intervals: an interval $r = [x_r - y_r]$ is nested in an interval $q = [x_q - y_q]$, written as $r \subseteq q$, if $x_q < x_r < y_r < y_q$. The intervals generated during construction of the abstract syntax tree have two properties:

1. All intervals are properly nested: $\forall r, q : r \subseteq q \vee q \subseteq r$.
2. All intervals are unique: $\forall r, q, r \neq q : x_r \neq x_q \vee y_r \neq y_q$.

If follows $\forall r, q : r \subset q \vee q \subset r$.

Any position should only be highlighted if the smallest enclosing interval belongs to a node included in the highlighted set. An interval r is the smallest enclosing interval of a position x, if there is no interval q such that q encloses x ($x \in q$) and r ($r \supset q$). Therefore the interval attribute of the nodes is changed to a set of attributes: If a node has an interval q that encloses an interval r of a different node, the interval r is removed by splitting the original interval r: Let $r = [x_r - y_r]$ be nested in interval $q = [x_q - y_q]$, the new interval is split into two new intervals $q_1 = [x_q - x_r[$ and $q_2 =]y_r - y_q]$. If this transformation is applied thoroughly, every interval will be unique.

Example 9.3: The resulting intervals for the example are shown in Figure 9.3 on the facing page.

The nodes are now mapped to non-overlapping intervals. To highlight any set of nodes, the sets of intervals of the nodes are joined and only the intervals in the resulting set are highlighted.

$1: [1-2], [15-15]$
$2: [3-3], [5-5], [7-7]$
$3: [4-4]$
$4: [6-6]$
$5: [10-10], [14-14]$
$6: [9-9]$
$7: [12-12]$
$8: [11-11]$
$9: [13-13]$

Figure 9.3: Intervals after transformation

Example 9.4: The fragment below shows the visualization of a backward slice for node 9, which consists of the node set $\{1, 2, 3, 4, 5, 7, 9\}$:

```
if (x < y) {
  x = x + z;
}
```

The next example shows the node set $\{1, 5, 6\}$ highlighted.

```
if (x < y) {
  x = x + z;
}
```

We have presented only the basic visualization techniques. However, visualization of standard slices has only a limited benefit in program comprehension as too much data is presented at once. Therefore, the next chapter will present some more focused slicing techniques together with their visualization.

9.3 Related Work

9.3.1 Graphical Visualization

In [Bal01] the same problems with visualizing dependence graphs are reported and a decomposition approach is presented: groups of nodes are collapsed into one node. The result is a hierarchy of groups, where every group is visualized independently. Three different decompositions are presented: The first decomposition is to group the nodes belonging to the same procedure together, the second is to group the nodes belonging to the same loop and the third is a combination of both. The result of the function decomposition is identical to the visualization of the call graph and the PDGs of the procedures presented in Section 9.1.1.

The CANTO environment [AFL+97] has a visualization tool PROVIS based on dot which can visualize PDGs (besides other graphs). Again, problems with

excessively large graphs are reported, which are omitted by only visualizing the subgraph which is reachable from a chosen node via a limited number of edges.

ChopShop [JR94b, JR94a] is a tool to visualize slices and chops, based on highlighting text (in emacs) or laying out graphs (with dot and ghostview). It is reported that even the smallest chops result in huge graphs. Therefore, only an abstraction is visualized: normal statements (assignments) are omitted, procedure calls of the same procedure are folded into a single node and connecting edges are attributed with data dependence information.

The decomposition slice visualization of Surgeon's Assistant [Gal96, GO97] visualizes the inclusion hierarchy of decomposition slices as a graph using VCG [San95].

An early system with capabilities for graphical visualization of dependence graphs is ProDAG [ROMA92]. Another system to visualize slices is [DKN01].

9.3.2 Textual Visualization

Every slicing tool visualizes its results directly in the source code. However, most tools are line based, highlighting only complete lines. CodeSurfer [AT01] has textual visualization with highlighting parts of lines, if there is more than one statement in a line. The textual visualization includes graphical elements like pop-ups for visualization and navigation along e.g. data and control dependence or calls. Such aids are necessary as a user cannot identify relevant dependences easily from source text alone. Such problems have also been identified by Ernst [Ern94] and he suggested similar graphical aids. However, his tool, which is not restricted to highlighting complete lines, does not have such aids and offers depth-limited slicing instead (see Section 10.1).

Steindl's slicer for Oberon [Ste98, Ste99a, Ste99b] also highlights only parts of lines, based on the individual lexical elements of the program.

SeeSlice [BE94] is a more advanced tool for visualizing slices. Files and procedures are not presented through source code but with an abstraction representing characters as single pixels. Files and procedures that are not part of computed slices are folded, such that only a small box is left. Slices highlight the pixels corresponding to contained elements.

CodeSurfer [AT01] also has a project viewer, which has a tree-like structural visualization of the SDG. This is useful for seeing "hidden" nodes, such as nodes that do not correspond to any source text.

Chapter 10

Making Slicing more Focused

In the last chapter we have seen that slicing alone does not meet the expectations. Usually, slices contain to much data to be comprehensible. Griswold stated in [Gri01] the need for "slice explainers" that answer the question why a statement is included in the slice and the need for "filtering". Chapter 13 will present a solution for the first need while this chapter contains various "filtering" approaches to slicing.

10.1 Distance-Limited Slices

One of the problems in understanding a slice for a criterion is to decide why a specific statement is included in that slice and how strong the influence of that statement is onto the criterion. A slice cannot answer these questions as it does not contain any qualitative information. Probably the most important attribute is *locality*: Users are more interested in facts that are near the current point of interest than on those far away. A simple but very useful aid is to provide the user with navigation along the dependences: For a selected statement, show all statements that are directly dependent (or vice versa). Such navigation is central to the VALSOFT system (chapter 14) or to CodeSurfer [AT01].

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

Definition 10.1 (Distance-Limited Slice)
The distance-limited *slice* $S(C, k)$ of a PDG for the slicing criterion nodes of set C consists of all nodes on which a node $n \in C$ (transitively) depends on a realizable path consisting of at most k edges:

$$S(C, k) = \{m \in N \mid p = m \rightharpoonup_R^\star n \wedge n \in C \wedge p = \langle n_1, \ldots, n_l \rangle \wedge l < k\}$$

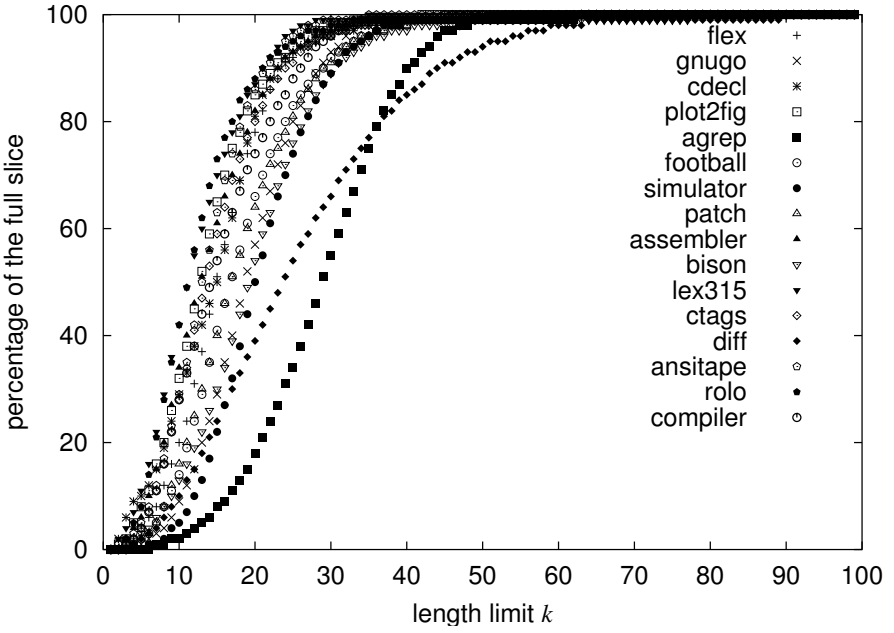


Figure 10.1: Evaluation of length-limited slicing

A node m is said to have a distance $l = d(m, n)$ from a node n, if a realizable path from m to n consisting of l edges exists and no other path with fewer edges exists:

$$d(m, n) = \min(\{l \mid p = m \rightarrow_R^\star n \wedge p = \langle n_1, \ldots, n_l \rangle\})$$

An efficient distance-limited slicing algorithm is a modified version of the interprocedural slicing algorithm 7.1 on page 87. To omit a priority queue sorted by the actual distance, a breadth-first search is done where the worklist is implicitly sorted.

Figure 10.1 shows an evaluation: For a series of test cases the average size of 1000 length limited slices has been computed, where the length limit ranges from 1 to 100 (x-axis). The y-axis shows the reached percentage of the full slices. This evaluation shows that length limited slices behave quite similar and independent of the analyzed program.

A more fine-grained approach is to replace the counting of edges by summarizing *distances* that have been assigned to the edges. Such distances can be used to give different classes of edges different weights. For example, a node that is reachable by a data dependence edge might be considered nearer than a node that is reachable by a summary edge. If the worklist is replaced by a priority queue sorted by the current sum of distances, the previous algorithm is able to compute such distance-limited slices.

Distance-limited slices can be visualized with the techniques presented in the last chapter without any modification. Another possibility is to illustrate the distances from the (slicing) criterion for any node in the (possibly distance-limited) slice. The textual visualization from section 9.2 is therefore modified not only to highlight the nodes in the textual representation, but to give any source code fragment a color representing the distance of the equivalent nodes to the criterion. The slicing algorithm need not be changed to accommodate the distance computation—it is sufficient to remember the distance of a node during breadth-first search.

Example 10.1: Figure 10.2 on the next page shows an example visualization of a slice. A backward slice for variable `u_kg` in line 33 is displayed. Parts of the program that have a small distance to the slicing criterion are darker than those with a larger distance. With this presentation one can see that the initialization in lines 8–10 and 13–14 have a close influence on the criterion. It can also be seen that the first few lines of the loop (17–19) have a close influence and that the whole next if-statement has a varying influence, where lines 26 and 28 have the strongest effect. Such information would not be visible in a simple slice visualization.

10.2 Chopping

Slicing identifies statements in a program that may influence a given statement (the slicing criterion), but it cannot answer the question why a specific statement is part of a slice. A more focused approach can help: Jackson and Rollins [JR94b] introduced *Chopping*, which reveals the statements involved in a transitive dependence from one specific statement (the source criterion) to another (the target criterion). A chop for a chopping criterion (s, t) is the set of nodes that are part of an influence of the (source) node s onto the (target) node t. This is basically the set of nodes that lie on a path from s to t in the PDG.

Definition 10.2 (Chop)
The *chop* $C(s, t)$ of an IPDG $G = (N, E)$ from the source criterion $s \in N$ to the target criterion $t \in N$ consists of all nodes on which node t (transitively) depends via an interprocedurally realizable path from node s to node t:

$$C(s,t) = \{n \in N \mid p = s \rightarrow_R^\star t \wedge p = \langle n_1, \ldots, n_l \rangle \wedge \exists i : n = n_i\}$$

Again, we can extend the chopping criteria to allow sets of nodes: The chop $C(S, T)$ of an IPDG from the source criterion nodes S to the target criterion nodes T consists of all nodes on which a node of T (transitively) depends via an interprocedurally realizable path from a node of $S \subseteq N$ to the node in $T \subseteq N$:

$$C(S,T) = \{n \in N \mid p = s \rightarrow_R^\star t \wedge s \in S \wedge t \in T \wedge p = \langle n_1, \ldots, n_l \rangle \wedge \exists i : n = n_i\}$$

```
01: const unsigned TRUE = 1;
02: const unsigned CTRL2 = 0;
03: const unsigned PB = 0;
04: const unsigned PA = 1;
05: void printf();
06: void main()
07: {
08:    int p_ab[2] = {0, 1};
09:    int p_cd[1] = {0};
10:    char e_puf[8];
11:    int u;
12:    int idx;
13:    float u_kg;
14:    float kal_kg = 1.0;
15:
16:    while(TRUE) {
17:       if ((p_ab[CTRL2] & 0x10)==0) {
18:          u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
19:          u_kg = (float) u * kal_kg;
20:       }
21:       if ((p_cd[CTRL2] & 0x01) != 0) {
22:          for (idx=0;idx<7;idx++) {
23:             e_puf[idx] = (char)p_cd[PA];
24:             if ((p_cd[CTRL2] & 0x10) != 0) {
25:                if (e_puf[idx] == '+')
26:                   kal_kg *= 1.01;
27:                else if (e_puf[idx] == '-')
28:                   kal_kg *= 0.99;
29:             }
30:          }
31:          e_puf[idx] = '\0';
32:       }
33:       printf("Artikel: %7.7s\n   %6.2f kg    ",e_puf,u_kg);
34:    }
35: }
36:
```

Figure 10.2: Distance visualization of a slice

Jackson and Rollins restricted s and t to be in the same procedure and only traversed control dependence, data dependence and summary edges but not parameter or call edges. The resulting chop is called a *truncated same-level chop* C^{TS}; "truncated" because nodes of called procedures are not included. In [RR95] Reps and Rosay presented more variants of precise chopping. A *non-truncated* same-level chop C^{NS} is like the truncated chop but includes the nodes of called procedures. They also present truncated and non-truncated *non-same-level* chops C^{TN} and C^{NN} (which they call *interprocedural*), where the nodes of the chopping criterion are allowed to be in different procedures. The following focuses on *non-truncated same-level chopping* first.

10.2.1 Context-Insensitive Chopping

In the intraprocedural or context-insensitive case a chop for a chopping criterion (s, t) can basically computed by the intersection of a backward slice for t with a forward slice for s: $C(s, t) = S^F(s) \cap S^B(t)$. An algorithm would not use intersection because set operations are expensive for large sets like slices. Instead, algorithm 10.1 on the following page is a two-phase approach: The first phase computes the backward slice and in the second phase the forward slice is done, where only nodes that have been visited during the backward phase are considered. This algorithm will be called *context-insensitive chopping (CIC)*.

10.2.2 Chopping with Summary Edges

The precise context-sensitive chopping algorithm 10.2 on page 133 uses summary edges. It basically starts with an intraprocedural chop, which is done with algorithm 10.1 on the following page. There is a slight modification to that algorithm: It has to do a chop only inside one single procedure and is not allowed to traverse parameter or call edges (but it is required to traverse summary edges). This is a *truncated same-level* chop C^{TS}.

From the calculated chop all pairs of actual-in/-out nodes which are connected by a summary edge are extracted and the initial worklist is filled with the pairs of the corresponding formal-in/-out nodes. Now, for every pair in the worklist a new intraprocedural chop is generated and added to the starting chop. Again, all summary edges are extracted and the pairs of the corresponding formal-in/-out nodes are added to the worklist if they have not been added before. This is repeated as long as there are elements in the worklist. The now extended chop is the resulting precise interprocedural chop.

The algorithm generates a *same-level non-truncated* chop C^{SN}: both nodes of the chopping criterion have to be in the same procedure. This algorithm is called *context-sensitive chopping (CSC)*. The intraprocedural version (algorithm 10.1 on the next page) is a same-level *truncated* chop.

This algorithm can be optimized by computing and caching slices instead of using algorithm 10.1. An intraprocedural chop for (n', m') is now computed by intersecting slices: $C(n', m') = S^F(n') \cap S^B(m')$. However, every computed

Algorithm 10.1 Context-Insensitive Chopping

Input: $G = (N, E)$ the given IPDG
$(s, t) \in N \times N$ the given chopping criterion
Output: The context-insensitive chop

Initialization
$W_B = \{t\}$, *mark* t *as visited in the backward phase*
$B = \{t\}$
while $W_B \neq \emptyset$ *worklist is not empty* **do**
  $W_B = W_B / \{n\}$ *remove one element from the worklist*
  foreach $m \rightharpoonup n \in E$ **do**
    if $m \notin B$ m *is not yet marked* **then**
      mark m *as visited in the backward phase*
      $W_B = W_B \cup \{m\}$
      $B = B \cup \{m\}$
if $s \in B$ s *has been marked in the backward phase* **then**
  mark s *as visited in the forward phase*
  $W_F = \{s\}$
  $F = F \cup \{s\}$
  while $W_F \neq \emptyset$ *worklist is not empty* **do**
    $W_F = W_F / \{n\}$ *remove one element from the worklist*
    foreach $n \rightharpoonup m \in E$ **do**
      if $m \in B$ m *has been marked in the backward phase*
      and $m \notin F$ m *has not yet been marked in the forward phase* **then**
        mark m *as visited in the forward phase*
        $W_F = W_F \cup \{m\}$
        $F = F \cup \{m\}$
return F *the set of all in the forward phase visited nodes*

Algorithm 10.2 Chopping with Summary Edges

Input: $G = (N, E)$ the given SDG
$(s, t) \in N \times N$ the given chopping criterion
Output: The context-sensitive chop

Let C be the intraprocedural chop for (s, t)
$W = \{n \rightharpoonup m | n, m \in C, n \rightharpoonup m \text{ is a summary edge}\}$
$M = \emptyset$
while $W \neq \emptyset$ *worklist is not empty* **do**
  $W = W/\{n \rightharpoonup m\}$ *remove one element from the worklist*
  Let n' be the to n corresponding formal-in node
  Let m' be the to m corresponding formal-out node
  if $(n', m') \notin M$ n', m' *has not been marked* **then**
    mark n', m' *as visited*
    $M = M \cup (n', m')$
    Let C' be the intraprocedural chop for (n', m')
    $C = C \cup C'$
    $W = W \cup \{n \rightharpoonup m | n, m \in C', n \rightharpoonup m \text{ is a summary edge}\}$
return C

slice is cached such that each slice is only computed once, which causes an asymptotic speedup together with some more optimizations [RR95].

10.2.3 Mixed Context-Sensitivity Chopping

Chopping with summary edges has a high asymptotic complexity and is expensive in practice. A simple improvement is to combine context-sensitive slicing with summary edges with context-insensitive chopping. The same two-phase approach as in context-insensitive chopping is used, but with context-sensitive slicing using summary edges instead of context-insensitive slicing. Besides, because same-level chopping is done, only descending into called procedures is needed.

Algorithm 10.3 on the following page, called *Mixed context-sensitivity chopping (MCC)*, is surprisingly precise as shown later. It *approximates* a context-sensitive chop (conservatively).

10.2.4 Limited/Folded Context Chopping

Now, the kLCS slicing algorithm from section 7.3.2 can be adapted for chopping (algorithm 10.4 on page 135). Again, a two-phase approach is used: First, the backward slice is done and all nodes are marked with the encountered call strings. Second, a forward slice is computed, only considering nodes which have been marked with a matching call string in the first phase. They also

Algorithm 10.3 Mixed Context-Sensitivity Chopping

Input: $G = (N, E)$ the given SDG
$(s, t) \in N \times N$ the given chopping criterion
Output: An approximate chop

mark t *as visited in the backward phase*
$W_B = \{t\}$
$B = \{t\}$
while $W_B \neq \emptyset$ *worklist is not empty* **do**
  $W_B = W_B / \{n\}$ *remove one element from the worklist*
  foreach $m \rightharpoonup n \in E$ not a parameter-in or call edges **do**
    if $m \notin B$ m *is not yet marked* **then**
      mark m *as visited in the backward phase*
      $W_B = W_B \cup \{m\}$
      $B = B \cup \{m\}$
if $s \in B$ s *has been marked in the backward phase*
mark s *as visited in the forward phase*
$W_F = \{s\}$
$F = \{s\}$
while $W_F \neq \emptyset$ *worklist is not empty* **do**
  $W_F = W_F / \{n\}$ *remove one element from the worklist*
  foreach $n \rightharpoonup m \in E$ not a parameter-out edge **do**
    if $m \in B$ m *has been marked in the backward phase*
    and $m \notin F$ m *has not yet been marked in the forward phase* **then**
      mark m *as visited in the forward phase*
      $W_F = W_F \cup \{m\}$
      $F = F \cup \{m\}$
return F *the set of all in the forward phase visited nodes*

Algorithm 10.4 Explicitly Context-Sensitive Chopping

Input: $G = (N, E)$ the given IPDG
$(s, t) \in N \times N$ the given chopping criterion
Output: A chop

$W_B = \{(t, \epsilon)\}$, *mark* t *with* (B, ϵ)
while $W_B \neq \emptyset$ W_B *is not empty* DO
  remove one element (m, c) *from* W_B
  foreach $n \rightharpoonup m \in E$ do
    if n has not been marked with (B, c')
      for which $\mathsf{equals}(c, c')$ holds **then**
      if $n \rightharpoonup m$ is a parameter-in or call edge **then**
        Let s_n be the call site of n
        if $\mathsf{match}(s_n, c)$ **then**
          $c' = \mathsf{up}(c)$
          $W_B = W_B \cup \{(n, c')\}$, *mark* n *with* (B, c')
      elsif $n \rightharpoonup m$ is a parameter-out edge **then**
        Let s_m be the call site of m
        $c' = \mathsf{down}(c, s_m)$
        $W_B = W_B \cup \{(n, c')\}$, *mark* n *with* (B, c')
      else
        $W_B = W_B \cup \{(n, c)\}$, *mark* n *with* (B, c)
if s has been marked with (B, ϵ) **then**
  $W_F = \{(s, \epsilon)\}$, *mark* s *with* (F, ϵ)
  while W_F is not empty **do**
    remove one element (m, c) from W_F
    foreach $m \rightharpoonup n \in E$ **do**
      if n has been marked with (B, c')
        for which $\mathsf{equals}(c', c)$ holds **then**
        if n has not been marked with (F, c')
          for which $\mathsf{equals}(c, c')$ holds **then**
          if $m \rightharpoonup n$ is a parameter-out edge **then**
            Let s_n be the call site of n
            if $\mathsf{match}(s_n, c)$ **then**
              $c' = \mathsf{up}(c)$
              $W_F = W_F \cup \{(n, c')\}$, *mark* n *with* (F, c')
          elsif $m \rightharpoonup n$ is a parameter-in or call edge **then**
            Let s_m be the call site of m
            $c' = \mathsf{down}(c, s_m)$
            $W_F = W_F \cup \{(n, c')\}$, *mark* n *with* (F, c')
          else
            $W_F = W_F \cup \{(n, c)\}$, *mark* n *with* (F, c)
return the set of all in the forward phase visited nodes

Algorithm 10.5 Merging Summary Edges

Input: $G = (N, E)$ the given SDG
$(s, t) \in N \times N$ the given chopping criterion
Output: The context-sensitive chop

Let C be the intraprocedural chop for (s, t)
$W = \emptyset$
foreach call site c in C **do**
$W \cup \{\{n \rightharpoonup m | n, m \in C, n \rightharpoonup m$ is a summary edge at $c\}\}$
$M = \emptyset$
while $W \neq \emptyset$ *worklist is not empty* **do**
$S = \emptyset$
$T = \emptyset$
$W = W/\{L\}$ *remove one element (set) from the worklist*
foreach $m \rightharpoonup n \in L$ **do**
Let n' be the to n corresponding formal-in node
Let m' be the to m corresponding formal-out node
if $(n', m') \notin M$ n', m' *has not been marked* **then**
mark n', m' *as visited*
$M = M \cup (n', m')$
$S = S \cup n'$
$T = T \cup m'$
Let C' be the intraprocedural chop for (S, T)
$C = C \cup C'$
foreach call site c in C' **do**
$W \cup \{\{n \rightharpoonup m | n, m \in C, n \rightharpoonup m$ is a summary edge at $c\}\}$
return C

have still to be unmarked with any matching context from the second phase. The definitions of down, up, match and equals are the same as in kLCS. This algorithm is called k-*limited context chopping (*k*LCC)*.

In the same style the kFCS slicing algorithm can be adapted to chopping. As this is straightforward (and due to space limitations) the algorithm is not presented here (but it has been implemented and will be evaluated later in this chapter). This algorithm will be called k-*limited folded context chopping (*k*FCC)*.

10.2.5 An Improved Precise Algorithm

Now, an improved precise chopping algorithm is presented which has a much higher speed. The earlier CSC algorithm of Section 10.2.2 calculates a new chop for every pair of formal-in and -out nodes that have a summary edge between the corresponding actual-in and -out nodes included in the chop. The following observation has been made: If two summary edges of one call site

are included in the chop, it is not needed to compute chops for corresponding pairs of formal-in/-out nodes separately. Instead, a single chop between the set of corresponding formal-in nodes and the set of corresponding formal-out nodes can be done without changing precision.

Proof 10.1
Let S be the set of all summary edges of one single call site which are all included in the starting chop. Let $(i_1, o_1) \in S$ and $(i_2, o_2) \in S$ be a randomly chosen pair of summary edges. Let (i_1', o_1') and (i_2', o_2') be the corresponding pairs of formal-in/-out nodes. Let C_1 be the chop $C(i_1', o_1')$ and $C_2 = C(i_2', o_2')$. Let C_0 be the chop $C(\{i_1', i_2'\}, \{o_1', o_2'\})$, which is a superset of $C_1 \cup C_2$ because $C(\{x_1, x_2\}, \{y_1, y_2\}) = C(x_1, y_1) \cup C(x_1, y_2) \cup C(x_2, y_1) \cup C(x_2, y_2)$.

A problem can only be caused by a node that is included in C_0 but not in $C_1 \cup C_2$. For any $c \in C_0$, $c \notin C_1$, $c \notin C_2$ one of the same-level paths $i_1' \rightharpoonup^* c \rightharpoonup^* o_2'$ or $i_2' \rightharpoonup^* c \rightharpoonup^* o_1'$ must exist. Therefore, a summary edge (i_1, o_2) or (i_2, o_1) must exist. If such a summary edge exists, it must be included in the starting chop (and thus also in S) because i_1, i_2, o_1, o_2 are all in the starting chop. Because of these summary edges, the chop $C_3 = C(i_1', o_2')$ or $C_4 = (i_2', o_1')$ will be added into the resulting chop and therefore c will be added to the resulting chop anyways. □

Algorithm 10.5 on the facing page is based on merging of summary edges dependent on their call site and is therefore called *summary-merged chopping (SMC)*. After computing the starting chop, all new summary edges of visited call sites are collected. Then a new chop is done for every call site between the set of the corresponding formal-in nodes and the set of the corresponding formal-out nodes. The resulting nodes are added to the starting chop and the procedure is repeated with the new resulting summary edges until there are no more new summary edges left. This causes low runtimes because a much smaller number of chops is computed.

10.2.6 Evaluation

To the authors knowledge, an evaluation of chopping algorithms has never been done, [RR95] reports only limited experience. All presented chopping algorithms (CIC, CSC, SMC, MCC, kLCC and kFCC) have been implemented to evaluate them completely. As chopping criteria every tenth of all same-level pairs of formal-in/-out nodes has been chosen. The time needed to do the complete set of chops is measured, as the average time to calculate one chop is sub-second.

Precision

To measure the precision of the different algorithms, the same approach as in section 7.4 is followed: CIC is considered to have 0% precision and CSC (or SMC) to have 100% precision. The results are shown in Figure 10.3: The first column contains the amount of chopping criteria (= amount of chops done) and

	chops	CIC	CSC	SMC	MCC	1LCC	2LCC	3LCC	4LCC	5LCC	6LCC
A	196	1050	262	262	96	6	6	6	6	6	6
B	2802	1373	259	259	97	9	48	50	52	52	68
C	16098	5177	–	794	99	26	26	26	57	57	57
D	2709	228	83	83	96	28	28	28	28	28	28
E	5303	1000	233	233	98	83	83	83	83	83	83
F	848	3562	700	700	97	63	64	75	75	75	75
G	33994	5573	–	818	99	37	42	48	56	56	56
H	2676	5399	1984	1984	99	5	*7*	7	7	–	–
I	1528	1450	767	767	94	43	43	43	–	–	–
J	4892	551	169	169	95	34	36	38	73	73	73
K	21839	3298	41	41	99	88	99	99	99	99	99
L	15672	6370	–	1855	98	42	46	46	46	46	46
M	42442	5265	–	522	99	*29*	30	30	30	30	–
						1FCC	2FCC	3FCC	4FCC	5FCC	6FCC
A						6	6	6	6	**6**	**6**
B						9	48	50	52	52	68
C						26	26	26	57	57	57
D						28	28	28	28	28	28
E						83	83	83	83	83	83
F						63	64	75	75	75	75
G						37	42	48	56	56	**56**
H						5	*7*	**8**	**12**	–	–
I						43	43	43	44	44	44
J						34	36	38	73	73	73
K						88	99	**99**	**99**	**99**	**99**
L						42	46	46	46	46	46
M						29	**30**	**30**	–	**91**	**93**

Figure 10.3: Precision of MCC, kLCC and kFCC (in %)

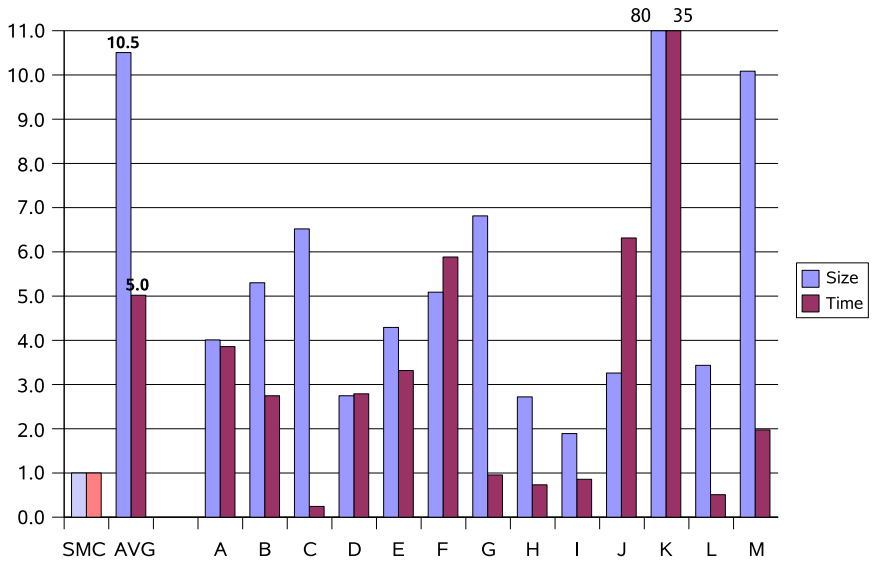


Figure 10.4: Context-insensitive vs. context-sensitive chopping

the next three columns give the average size of CIC, CSC and SMC chops. The first thing to see is that there are four cases where it was impossible to finish CSC in the given time boundaries. Therefore, a direct comparison between the context-insensitive and -sensitive algorithm is only shown for CIC and SMC in figure 10.4: context-insensitive chopping is *very* imprecise (in the most extreme case the context-insensitive algorithm produces chops 80 times larger than the chops computed by the context-sensitive one).

The simple MCC algorithm is surprisingly precise: Usually it has around 96% precision (figure 10.5) and is always more precise than 6LCC or 6FCC (figure 10.3). This shows that the main source of imprecision in chopping is the loss of the same-level property which is kept by the approximate algorithm MCC. The other experiences are in a kind of contrast to the experiences with the slicing algorithms: All other algorithms than CSC, SMC and MCC are much less precise and with increased k the precision of kLCC and kFCC is only increasing slowly (shown for kFCC in figure 10.6 on the next page).

Speed

The most important observation while comparing the runtimes of CSC and SMC is that SMC is always *much* faster. To denote an extremum, SMC only needs 1% of CSC's runtime in test case I. The MCC algorithm is not only surprisingly precise but also fast. For the larger test cases it is much faster than any of the other algorithms.

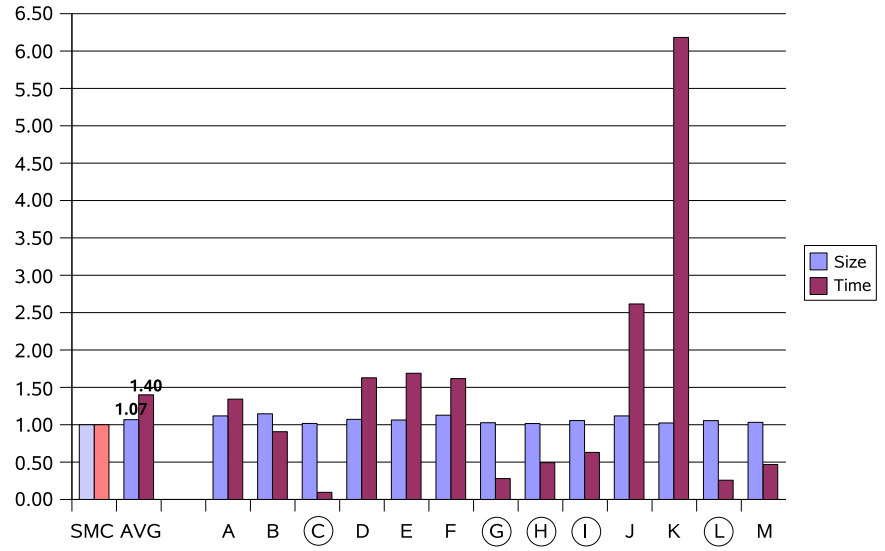


Figure 10.5: Approximate vs. context-sensitive chopping

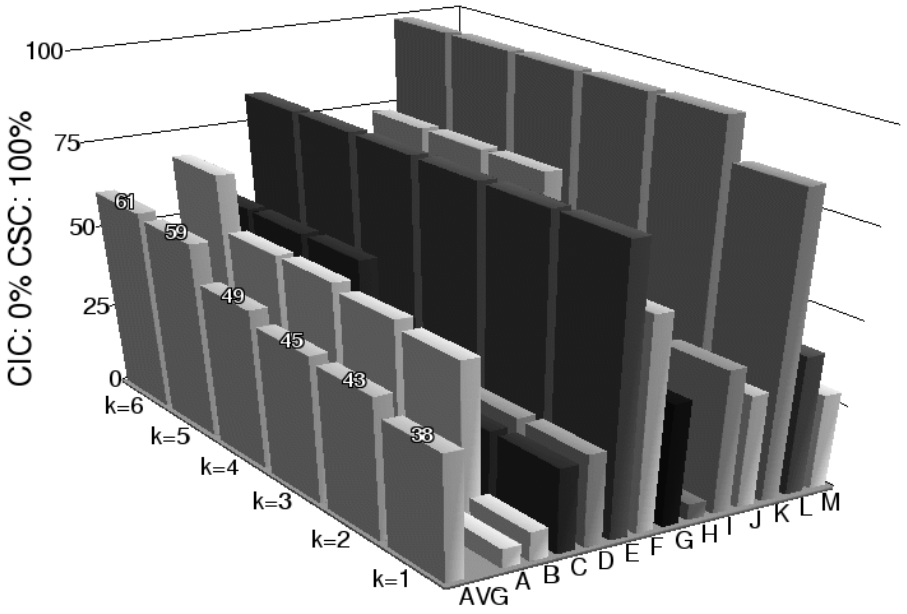


Figure 10.6: Precision of kFCC (avg. size)

	CIC	CSC	SMC	MCC	1LCC	2LCC	3LCC	4LCC	5LCC	6LCC
A	1,35	2,54	0,35	0,47	4,07	7,88	20,5	78,3	384	1964
B	32,4	242	11,8	10,7	88,2	108	185	371	1069	2663
C	1145	–	4691	451	2293	3389	4670	3113	2613	2101
D	9,60	31,1	3,44	5,60	20,0	29,6	45,5	68,9	65,0	49,5
E	59,7	511	18,0	30,4	85,0	125	223	787	2061	3581
F	20,3	14,6	3,45	5,58	40,9	154	159	226	451	530
G	3985	–	4172	1171	6647	8721	11884	16844	21892	28437
H	112	9098	153	75,4	420	993	2884	9253	–	–
I	18,1	2003	21,1	13,3	51,3	177	1778	–	–	–
J	60,8	44,1	9,63	25,2	140	185	268	323	1218	7921
K	1038	247	29,6	183	1127	1460	3137	3846	3167	2472
L	1202	–	2358	609	2302	3134	3967	4285	4119	4082
M	4037	–	2045	957	8452	13062	19314	25121	29800	–
					1FCC	2FCC	3FCC	4FCC	5FCC	6FCC
A					3,67	5,63	11,3	14,4	15,4	15,8
B					87,8	106	171	296	511	379
C					2295	3381	4697	3106	2612	2107
D					20,0	29,7	45,1	68,6	64,9	49,5
E					85,2	125	234	746	2087	3640
F					41,1	155	160	227	454	532
G					6486	8260	11081	15173	20264	28753
H					309	1131	3623	17486	–	–
I					46,2	91,7	122	176	232	214
J					138	179	237	151	159	154
K					1134	1476	3191	4104	3459	2858
L					2301	3134	3974	4288	4114	4099
M					8465	14981	28793	–	22959	24958

Figure 10.7: Runtimes of kLCC and kFCC (in sec.)

The comparison of kLCC with kFCC in terms of runtime reveals the same results as for kLCS and kFCS: In many cases kFCC is much faster than kLCC and less likely to suffer from combinatorial explosion. Also, increased k may sometimes result in lower runtime.

In section 7.4 it was seen that usually the context-sensitive slicing algorithm SIS is not slower than the context-insensitive algorithm CIS. This is not the case with chopping: In only three test cases CIC is slower than CSC, and in five test cases CIC is even faster than SMC (figure 10.4 on page 139). This is due to the much higher complexity of chopping with summary edges in comparison to slicing with summary edges. However, in cases where CIC is faster than SMC (circled in figure 10.5 on page 140), the approximate algorithm MCC can be used instead: It is much faster in those cases (it is always faster than CIC).

If the speed of kLCC and kFCC is compared against CSC and SMC, different results as in slicing are experienced: nine test cases are faster in chopping with 1LCC or 1FCC than CSC and one is even faster than SMC.

The results for chopping are not as clear as in slicing, but they lead to the same recommendation: due to the high precision SMC is preferable to all other chopping algorithms. In cases where SMC is slower than CIC, the approximate algorithm MCC can be used with acceptable loss of precision. The call string approaches most often have a poor precision.

10.2.7 Non-Same-Level Chopping

Thus far, only same-level chopping has been presented. This section will show how to compute chops for start and target nodes in different procedures. Again, a truncated and a non-truncated variant will be shown. A truncated chop is not allowed to contain nodes of called procedures: All nodes must lie on *truncated non-same-level paths* from source to target nodes. A truncated non-same-level path P can be split in two parts p_1 and p_2 with $p = p_1 p_2$, such that p_1 does not contain parameter-in and call edges and p_2 does not contain parameter-out edges. Why is this restriction needed? If p would contain a parameter-out edge e_1 to the right of a parameter-in or call edge e_2, it had also to contain a call or parameter-in edge e_3 (probably $e_3 = e_2$), where e_3 and e_2 have matching call sites (because p must be realizable). Such a path would violate the truncated restriction, because e_3 enters a called procedure and e_1 returns from it.

Instead of encoding this restriction into the algorithm, Reps and Rosay [RR95] found a more simple way to compute truncated non-same-level chops: A truncated non-same-level chop is computed by set operations on restricted forward and backward slices:

- $S^B_{\uparrow}(n)$ is a backward slice that does not traverse parameter-out edges.
- $S^B_{\downarrow}(n)$ is a backward slice not traversing parameter-in or call edges.
- $S^F_{\uparrow}(n)$ is a forward slice that does not traverse parameter-out edges.
- $S^F_{\downarrow}(n)$ is a forward slice that does not traverse parameter-in or call edges.

The truncated non-same-level chop C^{TN} is now computed according to:

$$\begin{aligned} W(S,T) &= S^F_{\downarrow}(S) \cap S^B_{\uparrow}(T) \\ C^{TN}(S,T) &= \left(S^F_{\downarrow}(S) \cap S^B_{\downarrow}(W(S,T))\right) \cup \left(S^F_{\uparrow}(W(S,T) \cap S^B_{\uparrow}(T)\right) \end{aligned}$$

The non-truncated non-same-level chop C^{NN} is computed by extending C^{TN} with same-level chops for all included summary edges similar to the extension presented in the previous sections.

10.3 Barrier Slicing and Chopping

The presented slicing and chopping techniques compute very fixed results where the user has no influence. However, during slicing and chopping a user might want to give additional restrictions or additional knowledge to the computation:

1. A user might know that a certain data dependence cannot happen. Because the underlying data flow analysis is a conservative approximation and the pointer analysis is imprecise, it might be clear to the user that a dependence found by the analysis cannot happen in reality. For example, the analysis assumes a dependence between a definition `a[i]=...` and a usage `...=a[j]` of an array, but the user discovers that `i` and `j` never have the same value. If such a dependence is removed from the dependence graph, the computed slice might be smaller.

2. A user might want to exclude specific parts of the program that are of no interest for his purposes. For example, he might know that certain statement blocks are not executed during runs of interest; or he might want to ignore error handling or recovery code, when he is only interested in normal execution.

3. During debugging, a slice might contain parts of the analyzed program that are known (or assumed) to be bug-free. These parts should be removed from the slice to make the slice more focused.

Both points have been tackled independently: For example, the removal of dependences from the dependence graph by the user has been applied in Steindl's slicer [Ste98, Ste99a]. The removal of parts from a slice has been presented by Lyle and Weiser [LW87] and is called *dicing*.

The following approach integrates both into a new kind of slicing, called *barrier slicing*, where nodes (or edges) in the dependence graph are declared to be a *barrier* that transitive dependence is not allowed to pass.

Definition 10.3 (Barrier Slice)
The *barrier slice* $S_{\#}(C, B)$ of an IPDG $G = (N, E)$ for the slicing criterion $C \subseteq N$ with the barrier set of nodes $B \subseteq N$ consists of all nodes on which a node

$n \in C$ (transitively) depends via an interprocedurally realizable path that does not pass a node of B:

$$S_{\#}(C,B) = \{m \in N \quad | \quad p = m \rightarrow_R^\star n \wedge n \in C \\ \wedge p = \langle n_1, \ldots, n_l \rangle \\ \wedge \forall 1 < i \leqslant l : n_i \notin B\}$$

The barrier may also be defined by a set of edges; the previous definition is adapted accordingly.

From barrier slicing it is only a small step to barrier chopping:

Definition 10.4 (Barrier Chop)
The *barrier chop* $C_{\#}(S,T,B)$ of an IPDG $G = (N,E)$ from the source criterion $S \subseteq N$ to the target criterion $T \subseteq N$ with the barrier set of nodes B consists of all nodes on which a node of T (transitively) depends via an interprocedurally realizable path from a node of S to the node in T that does not pass a node of $B \subseteq N$:

$$C_{\#}(S,T,B) = \{n \quad | \quad p = s \rightarrow_R^\star t \wedge s \in S \wedge t \in T \\ \wedge p = \langle n_1, \ldots, n_l \rangle \wedge \exists i : n = n_i \\ \wedge \forall 1 < j < l : n_j \notin B\}$$

The barrier may also be defined by a set of edges; the previous definition is adapted accordingly.

Again, the forward/backward, truncated/non-truncated, same-level/non-same-level variants can be defined, but are not presented here.

The computation of barrier slices and chops cause a minor problem: The additional constraint of the barrier destroys the usability of the summary edges as they do not obey the barrier. Even when the summary edges would comply with the barrier, the advantage of summary edges is lost: They can no longer be computed once and used for different slices and chops, because they have to be computed for each barrier slice and chop individually. However, the algorithm 7.2 on page 89 can be adapted to compute summary edges which obey the barrier: The new version (algorithm 10.6 on the next page) is based on blocking and unblocking summary edges. First, all summary edges stemming from calls that might call a procedure with a node from the barrier at some time are blocked. This set is a very conservative approximation and the second step unblocks summary edges where a barrier-free path exists between actual-in and -out nodes. The first phase replaces the initialization phase of the original algorithm and the second phase does not generate new summary edges, but unblocks them. Only the version where the barrier consists of nodes is shown.

This algorithm is cheaper than the complete recomputation of summary edges, because it only propagates node pairs to find barrier-free paths between actual-in/-out nodes if a summary edge and therefore a (not necessarily barrier-free path) exists.

Algorithm 10.6 Computation of Blocked Summary Edges

Input: $G = (N, E)$ the given SDG
$B \subset N$ the given barrier
Output: A set S of blocked summary edges

Initialization
$S = \emptyset, W = \emptyset$
Block all reachable summary edges
foreach $n \in B$ **do**
  Let P be the procedure containing n
  Let S_P be the set of summary edges for calls to P
  $S = S \cup S_P$
  $W = W \cup \{(n, n) \mid n \text{ is a formal-out node of } P\}$
repeat
  $S_0 = S$
  foreach $x \rightharpoonup y \in S$ **do**
    Let P be the procedure containing x
    Let S_P be the set of summary edges for calls to P
    $S = S \cup S_P$
    $W = W \cup \{(n, n) \mid n \text{ is a formal-out node of } P\}$
until $S_0 = S$

Unblock some summary edges
P = W
while $W \neq \emptyset$ *worklist is not empty* **do**
  $W = W/\{(n, m)\}$ *remove one element from the worklist*
  if n is a formal-in node **then**
    foreach $n' \xrightarrow{pi} n$ which is a parameter-in edge **do**
      foreach $m \xrightarrow{po} m'$ which is a parameter-out-edge **do**
        if $n' \xrightarrow{su} m' \in S$ **then**
          $S = S - \{n' \xrightarrow{su} m'\}$ *unblock summary edge*
          foreach $(m', x) \in P \wedge (n', x) \notin P$ **do**
            $P = P \cup \{(n', x)\}$
            $W = W \cup \{(n', x)\}$
  else
    foreach $n' \xrightarrow{dd,cd} n$ **do**
      if $n' \notin B \wedge (n', m) \notin P$ **then**
        $P = P \cup \{(n', m)\}$
        $W = W \cup \{(n', m)\}$
    foreach $n' \xrightarrow{su} n$ **do**
      if $n' \notin B \wedge n' \xrightarrow{su} n \notin S \wedge (n', m) \notin P$ **then**
        $P = P \cup \{(n', m)\}$
        $W = W \cup \{(n', m)\}$
return S *the set of blocked summary edges*

Example 10.2: Consider the example in figure 10.8 on the facing page: If a slice for `u_kg` in line 33 is computed, almost the complete program is in the slice: Just lines 11 and 12 are omitted. One might be interested why the variable `p_cd` is in the slice and has an influence on `u_kg`. Therefore a chop is computed: The source criterion are all statements containing variable `p_cd` and the target criterion is `u_kg` in line 33. The computed chop is shown in figure 10.9 on page 148. In that chop, line 19 looks suspicious, where the variable `u_kg` is defined, using variable `kal_kg`. Another chop from all statements containing variable `kal_kg` to the same target consists only of lines 14, 19, 26, 28 and 33 (figure 10.10 on page 149). A closer look reveals that statements 26 and 28 "transmit" the influence from `p_cd` on `u_kg`. To check that no other statement is responsible, a barrier chop is computed: The source are the statements with `p_cd` again, the target criterion is still `u_kg` in line 33, and the barrier is line 26 and 28. The computed chop is empty and reveals that lines 26 and 28 are the "hot spots".

10.3.1 Core Chop

A specialized version of a barrier chop is a *core chop* where the barrier consists of the source and target criterion nodes.

Definition 10.5 (Core Chop)
A *core chop* $C_\circ(S, T)$ is defined as:

$$C_\circ(S, T) = C_\#(S, T, S \cup T)$$

It is well suited for chops with large source and target criterion sets: Only the statements connecting the source to the target are part of the chop. Here is important that a barrier chop allows barrier nodes to be included in the criteria. In that case, the criterion nodes are only start or end nodes of the path and are not allowed elsewhere.

10.3.2 Self Chop

When slices or chops are computed for large criterion sets, it is sometimes important to know which parts of the criterion set influence themselves and which statements are part of such an influence. After identifying such statements, they can specially be handled during following analyses. They can be computed simply by a *self chop*, where a set is both source and target criterion:

Definition 10.6 (Self Chop)
A *self chop* $C_\bowtie(S)$ is defined as:

$$C_\bowtie(S) = C(S, S)$$

It computes the strongly connected components of the SDG which contain nodes of the criterion. These components can be of special interest to the user, or they are used to make core chops even stronger:

```
 1  #define TRUE 1
 2  #define CTRL2 0
 3  #define PB 0
 4  #define PA 1
 5
 6  void main()
 7  {
 8    int p_ab[2] = {0, 1};
 9    int p_cd[1] = {0};
10    char e_puf[8];
11    int u;
12    int idx;
13    float u_kg;
14    float kal_kg = 1.0;
15
16    while(TRUE) {
17      if ((p_ab[CTRL2] & 0x10)==0) {
18        u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
19        u_kg = (float) u * kal_kg;
20      }
21      if ((p_cd[CTRL2] & 0x01) != 0) {
22        for (idx=0;idx<7;idx++) {
23          e_puf[idx] = (char)p_cd[PA];
24          if ((p_cd[CTRL2] & 0x10) != 0) {
25            if (e_puf[idx] == '+')
26              kal_kg *= 1.01;
27            else if (e_puf[idx] == '-')
28              kal_kg *= 0.99;
29          }
30        }
31        e_puf[idx] = '\0';
32      }
33      printf("Artikel: %7.7s\n   %6.2f kg    ",e_puf,u_kg);
34    }
35  }
```

Figure 10.8: An example

```
1  #define TRUE 1
2  #define CTRL2 0
3  #define PB 0
4  #define PA 1
5
6  void main()
7  {
8    int p_ab[2] = {0, 1};
9    int p_cd[1] = {0};
10   char e_puf[8];
11   int u;
12   int idx;
13   float u_kg;
14   float kal_kg = 1.0;
15
16   while(TRUE) {
17     if ((p_ab[CTRL2] & 0x10)==0) {
18       u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
19       u_kg = (float) u * kal_kg;
20     }
21     if ((p_cd[CTRL2] & 0x01) != 0) {
22       for (idx=0;idx<7;idx++) {
23         e_puf[idx] = (char)p_cd[PA];
24         if ((p_cd[CTRL2] & 0x10) != 0) {
25           if (e_puf[idx] == '+')
26             kal_kg *= 1.01;
27           else if (e_puf[idx] == '-')
28             kal_kg *= 0.99;
29         }
30       }
31       e_puf[idx] = '\0';
32     }
33     printf("Artikel: %7.7s\n   %6.2f kg    ",e_puf,u_kg);
34   }
35 }
```

Figure 10.9: A chop for the example in figure 10.8

```
1  #define TRUE 1
2  #define CTRL2 0
3  #define PB 0
4  #define PA 1
5
6  void main()
7  {
8    int p_ab[2] = {0, 1};
9    int p_cd[1] = {0};
10   char e_puf[8];
11   int u;
12   int idx;
13   float u_kg;
14   float kal_kg = 1.0;
15
16   while(TRUE) {
17     if ((p_ab[CTRL2] & 0x10)==0) {
18       u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
19       u_kg = (float) u * kal_kg;
20     }
21     if ((p_cd[CTRL2] & 0x01) != 0) {
22       for (idx=0;idx<7;idx++) {
23         e_puf[idx] = (char)p_cd[PA];
24         if ((p_cd[CTRL2] & 0x10) != 0) {
25           if (e_puf[idx] == '+')
26             kal_kg *= 1.01;
27           else if (e_puf[idx] == '-')
28             kal_kg *= 0.99;
29         }
30       }
31       e_puf[idx] = '\0';
32     }
33     printf("Artikel: %7.7s\n   %6.2f kg    ",e_puf,u_kg);
34   }
35 }
```

Figure 10.10: Another chop for the example in figure 10.8

Definition 10.7 (Strong Core Chop)
A *strong core chop* $C_\bullet(S,T)$ is defined as:

$$C_\bullet(S,T) = C_\#(S \cup C_\bowtie(S), T \cup C_\bowtie(T), S \cup T \cup C_\bowtie(S) \cup C_\bowtie(T))$$

A strong core chop only contains statements that connect the source criterion to the target criterion, none of the resulting statements will have an influence on the source criterion, and the target criterion will have no impact on the resulting statements.

Thus, the strong core chop only contains the most important nodes of the influence between the source and target criterion.

10.4 Abstract Visualization

For program understanding-in-the-large the presented visualization techniques are not very helpful. If an unknown program is analyzed, the very detailed information of program dependence and slices is overwhelming and a much less detailed information is needed. The understanding starts with variables and procedures and not statements. This section will show how slicing and chopping can help to visualize programs in a more abstract way, illustrating relations between variables or procedures.

10.4.1 Variables or Procedures as Criterion

It is possible to define slices for variables or procedures informally:

1. A (backward) slice for a criterion variable v is the set of statements (or nodes in the PDG) which may influence variable v at some point of the program.

2. A (backward) slice for a criterion procedure P is the set of statements (or nodes in the PDG) which may influence a statement of P.

The example in the last section already used this to compute some chops. The only effort needed is to compute the node sets that match the above criterion definitions:

Variables as Criterion

Through the fine-grained representation it is easy to identify all nodes that access a criterion variable v: Such nodes are nodes with the "reference" operator (μ_{op}, see chapter 4) and the variable v as value (μ_{val}). The slice to be computed is defined as:

$$S(v) = S(\{n \mid \mu_{op}(n) = \mathsf{reference} \wedge \mu_{val}(n) = v\})$$

Procedures as Criterion

Again, it is very easy to identify all nodes of a procedure P, because nodes of the fine-grained PDG contain an attribute μ_{prc} that links a node to the embedding procedure:

$$S(P) = S(\{n \mid \mu_{prc}(n) = P\})$$

The adaption to the various slicing and chopping variants is obvious.

10.4.2 Visualization of the Influence Range

To understand a previously unknown program, it is helpful to identify the 'hot' procedures and global variables—the procedures and variables with the highest impact on the system. A simple measurement is to compute slices for every procedure or global variable and record the size of the computed slices. However, this might be too simple and a slightly better approach is to compute chops between the procedures or variables. A visualization tool has been implemented that computes a $n \times n$ matrix for n procedures or variables, where every element $n_{i,j}$ of the matrix is the size of a chop from the procedure or variable n_j to n_i. The matrix is painted using a color for every entry, corresponding to the size—the bigger, the darker. Figure 10.11 on the next page shows such a visualization for the `ansitape` program. The columns show variables 0–34 as source criteria and the rows as target criteria. This matrix can be interpreted like the following examples:

- The global variables `stdin` (column two), `stdout` (3) and `stderr` (4) have empty chops with all other variables (light columns 2–4). This is obvious for `stdout` and `stderr` while `stdin` has no influence because the program does only read from tapes.

- The variable `stdout` (row 3) is not influenced (empty chops with stdout as target criterion), but `stderr` (row 4) is. The ansitape program normally writes all messages to `stderr` and produces no other output.

- Row 12 has the biggest chops (and is the darkest row). This is variable `tcb`, the tape control block, which is the main global variable of the program.

An implementation is shown in figure 10.12 on page 153: The three windows contain the chop matrix visualization (in this case for procedure-procedure-chops), a color scale and a window that shows the names of the procedures and their chop's size for the last chosen matrix element. With this tool, it is easy to get a first impression of the software to analyze. Important procedures or global variables can be identified on first sight and their relationship be studied. Doing this as a preparing stage aids in later, more thorough investigations with traditional slicing visualizations like the ones presented in the previous chapter 9.

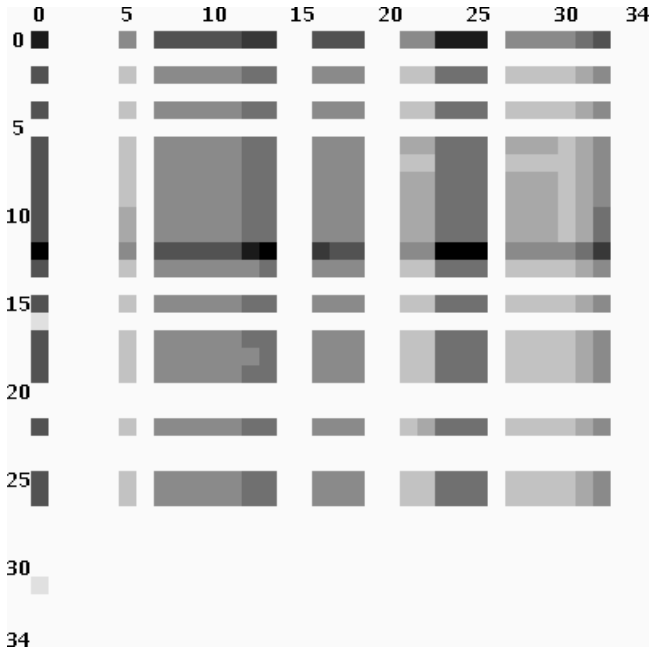


Figure 10.11: Visualization of chops for all global variables

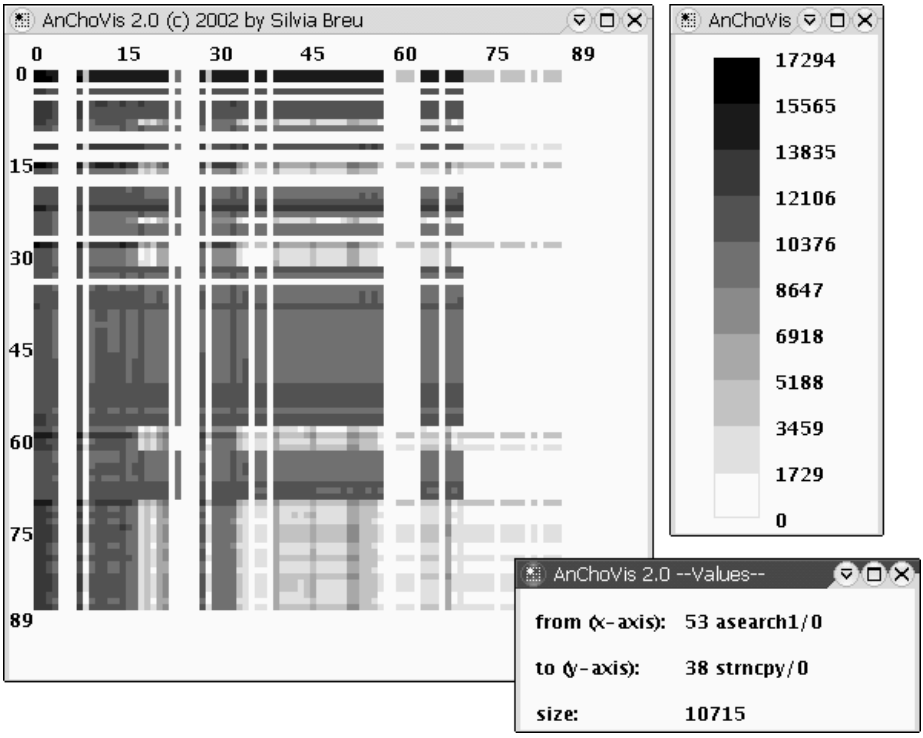


Figure 10.12: GUI for the chop visualization

10.5 Related Work

Chopping as presented here has been introduced by Jackson and Rollins [JR94b] and extended by Reps and Rosay [RR95]. The only other tool able to compute precise chops is CodeSurfer [AT01]. The evaluation of chopping algorithms in section 10.2 is the first study (previously published in [Kri02]) while Reps and Rosay [RR95] only report limited experience.

The SeeSlice slicing tool [BE94] already included some of the presented focusing and visualization techniques, e.g. the distance-limited slicing, visualizing distances, etc. The slicer of the CANTO environment [AFL+97] can be used in a stepping mode which is similar to distance-limited slicing: At each step the slice grows by considering one step of data or control dependence.

A *decomposition slice* [GL91, Gal96, GO97] is basically a slice for a variable at all statements writing that variable. The decomposition slice is used to form a graph using the partial ordering induced by proper subset inclusion of the decomposition slices for all variables.

Beck and Eichmann [BE93] use slicing to isolate statements of a module that influence an exported behavior. Their work uses *interface dependence graphs* and *interface slicing*.

Steindl [Ste98, Ste99a] has developed a slicer for Oberon where the user can choose certain dependences to be removed from the dependence graph.

Set operations on slices produce various variants: Chopping uses intersection of a backward and a forward slice. The intersection of two forward or two backward slices is called a *backbone slice*. Dicing [LW87] is the subtraction of two slices. However, set operations on slices need special attention because the union of two slices may not produce a valid slice [DLHHK03].

Orso et al [OSH01b] presents a slicing algorithm which augments edges with types and restricts reachability onto a set of types, creating slices restricted to these types. Their algorithm needs to compute the summary edges specific to each slice (similar to algorithm 10.6 on page 145). However, it only works for programs without recursion.

10.5.1 Dynamic Slicing

During debugging not all possible executions of a program are of interest. Usually, only one test case where a problem arises is in focus. Therefore, Korel and Laski have introduced *dynamic slicing* [KL88]: Dynamic Slicing focuses on a single execution of a program instead of all possible executions like in Weiser's (static) slicing. The slicing criterion for a dynamic slice additionally specifies the (complete) input for the program. The criterion is now a triple $c = (I, s, V)$ consisting of the input I, a statement s and a subset V of the variables of the analyzed program.

Definition 10.8 (Dynamic Slice)
A *dynamic slice* $S(c)$ of a program P on a slicing criterion c is any executable program P', where

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

Korel and Laski also presented an algorithm to compute dynamic slices based on the computation of sets of dynamic def-use relations. However, the algorithm has been shown to be imprecise in [AH90]. The algorithm of Gopal [Gop91], based on dynamic versions of information-flow relations [BC85], may compute non-terminating slices under certain loop conditions.

Miller and Choi [MC88] introduced a dynamic version of program dependence graphs which they use for flowback analysis. Agrawal and Horgan [AH90] have developed dynamic slicing on top of dependence graphs. They present four algorithms differing in precision and complexity:

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

This work has been extended to *interprocedural* dynamic slicing in [ADS91]. Kamkar's algorithms [KSF92, Kam93, KFS93a] are similar and focus on interprocedural dynamic slicing. An approach to interprocedural flowback analysis is presented in [CMN91].

If the computed dynamic slices have to be executable, the use of unstructured control flow requires special treatment, like in static slicing. Algorithms for such circumstances have been developed by Korel [Kor95, Kor97] and Huynh [HS97].

To overcome the space requirements of dynamic dependence graphs, Goswami and Mall [GM02] have suggested to compute the graphs based on the paths between loop entries and exits. If the same path is traversed in a later iteration of the loop, the earlier execution of the path is ignored. Obviously, this produces incorrect dynamic slices: Consider a loop that contains two alternating paths

A and B through the loop. Assume an execution where the loop is iterated at least four times and the B path is executed at last; such an execution will be similar to ...ABAB. The suggested algorithm will now only consider AB as an execution and will omit dependences originating from statements in B going to statements in A.

A correct and efficient technique has been presented by Mund et al [MMS02]: Their dynamic slicing algorithm uses the standard PDG where unstable edges are handled specially. Unstable edges are data dependence edges with the same target node and where the source nodes define the same variable. During execution the unstable edges are marked corresponding to which of them has been executed lately and the dynamic slice for each node is updated accordingly. Its space complexity is $O(n^2)$ where n is the number of statements in the program. The authors claim that the time complexity is also $O(n^2)$, which is clearly wrong as it must be dependent on the number of executed statements N. The time complexity must be at least $O(nN)$.

Beszédes et al [BGS+01] have developed a technique that computes dynamic slices for each executed statement during execution. They employ a special representation, which combines control and data dependence. During execution, the presented algorithm keeps track of the points of the last definition for any used variable. An implementation of the algorithm is able to slice C including unstructured control flow and procedure calls.

Dynamic slicing has been formalized based on natural semantics in [GM99]. This lead to a generic, language-independent dynamic slicing analysis, which can be instantiated for imperative, logic or functional languages.

Extensions for object-oriented programs are straightforward [OHFI01]. Extensions to dynamic slicing for concurrent programs have been presented in [KF92, DGS92, Che93, KK95, GM00].

Another work on dynamic slicing is [OHFI01], which includes lightweight tracing information like tracing procedure calls [NJKI99, TOI01]. Such an approach, called *hybrid slicing*, has been presented earlier by Gupta and Soffa [GS95].

10.5.2 Variations of Dynamic Slicing

Hall [Hal95] has presented another form of dynamic slicing: *simultaneous dynamic slicing*, where a dynamic slice for a set of inputs is computed. This is not just the union of the dynamic slices for each of the inputs, as the union may not result in a correct dynamic slice for an input. Beszédes [BFS+02] uses unions of dynamic slices to approximate the minimal static slice.

Between static and dynamic slicing lies *quasi static slicing*, presented by Venkatesh [Ven91], where only some of the input variables have a fixed value (as specified in the slicing criterion) and the others may vary like in static slicing. A generalization is *conditioned slicing* [DLFM96, CCD98, DFHH00]: The input variables are constrained by first order logic formula. A similar approach is *parametric program slicing*, presented by Field et al [FRT95].

A survey on dynamic slicing approaches can be found in [Tip95, KR98].

Chapter 11

Optimizing the PDG

Two things matter for slicing and chopping based on program dependence graphs: size and precision. Previous chapters have shown that larger graphs cause higher runtimes and bad precision can cause an avalanche effect on the size of computed chops. Program dependence graphs have originally been developed for intermediate representation for optimizations in compilers. So why not use optimization techniques to decrease the size and increase the precision of program dependence graphs? This is exactly what this chapter is about. The next section will present graph compression techniques that have no influence on precision. The second section contains more advanced techniques from compiler optimization, which mainly increase the precision of the program dependence graphs. These optimization can also increase the precision of path condition computation, which will be discussed in chapter 13.

11.1 Reducing the Size

A way to reduce the running time of algorithms working in dependence graphs is to make the graphs smaller. However, many approaches to reduce the size are based on abstraction and approximation: The reduced graph causes a reduced precision of algorithms working on it. This is undesirable for program slicing and chopping, where highest precision is important. For that reason the following sections will present three techniques to reduce the size of interprocedural program dependence graphs and/or system dependence graphs, which will not cause a loss of precision. The first approach reduces fine-grained to coarse-grained graphs (similar to traditional dependence graphs), the second approach folds cycles and the last technique removes redundant nodes representing actual and formal parameters. As will be shown, this removal causes a significant size reduction. However, the folding technique has the highest impact: around 40% size reduction.

Program	time	$\|N\|$	$\|E\|$	$\|N'\|$	$\|E'\|$	r_N	r_E
agrep	0.04	22820	71328	20812	67375	8%	5%
ansitape	0.02	8501	35086	7353	32901	13%	6%
assembler	0.07	16049	247920	15046	245955	6%	0%
bison	0.06	33158	141231	30906	136796	6%	3%
cdecl	0.03	13383	62259	7092	49672	47%	20%
compiler	0.03	16833	97897	16167	96687	3%	1%
ctags	0.02	12958	51917	12075	50203	6%	3%
diff	0.29	65682	1288579	62207	1281892	5%	0%
flex	0.19	48098	680973	42957	670711	10%	1%
football	0.03	18879	63729	16948	59950	10%	5%
gnugo	0.01	7786	20357	6993	18856	10%	7%
loader	0.00	3718	8870	3447	8353	7%	5%
patch	0.08	30774	237832	26520	229496	13%	3%
plot2fig	0.00	2938	8154	2647	7606	9%	6%
rolo	0.18	42272	682510	40582	679279	3%	0%
simulator	0.02	14699	39319	13096	36259	10%	7%
					$\emptyset$	10.2%	4.4%

Table 11.1: Fine-grained vs. coarse-grained SDGs

11.1.1 Moving to Coarse Granularity

For most applications the fine-grained representation of dependence graphs is not needed and the standard representation, where nodes represent complete statements, is good enough. Such a coarse-grained PDG can be constructed from a fine-grained by folding the nodes of an expression. Because an expression can contain multiple side effects, the expression nodes are separated into equivalence classes: All nodes of an equivalence class belong to an expression with at most one side effect. After folding the nodes of each equivalence class into one node, the resulting smaller graph is similar to a standard PDG.

Table 11.1 shows the comparison of the fine and coarse grained SDGs: The first column shows the name of the analyzed program, the second the time needed (in seconds) to construct the coarse grained SDG. The next columns show the sizes of the fine and the coarse grained SDGs in amount of nodes ($|N|$ and $|N'|$) and edges ($|E|$ and $|E'|$). The last two columns show the size reduction in percent: $r_N = (|N| - |N'|)/|N|$ and $r_E = (|E| - |E'|)/|E|$. The size reduction is significant though smaller than expected: On average, the coarse grained SDG has only 10% fewer nodes and only 4% fewer edges.

11.1.2 Folding Cycles

For reachability problems like slicing and chopping, cycles in graphs have the effect that if any node of a cycle is in the slice or chop, all other nodes of the cycle must be part of it, too. A simple but very effective technique is to replace

```
1    sum = 0
2    mul = 1
3    a = 1
4    b = read()
5    while (a <= b) {
6      sum = sum + a
7      mul = mul * a
8      a = a + 1
9    }
10   write(sum)
11   write(mul)
```

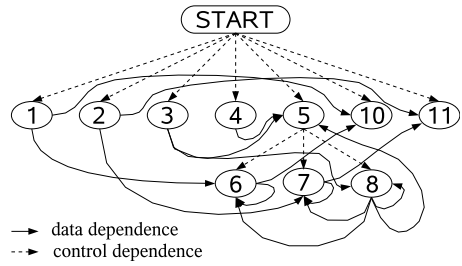


Figure 11.1: Example for a PDG with cycles

the cycles, which form strongly connected components, with one single node. This technique is called *folding*. It does not cause loss of precision, because as soon as one node of the strongly connected component is included in a chop or a slice, all other nodes of the component must be included too.

Example 11.1: Figure 11.1 shows a program and its PDG with a cycle that contains nodes 5 and 8. Any slice or chop that contains one of these nodes must also contain the other. Both nodes can be replaced by a single one.

In this example, the size reduction is very small, only one node is eliminated. However, an evaluation will show that in real programs the folding has high impact.

The folding of strongly connected components is simple for programs without procedures. Folding cycles that span multiple procedures invalidates the interprocedurally realizable path property: Slices or chops will lose context-sensitivity and that will cause a reduced precision. Therefore, the strongly connected components must be restricted to control dependence, data dependence, and summary edges, which are intraprocedural edges. If only cycles from those edges are folded, the interprocedural structure will stay intact.

This reduction technique can be applied to both fine- and coarse-grained SDGs. Because the results are very similar, only the results for the fine-grained graphs are presented. This technique is very fast because it has linear complexity—for the analyzed programs it is always sub-second, as shown in table 11.2 on the following page. Again, the first column shows the analyzed program and the next two columns show the time (in seconds) for identifying the strongly connected components in fine-grained SDGs (t_{SCC}) and for folding the components into one single node (t_{fold}). Figure 11.2 on the next page shows the size reduction of the modified SDG (amount of removed nodes and edges): On average[1] 41% of the nodes and 43% of the edges are removed due to folding. For every program, 1000 backward slices have been computed in both the original

[1]The average over all measured percentages.

Program	t_{SCC}	t_{fold}
agrep	0.04	0.06
ansitape	0.02	0.02
assembler	0.07	0.06
bison	0.07	0.09
cdecl	0.03	0.04
compiler	0.04	0.05
ctags	0.03	0.05
diff	0.39	0.45
flex	0.22	0.30
football	0.04	0.05
gnugo	0.01	0.02
loader	0.01	0.01
patch	0.09	0.15
plot2fig	0.00	0.01
rolo	0.21	0.31
simulator	0.03	0.04

Table 11.2: Time needed to fold cycles in SDGs

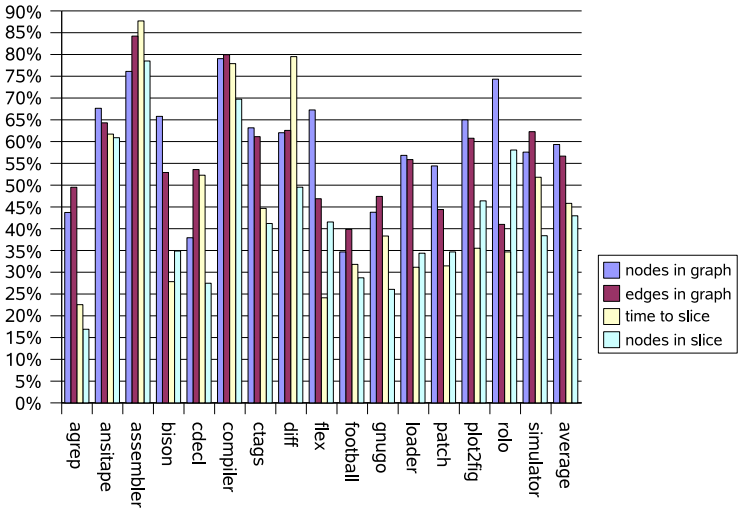


Figure 11.2: Size reduction and effect on slicing

and the folded graph. Because a mapping between original and folded nodes was kept, the slices in the folded graph were computed for the corresponding criterion nodes in the original graph. The time to compute the slices and the average number of nodes in a slice were measured for both graphs. The relative values show a high impact on slicing: On average, slicing needs only 46% of the original time and generates slices with 43% of the original size (no influence on precision; if the nodes are unfolded in the resulting slice it will contain 100% of the original nodes). There is a clear correspondence between the size of the folded graph and the slicing time or slice sizes—the higher the reduction of the graph, the higher the reduction in slicing. The average numbers show that this technique has high impact: The reduced SDGs will only contain around 60% of its original size—without any loss of precision!

Chopping in Folded Graphs

Precise chopping is possible In the presented folded graphs. However, the efficient algorithm for precise chopping relies on the presence of summary edges that can be clearly mapped to call sites. Because folding may merge nodes and edges of different call sites, this mapping is not recoverable from the folded graph. It could easily be maintained externally to the graph, but different to that an experiment has been done, where the folding of parameter and call nodes has been prohibited. Though the reached size reduction will be smaller, no external mapping is required and the chopping algorithm can be applied without modification.

Figure 11.3 on the following page shows the evaluation. Because parameter and call nodes are not allowed to be folded, the size reduction due to folding is much less—it is now only 26% of the nodes and 16% of the edges. Therefore the time needed to chop as well as the average amount of nodes in a chop is not as much reduced: On average, 86% of the original time is needed and produces chops with 64% of the primary size.

Folding Interprocedural Cycles

When interprocedural cycles (i.e. cycles with parameter and call edges) are folded, context-sensitive analyses like slicing and chopping cannot ensure the interprocedurally realizable path property any longer. As presented in sections 7.4 and 10.2.6, the loss of context-sensitivity can cause high imprecision. Because the amount of imprecision is not clear at first sight in this case, another experiment has been conducted. Within the same setup 1000 slices are computed again for every program. Now, the folding disregards the type of nodes and edges and simply folds all nodes and edges of a strongly connected region together. The results, shown in figure 11.4 on the next page, are catastrophic: The folded graph only contains 34% of the original nodes and only 24% of the edges, but the computed slices are extremely imprecise: On average, they have more than four times more nodes if the nodes are unfolded in the resulting slices. In one case (bison) the resulting slices are more than eleven

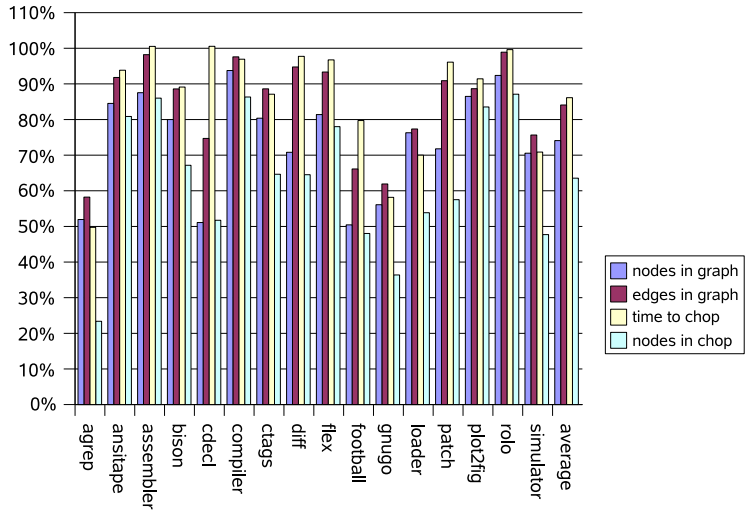


Figure 11.3: Size reduction and effect on chopping

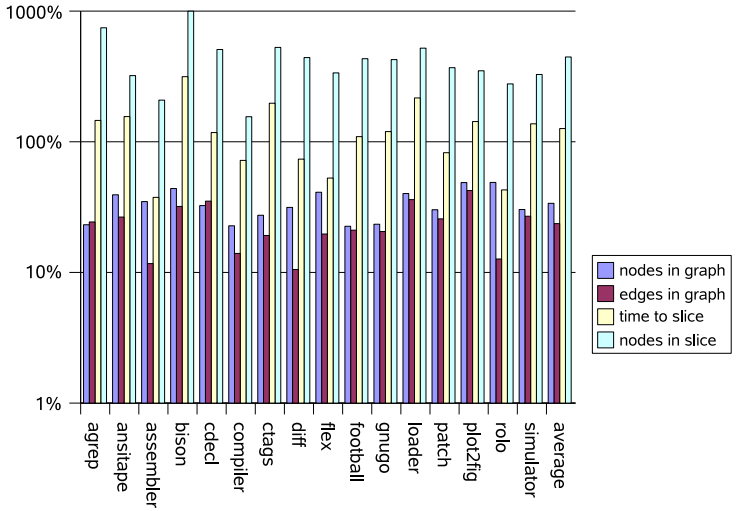


Figure 11.4: Size reduction and effect of context-insensitive folding

```
1  f(x,y) {
2    x = 2*y
3    y = y+1
4  }
5  g() {
6    a = 1
7    b = 2
8    f(a,b)
9    print a
10 }
```

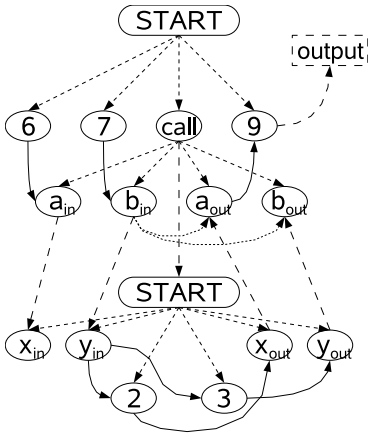


Figure 11.5: Example for a SDG with redundant nodes

times larger! Because the slices are bigger, the time needed for slicing increases instead. These results are backed up by section 10.2.6, where the catastrophic effect of context-insensitivity has already been noted.

11.1.3 Removing Redundant Nodes

One obvious source of imprecision is the usage of GMOD and GREF sets for computation of the nodes representing actual and formal parameters: Because they are based on flow-insensitive computation, it is assumed that any variable modified or referenced in a called procedure is live, i.e. a definition-free path to a usage exists. This is a conservative approximation. It is also assumed that a modified variable is always killed, i.e. no definition-free path through the called procedure exists. This is not a conservative approximation, but if a variable is not killed, a summary edge will be generated restoring the correctness. The opposite assumption that a definition-free path exists would add inaccurate dependence edges, if, in fact, no such path exists.

An interprocedural flow-sensitive analysis could check if all paths through a called procedure are either free of references to variable v or all usages of v must happen after a definition of v (v is not live). If no other path exists, the actual- and formal-in nodes for this variable can be omitted.

Example 11.2: Figure 11.5 shows such an SDG with redundant nodes. In this example the called procedure `f` never uses but always defines the parameter `x` and its formal-in node is redundant (together with the actual-in node for `a`). Also, the value of `b` is not used and its actual-out node is redundant (`b` is not live at the call site). Because this is the only call site of `f`, the formal-out node for `y` is redundant, too (`y` is not live at the exit of the procedure).

If the redundant nodes are removed, slicing will be a little bit more precise: Because of the redundant nodes, a forward slice from line 6 will include the call in line 8. However, line 6 has no impact on the call and can safely be omitted from the slice. The same slice on a graph without redundant nodes will not include the call.

Two independent approaches [FG97] and [LC94b] are based on this insight and presented solutions using flow-sensitive data flow analysis. The approach of [FG97] uses live variables to compute only needed actual and formal nodes. [LC94b] computes for every global and formal parameter if it is *sometimes*, *always* or *never* modified by a call to its function.

Flow-sensitive data flow analysis may be expensive. The following new approach is based on the observation that it is possible to eliminate redundant actual- and formal-in nodes *after* construction of the IPDG or SDG.

- A formal-in node n of a variable that is not live, i.e. always defined before used on all paths through the procedure belonging to n, cannot have an incident data dependence edge. It can be removed from the IPDG including its adjacent actual-in nodes.

- An actual-out node n of a variable that is not live, i.e. always defined before used on all paths to the EXIT node of the procedure belonging to n, cannot have an incident data dependence edge. It can be removed from the IPDG. If the adjacent formal-out node has no other adjacent actual-out node, it can be removed, too.

During the removal of nodes the incident edges are removed, too. If a procedure p does not use or define a variable v directly, but only through calls of other procedures, the IPDG will contain data dependence edges between actual or formal nodes. If such edges are removed because an incident node is removed, the adjacent node must be checked and may probably removed, too. Algorithm 11.1 on page 166 is a worklist algorithm for this approach. The worklist may contain actual or formal nodes, where the four variants are handled differently:

1. A formal-in node can be removed, if it has no incident data dependence edges (which must be outgoing) and therefore no other node is data dependent on it. All incident actual-in nodes are also removed by inserting them into the worklist.

2. A formal-out node can be removed, if it has no incident parameter-out edges—all incident parameter-out edges have been removed at this time due to removal of the adjacent actual-out nodes. If the incident data dependence edges are coming from actual-out or formal-in nodes, such nodes are inserted into the worklist, because the data dependence edge may be the last or only incident data dependence edge.

3. An actual-out node can be removed, if it has no incident data dependence edges (which must be outgoing) and therefore no other node is

data dependent on it. All incident formal-out nodes are inserted into the worklist, because the connecting parameter edge may be the last or only one.

4. An actual-in node can be removed, if it has no incident parameter-in edges—all incident parameter-in edges have been removed at this time due to removal of the adjacent formal-in nodes. If the incident data dependence edges are coming from actual-out or formal-in nodes, such nodes are inserted into the worklist, because the connecting data dependence edge may be the last or only one.

Be aware that the handling of the actual-in and formal-out nodes is similar enough to be merged, which is also possible for the handling of the actual-out and formal-in nodes.

The evaluation of this algorithm has been done on fine- and coarse-grained SDGs. Only the results for the fine-grained SDGs are shown, as the results for coarse-grained SDGs are almost the same. The needed times are shown in table 11.3 on page 167 and the amount of removed nodes and edges is shown in figure 11.6 on page 167. The assessment basically shows:

- The time needed to remove the redundant nodes and edges is very small. For the examples it is always sub-second.

- On average, 9% of the nodes and 13% of the edges are removed. This shows that a significant amount of nodes and edges are redundant.

This approach can easily be modified so that the removal is done in linear time $O(|N|)$, if only nodes are added to the worklist when they have no incident data dependence resp. parameter edges. In that case, each node is visited only once.

The results cannot be compared to the similar approaches of [FG97, LC94b], because these approaches have not been evaluated. However, the approach of [LC94b] can neither identify nor remove redundant actual- or formal-in nodes and will still generate them.

It should be noted that this new reduction of the IPDG size has no negative influence on the precision of slicing. As a benefit, it has a slightly increased precision because of omitted call statements. If this approach is extended to traverse nodes that are normal statements, too, it can be used to detect redundant statements. Such statements have no outgoing edges (no other statement is dependent on them) and can be removed. In the example of figure 11.5 on page 163, this would remove nodes 3 and 6 (the computations of `a` and `y`). This only requires that nodes producing user-visible output are linked to special output nodes (such nodes have already been suggested for other reasons in [OO84]).

Because the presented approach is of linear complexity, we believe that it is more efficient than expensive flow-sensitive and context-sensitive data flow analysis for IPDG computation.

Algorithm 11.1 Removing Redundant Nodes

Input: An IPDG $G = (N, E)$
Output: A probably smaller IPDG $G' = (N', E')$

$W = \{n \in N \mid n \text{ is a formal-in or actual-out node}\}$
while W is not empty **do**
  $W = W/\{n\}$, *remove one node* n *from* W
  if n is a formal-in node **then**
    if n has no incident data dependence edges **then**
      foreach parameter edge $m \rightharpoonup n \in E$ **do**
        add all adjacent (actual-in) nodes to the worklist
        $W = W \cup \{m\}$
      remove n
      $N = N/\{n\}$
      $E = E/\{e \mid e = n \rightharpoonup x \vee e = x \rightharpoonup n\}$
  if n is a formal-out node **then**
    if n has no incident parameter edges **then**
      foreach data dependence edge $m \rightharpoonup n \in E$ **do**
        add all adjacent actual or formal nodes to the worklist
        if m is an actual or formal node **then**
          $W = W \cup \{m\}$
      remove n
      $N = N/\{n\}$
      $E = E/\{e \mid e = n \rightharpoonup x \vee e = x \rightharpoonup n\}$
  if n is a actual-out node **then**
    if n has no incident data dependence edges **then**
      foreach parameter edge $m \rightharpoonup n \in E$ **do**
        add all adjacent (formal-out) nodes to the worklist
        $W = W \cup \{m\}$
      remove n
      $N = N/\{n\}$
      $E = E/\{e \mid e = n \rightharpoonup x \vee e = x \rightharpoonup n\}$
  if n is a actual-in node **then**
    if n has no incident parameter edges **then**
      foreach data dependence edge $m \rightharpoonup n \in E$ **do**
        add all adjacent actual or formal nodes to the worklist
        if m is an actual or formal node **then**
          $W = W \cup \{m\}$
      remove n
      $N = N/\{n\}$
      $E = E/\{e \mid e = n \rightharpoonup x \vee e = x \rightharpoonup n\}$
  return G, *the reduced IPDG*

Program	time
agrep	0.03
ansitape	0.02
assembler	0.07
bison	0.06
cdecl	0.02
compiler	0.02
ctags	0.02
diff	0.30
flex	0.17
football	0.03
gnugo	0.01
loader	0.00
patch	0.09
plot2fig	0.01
rolo	0.19
simulator	0.01

Table 11.3: Evaluation

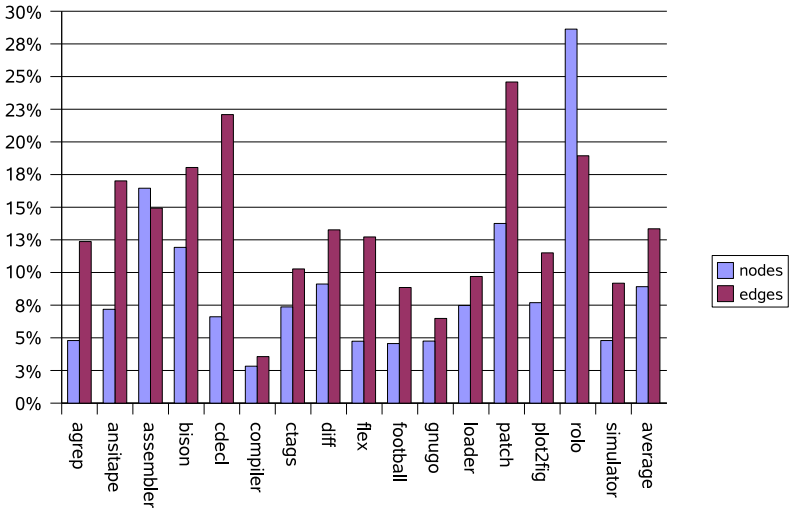


Figure 11.6: Size reduction (amount of nodes and edges removed)

11.2 Increasing Precision

As already been seen in section 11.1.3, the system dependence graphs contain enough information to do analyses equivalent to intra- and interprocedural flow- and context-sensitive data flow analysis. This information can be used to revert the influence of imprecise data flow analysis used for construction of the system dependence graph. Section 11.1.3 contains the first application, which compensates the flow-insensitive interprocedural data flow analysis for GREF and GMOD. The following sections will present more applications like improving the precision of call graphs, constant propagation and common subexpression elimination.

11.2.1 Improving Precision of Call Graphs

As presented in Section 14.1.3, the call graph in the VALSOFT system is computed by a very imprecise solution: First, it is assumed that function variables can only contain functions that have been used in an expression. Also, at call sites that use function variables is assumed that all functions with matching signatures can be called. This approach results in an imprecise call graph. Instead of using a more precise call graph construction approach, this section will present the approach of removing calls that cannot be realized from the program dependence graph.

This approach relies on a special feature of the program dependence graph: If a procedure p can be called at a call site c, the function constant p must reach c via a realizable path of data dependence and parameter edges. This leads to a straightforward technique, where for any call edge $c \rightharpoonup p$ is checked that p is used as a function constant in a backward *data slice* $S^D(c)$. A data slice is a slice that ignores control dependence and call edges, traversing only data dependence and parameter edges. If the function constant p is not used in $S^D(c)$, the call $c \rightharpoonup p$ can safely be removed from the program dependence graph together with its parameter edges. The algorithm 11.2 repeats this removal until a (maximal) fixed point is reached.

Algorithm 11.2 Removing Unrealizable Calls

Input: $G = (N, E)$ the given IPDG
Output: $G' = (N, E')$ the IPDG with unrealizable calls removed

repeat
  foreach call edge $c \rightharpoonup p$ **do**
    if $\exists n \in S^D(c) : n$ uses function constant p **then**
      Remove $c \rightharpoonup p$ and all corresponding parameter edges from E
until no more edges are removable
return $G' = (N, E)$

This algorithm has four weak spots:

1. The computed call graph includes calls from dead procedures (procedures that are never called). This can easily be fixed by removing dead code first[2].

2. If the data slice S^D is computed within the system dependence graph using summary edges, the summary edges must be computed differently to ignore control dependence edges. Otherwise a small imprecision of S^D would be caused.

3. With real world but *flawed* C programs, the algorithm may encounter a procedure `p` used as function constant in a node reaching a call site `c` which does not call `p`. In these situations, the procedure `p` has a signature that does not match the signature of the call at `c` and the approximation of the call graph will not record that `p` may be called at `c`. Figure 11.7 on the following page shows such an example: Despite that procedure `f` has a signature that does not match the signature of `p1`, the assignments circumvent the type rules leading to an illegal call of `f` with too few arguments.

   Such situations are considered flawed programs and are reported to the user, who has the obligation to fix the program.

4. The IPDG may contain invalid dependence edges because the pointer analysis which is needed to compute the data dependences has used the inaccurate call graph.

An evaluation has showed that the approach to first compute a simple approximation of the call graph and improving it on demand within the program dependence graph is sensible. In typical C programs, function variables are rarely used and if they are used, the signatures are distinct. In the presented set of test cases, only two test cases had imprecise call graphs that could be improved with algorithm 11.2 on the preceding page.

11.2.2 Constant Propagation

The goal of constant propagation is to find all variable accesses of a program where the variable always has the same (constant) value. If a compiler can identify such variable accesses, it can replace the variable by the constant. If the constants are propagated through the program, it may be possible that more variable accesses will be constant. It may even be possible to eliminate branches of an if-statement if the predicate is constant. Constant propagation can also be used in program dependence graphs to replace variable accesses by constants, eliminate data dependence edges or eliminate complete branches of if-statements. Binkley [Bin94] has presented such an approach, based on

[2]Dead code removal increases the precision of backward slicing, because dead code may reach the criterion but actually cannot influence the criterion.

```
void f (int x, int y) {
  ...
}

void main (void) {
  void (*p0)();
  void (*p1)(int);
  p0 = f;
  p1 = p0;
  p1(1);
}
```

Figure 11.7: A flawed C program

Selke's graph rewriting semantics for program dependence graphs [Sel89]. The idea is to find all nodes containing constant values. The constant value is then propagated along the outgoing data dependence edges. A variable usage in a node may be replaced by a constant if all incoming data dependence edges propagate the same constant value. The replacement will eventually make more nodes constant and the propagation and replacement is repeated until no more constants are found. If a predicate of an if-statement is found to be constant, the statement is replaced by the statements of one branch. Loops are unrolled if such a transformation makes the predicate constant for some iterations.

Binkley has used this approach to find constants and enable a compiler to use that information. For the analysis system underlying this thesis his approach has been implemented to make the program dependence graph smaller by eliminating nodes.

11.2.3 Common Subexpression Elimination

Common subexpression elimination is another compiler optimization technique. It is basically used to identify situations where a computation is repeated. The second computation is not needed because the result of the first can be reused (if it is stored temporarily). The elimination inserts assignments that save the result of the first computation and replaces the second computation by the temporary variable.

Example 11.3: The following example shows how a code fragment is optimized:

```
...
a = b + c + d;
e = b + c + q;
...
```

is transformed to

```
...
t = b + c;
a = t + d;
e = t + q;
...
```

Data flow analysis is needed to identify common subexpressions, because the used variables are not allowed to change between the execution of the first and second subexpression.

Example 11.4: The following code cannot be transformed because variable `c` might change in between:

```
...
a = b + c + d;
if (...)
  c = x;
e = b + c + q;
...
```

Common subexpression elimination can also be used to compute smaller program dependence graphs. One way is to employ common subexpressions during data flow analysis and compute the program dependence graph based on the optimized code. However, it is also possible to compute the fine-grained program dependence graph as before and do common subexpression elimination directly in the graph: The fine-grained structure enables the identification of subexpressions with the same composition (except of commutative operations) and the data dependence edges can be used to ensure that both subexpressions compute a common subexpression.

Definition 11.1 (Common Subexpressions in the PDG)
Two subexpressions in a PDG are *common subexpressions*, iff

1. The subgraphs consisting of nodes and value and expression dependence edges are isomorphic.
2. The composition is the same except for commutativity.
3. The incoming data dependence edges for equivalent nodes are coming from the same nodes.

This definition ensures that the subgraphs are common subexpressions and the requirement for the data dependence edges ensures that one subexpression can be eliminated without changing the result.

The elimination is straightforward, because no temporary variables are needed. Common subexpressions always form a subgraph with one incoming expression dependence edge, one outgoing value dependence edge and the set of incoming data dependence edges. One of the subexpressions can be eliminated by first rerouting the value and expression dependence edge to the other subexpression and removing the nodes as well as the incoming data dependence edges.

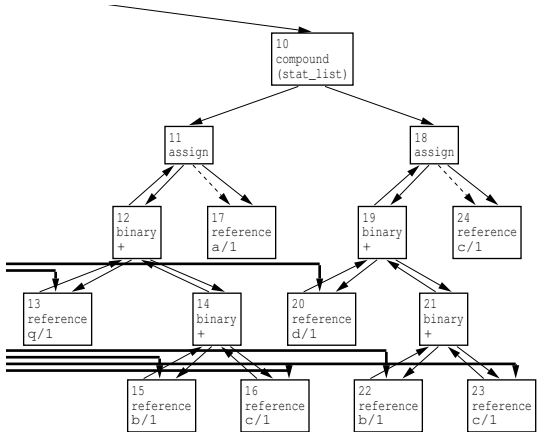


Figure 11.8: PDG before common subexpression elimination

Example 11.5: Figure 11.8 shows a fragment of a PDG before common subexpression elimination (the PDG corresponds to the earlier example). The subtree consisting of nodes 14, 15 and 16 is isomorphic to the subtree consisting of nodes 21, 22 and 23. Because the incoming data dependence edges are coming from the same nodes (not shown), one of the subtrees can be eliminated. Figure 11.9 on the next page shows the modified PDG. The second subexpression is connected to node 19 and the connecting value and expression dependence edges have been rerouted to the first subexpression at node 14. The nodes 21, 22 and 23 of the second subexpression have been eliminated together with incident edges.

Of course, program dependence graphs where common subexpressions have been eliminated have no higher precision, only a decreased size. For slicing, a slight unpleasantness is caused: If a slicing criterion which includes a node from a common subexpression is chosen, all equivalent nodes in the other subexpressions will also be included into the criterion.

11.3 Related Work

[FG97] also eliminates redundant nodes for actual and formal parameters using interprocedural data flow analysis instead of GMOD and GREF. Another similar approach is [LC94b], which computes may-modify, must-modify and kill sets at call sites.

In [HHD+01] a theory to merge nodes of control flow graphs is presented: a node coarsening calculi.

The presented folding of cycles is not restricted to dependence graphs and slicing. Cycle elimination is a very general optimization, for example, it is

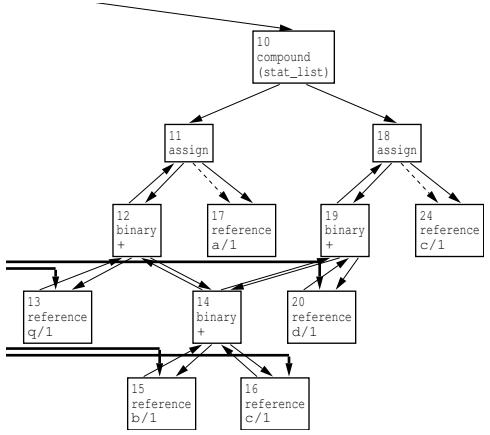


Figure 11.9: PDG after common subexpression elimination

essential for fast pointer analysis [FFSA98, RC00, HT01]. It is applicable to all graph based reachability problems. Even for context-free language reachability problems [Rep98, RHS95], where (not necessarily procedure based) context-sensitive analyses are done, the folding technique can reduce the size of the graphs.

The presented removal of unrealizable calls in dependence graphs is similar to Lakhotia's approach [Lak93] to construct call graphs from dependence graphs. He uses the (incomplete) system dependence graph to propagate function constants to call sites where function variables are used. If a function constant reaches such a call site, a call to the function is inserted there. However, Lakhotia's approach cannot be used for programs with global variables or pointers (unlike the approach in section 11.2.1). Besides using the dependence graphs for constant propagation [Bin94] or call graph construction [Lak93], Bodik and Gupta [BG97] presented partial dead code elimination with dependence graphs and slicing.

The applications of the system dependence graphs in this chapter to do analyses equivalent to data flow analyses are similar to program analysis via context-free language reachability [Rep98, RHS95], which bases the analysis on reachability problems in (interprocedural) graphs.

Chapter 12

Identifying Similar Code

Duplicated code is common in all kinds of software systems. Although cut-copy-paste (-and-adapt) techniques are considered bad practice, every programmer uses them once in a while. Code duplication is easy and cheap during software development, but it makes software maintenance more complicated:

- Errors may have been duplicated together with the duplicated code.
- Modifications of the original code often must also be applied to the duplicated code.

Especially for software renovation projects, it is desirable to detect duplicated code; a number of approaches have been developed [KH01, DRD99, BYM+98, Kon97, MLM96]. These approaches are graph-based [KH01], text-based (and language independent) [DRD99], syntax-based [BYM+98] or based on metrics (syntax- and/or text-based) [Kon97, MLM96]. Some approaches can only detect (textual or structural) identical duplicates, which are not typical in software systems as most duplicates are adapted to the environment where they are used.

In Figure 12.1 on the following page two similar pieces of code in `main.c` from the `agrep` program are shown that have been detected as duplicates by our prototype tool. Assuming that the left part is the original and the right part is the duplicate, some typical modifications in the duplicate can be identified:

1. Parts of the code will be executed under different circumstances (lines 742 and 743 have been moved into an `if` statement in lines 473-476).
2. Variables and/or expressions are changed (lines 743/478, 747/483, ...).
3. Parts of the code are inserted or deleted ("`lasti = i-1`" in line 758).
4. Code is moved to different locations ("`j++`" in line 481/748).

Modifications disturb the structure of the code and duplicated code is more complicated to identify. This causes a trade-off between precision (amount

```
740 if(c != Newline)
741 {
742   r1 = Init1 & r3;
743   r2 = (Next[r3] & CMask) | r1;
744 }
745 else {
746   r1 = Init1 & r3;
747   r2 = Next[r3] & CMask | r1;
748   j++;
749   if(TAIL) r2 = Next[r2] | r2 ;
750   if(( r2 & 1) ^ INVERSE) {
751     if(FILENAMEONLY) {
752       num_of_matched++;
753       printf("%s\n", CurrentFileName);
754       return;
755     }
756     r_output(buffer, i-1, end, j);
757   }
758   lasti = i - 1;
759   r3 = Init0;
760   r2 = (Next[r3] & CMask) | Init0;
761 }
762 c = buffer[i++];
763 CMask = RMask[c];
...

472 if(c != Newline)
473 { if(CMask != 0) {
474   r1 = Init1 & r3;
475   r2 = ((Next[r3>>hh] | Next1[r3&LL]) & CMask) | r1;
476  }
477  else  {
478   r2 = r3 & Init1;
479  }
480 }
481 else { j++;
482   r1 = Init1 & r3;
483   r2 = ((Next[r3>>hh] | Next1[r3&LL]) & CMask) | r1;
484   if(TAIL) r2 = (Next[r2>>hh] | Next1[r2&LL]) | r2;
485   if(( r2 & 1 ) ^ INVERSE) {
486     if(FILENAMEONLY) {
487       num_of_matched++;
488       printf("%s\n", CurrentFileName);
489       return;
490     }
491     r_output(buffer, i-1, end, j);
492   }
493   r3 = Init0;
494   r2 = (Next[r3>>hh] | Next1[r3&LL]) & CMask | Init0;
495
496 }
497 c = buffer[i++];
498 CMask = Mask[c];
...
```

Figure 12.1: Two similar pieces of code from agrep

of false positives) and recall (amount of undiscovered duplicates) in text- or structure-based detection methods. To also detect non-identical but similar duplicates (increased recall), the methods have to ignore certain properties. However, this may lead to false positives (reduced precision). This trade-off has been studied in [Kon97]. Some of the approaches suffer the *split duplicates* symptom: A simple modification in a duplicate causes a detection of two independent duplicates: one duplicate for the code before the modifications and one after it. If the unchanged parts are too small, the duplicate cannot be identified.

The following approach is not affected by the trade-off between recall and precision and modified duplicates can still be detected. Such an approach cannot just be based on text or syntax but has to consider semantics, too. The presented approach is based on the fine-grained program dependence graphs, which represent the structure of a program and the data flow within it. In these graphs, we try to identify similar subgraph structures which stem from duplicated code. Identified similar subgraphs can be mapped back directly onto the program code and presented to the user.

The benefit of fine-grained program dependence graphs is the structural representation of expressions and the richness of the attributes that ease the identification of similar or identical nodes and edges. This is presented next, including a prototype implementation together with an evaluation, showing that this approach is feasible even with the non-polynomial complexity of the problem.

12.1 Identification of Similar Subgraphs

Program dependence graphs are *attributed directed graphs*, where the node attributes are the class, the operator and the value of the nodes and the edge attributes are the class and the label of the edges.

An *attributed directed graph* is a 4-tuple $G = (N, E, \mu, \nu)$ where N is the set of nodes, $E \subseteq N \times N$ is the set of edges, $\mu : N \to A_N$ maps nodes to the node attributes and $\nu : E \to A_E$ maps edges to the edge attributes. Let $\Delta : E \to A_N \times A_E \times A_N$ be the mapping $\Delta(n_1, n_2) = (\mu(n_1), \nu(n_1, n_2), \mu(n_2))$.

Definition 12.1 (Node-Edge-Path)
A *node-edge-path* is a finite sequence $n_0, e_1, n_1, e_2, n_2, \ldots, e_l, n_l$ of edges and nodes where $e_i = (n_{i-1}, n_i)$ for all $1 \leqslant i \leqslant l$. A k-*limited node-edge-path* is a path $n_0, e_1, n_1, e_2, \ldots, e_l, n_l$ with $l \leqslant k$.

Different to normal paths, node-edge-paths contain both visited edges and nodes.

Two attributed directed graphs $G_1 = (N_1, E_1, \mu_1, \nu_1)$ and $G_2 = (N_2, E_2, \mu_2, \nu_2)$ are *isomorphic*, if a bijective mapping $\phi : N_1 \to N_2$ exists with:

$$(n_i, n_j) \in E_1 \iff (\phi(n_i), \phi(n_j)) \in E_2, \quad \Delta_1(n_i, n_j) = \Delta_2(\phi(n_i), \phi(n_j))$$

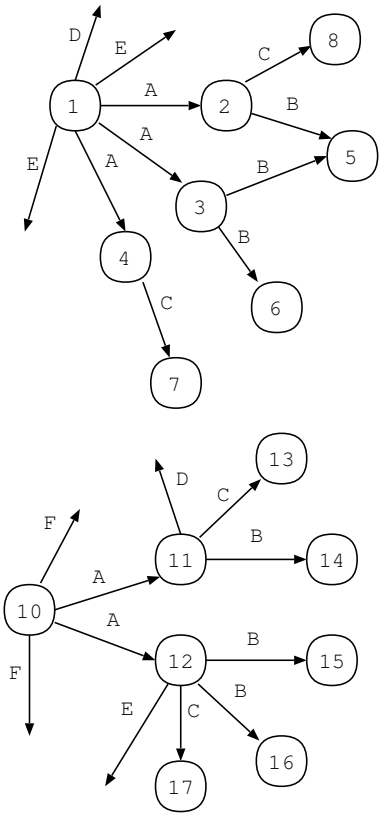


Figure 12.2: Two simple graphs

This means that two graphs are isomorphic iff every edge is matched bijectively to an edge in the other graph and the attributes of the edges and incident nodes are the same. The question *whether two given graphs are isomorphic* is NP-complete in general.

Example 12.1: In Figure 12.2 two simple attributed graphs are shown, where the edge labels represent the complete attribute-tuple of the node and edge attributes. There is no single pair of maximal isomorphic subgraphs, at least three exist with six nodes each:

$$\phi_1: \begin{array}{l} 1 \leftrightarrow 10 \\ 4 \leftrightarrow 11 \\ 3 \leftrightarrow 12 \\ 7 \leftrightarrow 13 \\ 5 \leftrightarrow 15 \\ 6 \leftrightarrow 16 \end{array} \qquad \phi_2: \begin{array}{l} 1 \leftrightarrow 10 \\ 2 \leftrightarrow 11 \\ 4 \leftrightarrow 12 \\ 8 \leftrightarrow 13 \\ 5 \leftrightarrow 14 \\ 7 \leftrightarrow 17 \end{array} \qquad \phi_3: \begin{array}{l} 1 \leftrightarrow 10 \\ 2 \leftrightarrow 11 \\ 3 \leftrightarrow 12 \\ 8 \leftrightarrow 13 \\ 5 \leftrightarrow 14 \\ 6 \leftrightarrow 15 \end{array}$$

For the current purpose it is more interesting to look at *similar* subgraphs which do not have to be isomorphic. Defining something to be similar is always tricky, as similarity is nothing precise but something vague. Nevertheless, similarity between graphs is defined by relaxing the mapping between edges:

Definition 12.2 (Matching Paths)
A node-edge-path $n_0, e_1, n_1, e_2, \ldots, e_k, n_k$ is *matching* another node-edge-path $n'_0, e'_1, n'_1, e'_2, \ldots, e'_k, n'_k$, if the sequence of node and edge attributes are identical:

$$\forall 1 \leqslant i \leqslant k : \Delta(e_i) = \Delta'(e'_i)$$

Definition 12.3 (Similar Graphs)
Two graphs G and G′ are considered to be *similar graphs* if for every path $n_0, e_1, n_1, e_2, \ldots, e_k, n_k$ in one graph a matching path $n'_0, e'_1, n'_1, e'_2, \ldots, e'_k, n'_k$ exits in the other graph. Additionally, all paths have to start at a single node n in G and at n' in G′ ($n_0 = n, n'_0 = n'$ for all such paths).

A naive approach to identify the maximal similar subgraphs would now calculate all (cycle free) paths starting at n and n' and would do a pairwise comparison afterwards. Of course, this is infeasible and even if the paths are length limited, the maximal length would be unusable small.

Our approach constructs the maximal similar subgraphs by induction from the starting nodes n and n' and matching length limited paths. What makes this approach feasible is that it considers all possible matchings *at once*. In many cases, an edge under consideration can be matched to more than one other edge. Instead of checking every possible pair, the algorithm checks the complete set of matching edges.

Example 12.2: This approach can be seen best at the example in figure 12.2 on the facing page:

1. The algorithm starts with $n = 1$ and $n' = 10$. These nodes are considered the endpoints of matching paths of the length zero.

2. Now, the matching paths are extended: The incident edges are partitioned into equivalence classes based on the attributes. There is only one pair of equivalence classes that share the same attributes in both graphs: $\{(1,2),(1,3),(1,4)\}_A$ and $\{(10,11),(10,12)\}_A$.

3. The reached nodes are now marked as being part of the maximal similar subgraphs and the algorithm continues with the sets of reached nodes $\{2,3,4\}$ and $\{11,12\}$.

4. Again the incident edges are partitioned into the first pair of edge sets $\{(2,5),(3,5),(3,6)\}_B$ and $\{(11,14),(12,15),(12,16)\}_B$ and the second pair $\{(4,7),(2,8)\}_C$ and $\{(11,13),(12,17)\}_C$. The algorithm continues recursively for both pairs.

5. The reached nodes $\{5, 6\}$ and $\{14, 15, 16\}$ are marked as parts of the maximal similar subgraphs. No edges are leaving these nodes.
6. The other set pair of reached nodes $\{7, 8\}$ and $\{13, 17\}$ are marked. No edges leave these nodes.
7. As no more set pairs exist, the algorithm terminates.

In the end, the algorithm has marked the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$ together with $\{10, 11, 12, 13, 14, 15, 16, 17\}$ which induce the maximal similar subgraphs. By accident, this is identical to the union of all maximal isomorphic subgraphs. In general, it will only be similar to maximal isomorphic subgraphs.

A simplified version of the algorithm is shown in Figure 12.1. It calculates the maximal similar subgraphs G_1 and G_2 which are induced by k-limited paths starting at the nodes n_1 in G_1 and n_2 in G_2. We call these graphs *maximal similar k-limited path induced subgraphs* $G^k_{n_1}$ *and* $G^k_{n_2}$.

Algorithm 12.1 Generate $G^k_{n_1}$ and $G^k_{n_2}$

Input: An attributed directed graph $G = (N, E, \mu, \nu)$
Start nodes $n_1 \in N$ and $n_2 \in N$
Maximal path length k
Output: Similar subgraphs $G^k_{n_1}$ and $G^k_{n_2}$

proc `propagate`(N_1, N_2, l):
$N_1 \subset N$ *and* $N_2 \subset N$ *are the endpoints of similar paths*
if $l \leqslant k$ **then**
Let E_1 and E_2 be the edges leaving the nodes of N_1 and N_2
Partition E_1 and E_2 into equivalence classes E^i_1 and E^i_2 based on Δ
foreach E^i_1 with their corresponding E^i_2 **do**
Add edges from E^i_1 and E^i_2 to $G^k_{n_1}$ and $G^k_{n_2}$
Let N^i_1 and N^i_2 be the nodes reached by the edges in E^i_1 and E^i_2
Call `propagate`$(N^i_1, N^i_2, l + 1)$

Initialize G_{n_1} and G_{n_2} to be empty
Call `propagate`$(\{n_1\}, \{n_2\}, 1)$
return $G^k_{n_1}$ and $G^k_{n_2}$ as result

Before maximal similar k-limited path induced subgraphs G^k_n and $G^k_{n'}$ can be found, the possible pairs (n, n') have to be detected. A naive approach would check all pairs $N \times N$, leading to a complexity of $O(|N|^2)$ (independent of the complexity of the subgraphs' generation themselves). Even with smarter approaches, this complexity cannot be reduced. Therefore, only a subset of N should be considered as "starting" nodes, as most other nodes are reached during the construction of the maximal subgraphs. This subset should be based on specific features of the nodes which are highly application specific.

12.2 Implementation

To find similar code based on identifying maximal similar subgraphs in fine-grained PDGs, the subset of the nodes has to be identified that are used in the pairwise construction of the subgraphs. The chosen nodes are entry and predicate nodes: Starting at entry nodes the algorithm will identify similar procedures. Starting at predicate nodes it finds similar code fragments independent of procedures. For every pair of predicate or entry nodes (n_1, n_2) the maximal similar $G^k_{n_1}$ and $G^k_{n_2}$ are generated. The generation is basically a recursive implementation of the induction from algorithm 12.1 on the facing page.

12.2.1 Weighted Subgraphs

Taking the subgraphs as direct result, they just represent *structural* similarity which can also be achieved via less expensive techniques like [BYM+98]. The subgraphs can be large even if they do not have a similar semantic. The reason is that the data dependence edges may not match and the subgraphs are mainly induced by control dependence edges. For example, two nodes A and B may be included in the subgraph because they are reached by control dependence edges which match in the similar subgraph. It is possible that a data dependence edge from A to B is not included in the subgraph, because there is no matching edge in the similar subgraph. Only if the data dependence edges are considered special it is guaranteed that the subgraphs have a similar semantic.

Therefore the constructed subgraphs have to be weighted. A simple criterion is just the number of data dependence edges in the subgraphs. As our evaluation in the next section shows, this criterion is good enough. However, other, more sophisticated criteria are possible like the percentage of data dependence edges or the amount and length of paths induced by data, value and reference dependence edges.

Another possibility would be to reduce the constructed subgraphs, which are edge induced, to a connected node induced component. In that case no pair of nodes exists that is connected by an edge not included in the subgraph.

12.2.2 Visualization

To visualize the detected fragments of similar code, a tool has been implemented that shows the fragments as HTML files to be presented in any browser: For any pair of identified similar subgraphs, the nodes are projected onto ranges in the source code (see section 9.2). The identified ranges are then highlighted in the source code of the enclosing procedures.

Example 12.3: Figure 12.3 on the next page shows the visualization for the `agrep` program: To the left a list of identified pairs of similar code is presented. If one of that entries is selected, the similar fragments of that pair is shown in the upper and lower window.

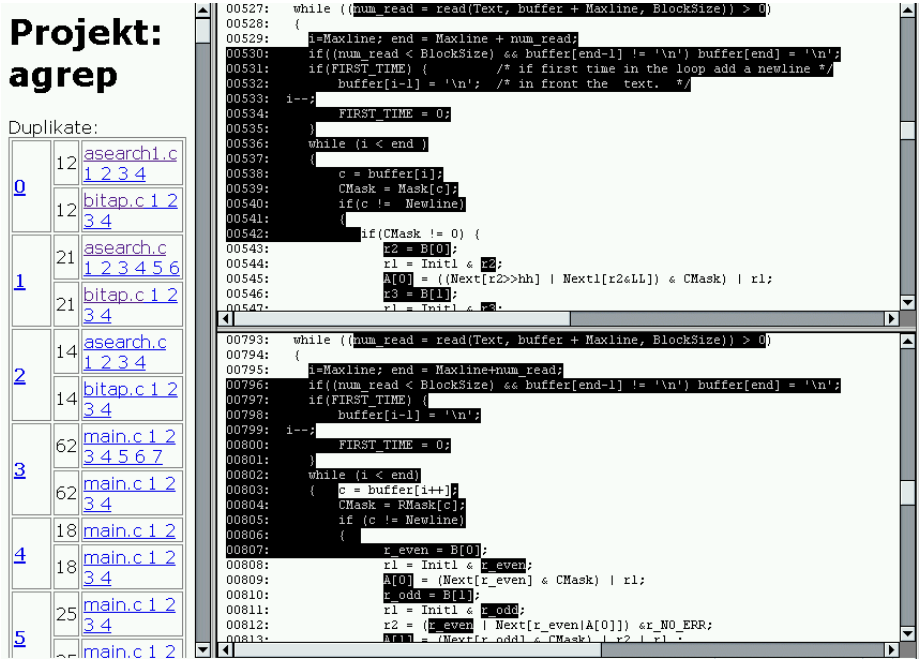


Figure 12.3: Visualization of similar code fragments

Project	LOC	Edges	Nodes
`agrep`	3968	69032	22588
`bison`	8303	79030	28071
`cdecl`	3879	40578	12939
`compiler`	2402	99219	16497
`ctags`	2933	45249	12446
`diff`	17485	169508	43518
`fft`	3242	35701	16446
`flex`	7640	124730	37073
`football`	2261	63833	18718
`larn`	10410	817432	158077
`patch`	7998	196106	29766
`rolo`	5717	50816	17438
`simulator`	4476	34939	14438
`spim`	19739	1338294	122819
`twmc`	24950	1605532	181281

Table 12.1: Some test cases

12.3 Evaluation

As any other k-limited technique, the presented work had to be 'tuned' to find an appropriate value for k. Therefore, a set of test programs stemming from different sources was checked for duplicated code. Early evaluations showed that points-to information has a small *negative* influence on the precision of identified subgraphs. Because the used points-to information is flow-insensitive, it normally just adds data dependence edges. This enables more possible similar paths in a subgraph and therefore larger similar subgraphs. The test cases shown in Table 12.1 are all PDGs generated *without points-to information*. The size of the programs are given in terms of lines of code and the number of nodes and edges in the PDG.

The running times can be seen for some examples in Table 12.2 on the next page. For different limits between $k = 10$ and $k = 50$ the running times are given (measured in seconds of user time spent). Some plots, showing the running times in more detail for different k, will follow; more plots can be found in the Appendix. A direct relation between the size of a program and the running time does not exist as the running time is mostly dependent on the size and the amount of similar subgraphs within a program. However, due to the pairwise comparison a quadratic complexity is expected.

Table 12.3 on the following page shows the amount of discovered duplicates with a minimum weight of 10, 20 and 50. The limit used was $k = 20$ and only minimal differences exist for larger k (except for `twmc`). Again, some plots will follow, showing the amount of discovered duplicates in more detail—completed by more plots in the Appendix.

Project	Time f. limit k (sec)					
	k=10	k=20	k=30	k=40	k=50	k=100
agrep	26.4	207.9	1465	7150	38848	-
bison	8.9	47.4	249.2	714.5	920.3	921.6
cdecl	0.6	0.6	0.6	0.6	0.6	0.6
compiler	226.8	237.6	237.6	237.6	237.6	237.6
ctags	0.6	0.8	0.8	0.8	0.8	0.8
diff	2.5	9.1	32.0	61.4	63.6	63.6
fft	6.0	53.4	297.2	892.9	1292	1296
flex	3.3	3.8	4.2	4.3	4.3	4.3
football	30.3	49.9	54.7	54.7	54.7	54.7
larn	271.4	4242	5878	5905	5867	5876
patch	6.3	7.5	8.6	9.2	9.3	9.2
rolo	0.7	0.7	0.7	0.7	0.7	0.7
simulator	1.4	2.4	2.6	2.6	2.6	2.6
spim	525.9	703.5	798.5	809.1	809.1	807.2
twmc	918.4	24263	-	-	-	-

Table 12.2: Running times

Project	Duplicates		
	$\geqslant$10	$\geqslant$20	$\geqslant$50
agrep	155	91	12
bison	34	22	0
cdecl	0	0	0
compiler	94	67	51
ctags	0	0	0
diff	40	10	0
fft	16	14	8
flex	16	0	0
football	50	2	0
larn	91	53	6
patch	2	0	0
rolo	0	0	0
simulator	0	0	0
spim	30	16	0
twmc	1383	992	639

Table 12.3: Sizes

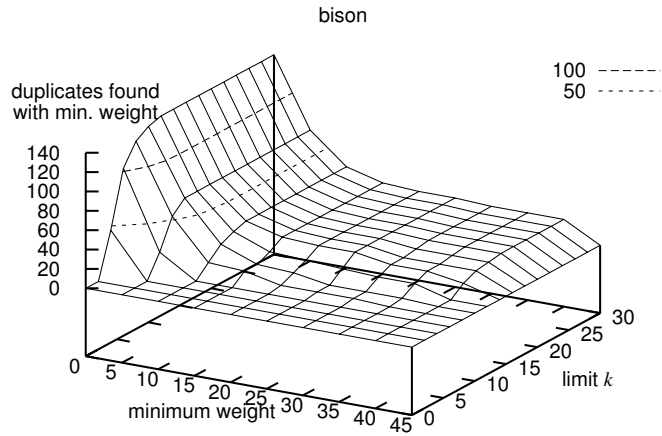


Figure 12.4: Results for `bison`

12.3.1 Optimal Limit

To ensure highest possible recall, a very high k-limit is desirable. However, this is not possible due to the exponential complexity of the graph comparison (e.g. test case `twmc` did not finish within the limit of 50 hours for k larger than 25). Now, the claim is that a small k is sufficient and that a limit above this value will not increase recall. It was found to be true for almost any test case. A typical case is `bison`, for which the results are shown in Figure 12.4. All test cases were repeated for limits $0 \leqslant k \leqslant 30$ (y-axis). Also shown is how many duplicates (z-axis) are reported being above a specific minimum weight (y-axis). As it can be seen, for very small k ($< 5-10$) almost no duplicates are reported. For larger (but still small) k ($< 15-20$) the amount of reported duplicates is increasing fast, for larger k (> 20) it is not changing any more. This has been found to be the same for almost any other test case—a k-limit around 20 seems to be sufficient for highest recall.

Because the running times are exalting for large k, the presented algorithm is a good candidate for an *any-time algorithm*. Any-time algorithms can be interrupted at any time and still give usable results. The longer the algorithm runs, the better results it produces. An any-time version of the presented algorithm would use iterative-deepening [Kor85], because a breadth-first version produces exponentially growing temporary data that cannot be stored. The iterative-deepening version starts from a small k and analyzes the complete program very fast with that small k; the results are stored. Then, the process is redone with an increased k (e.g. $k' = k + 1$). If the process is not interrupted, the results replace the last stored results. This repeats until the process

is interrupted or a maximal k is reached, then the stored results for the last k are returned. As long as the running times increase exponentially for larger k (which is the case for many analyzed programs, see the Appendix), the repetition of computations for small k has only a small effect on running times.

12.3.2 Minimum Weight

The other "tunable" parameter in the presented technique is the minimum weight that a similar subgraph must have to be reported. This value is not critical like the k-limit, as it does not influence the comparison itself. Usually, all possible duplicates are identified independent of their weights and the minimum weight just changes the amount of *reported* duplicates. The `bison` test case is an ideal example: for small minimum weights, many duplicates are reported. For larger minimum weights this changes quickly, which shows that the majority of duplicates are small pieces of code. For minimum weights between 10 and 40, around 40 duplicates are reported. For minimum weights above 45, no duplicates are reported, which shows that the maximum weight of all duplicates is less than 45.

It has been discovered that there is no "ideal" minimum weight as every test case has different amounts of reported duplicates with varying minimum weights. This is not unexpected, as duplication is different in every program. Different from k, the minimum weight can be tuned *after* the identification has finished, during presentation to the user.

12.3.3 Running Time

Figure 12.5 on the facing page shows the times for the `bison` example, which are increasing exponentially for large k. It was claimed that a k-limit around 20 is optimal for recall: 47 seconds are needed to analyze `bison` under this limit. For some test cases an interesting behavior has been found—the running time does not increase exponentially but reverse logarithmic for increased k. This is shown in Figure 12.6 on the next page for the test case `compiler`. In figure 12.7 on page 188 it can also be seen that for k-limits larger than ten the amount of reported duplicates stays the same (50 duplicates with a weight larger than 50). This means that there are no similar paths longer than 10 edges in that software and the limit is not reached for larger k-limits. The result is that the time needed to calculate the similar graphs is independent of k for $k > 10$. Thus, the overall needed time is not changing above that. The same behavior can be seen for most of the test cases in Figure 12.2 on page 184: Only two of the test cases have differences in running time for the limits $k = 50$ and $k = 100$. For all others, even the amount of reported duplicates does not change for $k > 20$ (not shown).

One of the test cases (see figure 12.8 on page 188) was different from all the others: First of all, k-limits larger than 25 could not be tested because the running time was already at 46 hours. Also, the amount of reported duplicates was incredibly high: more than 500 with a weight larger than 50 and more

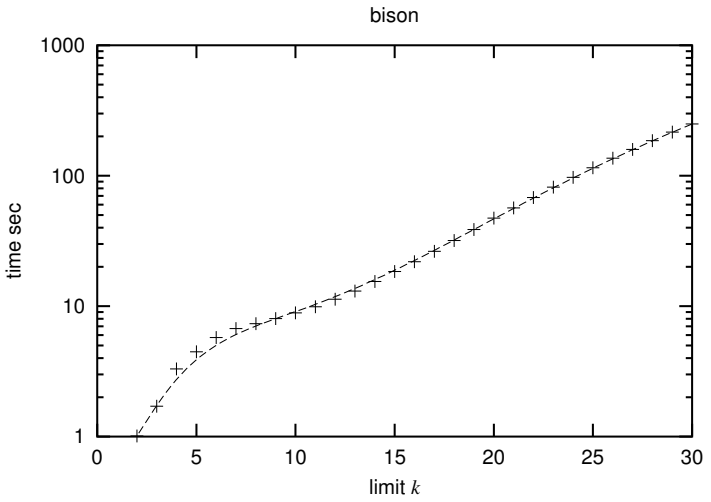


Figure 12.5: Running times of `bison`

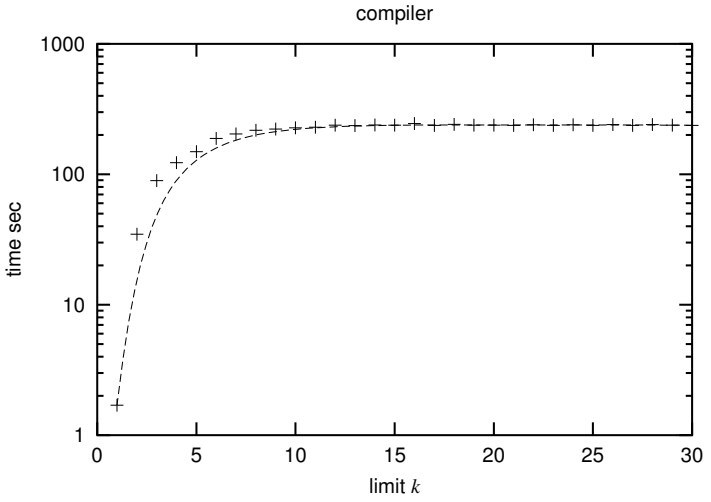


Figure 12.6: Running times of `compiler`

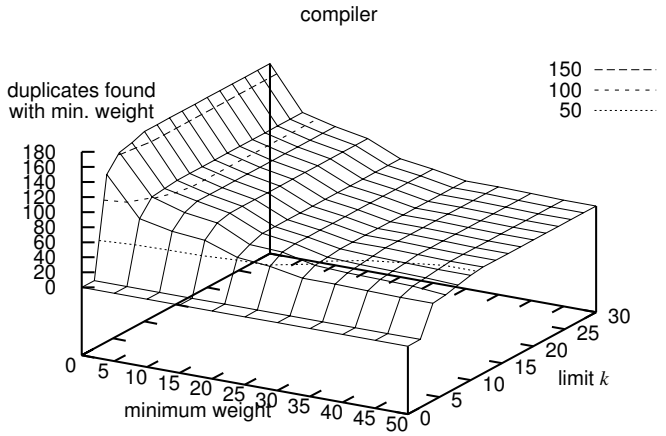


Figure 12.7: Results for `compiler`

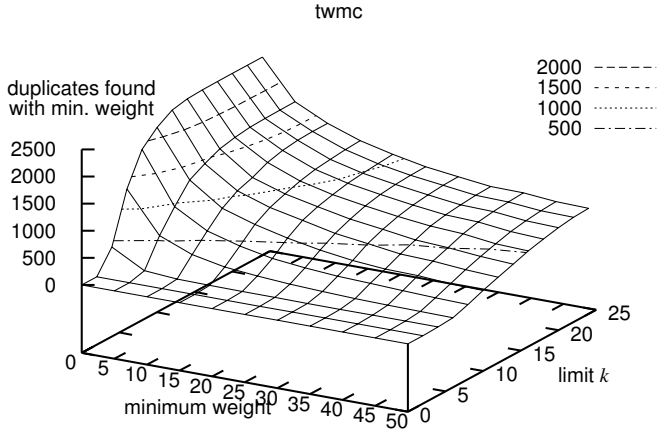


Figure 12.8: Results for `twmc`

	Tool	Technique	
B. Baker	Dup	lexical	[Bak95]
I. Baxter	CloneDR	structural	[BYM+98]
T. Kamiya	CCFinder	lexical	[KKI02]
J. Krinke	Duplix	dependence	
E. Merlo	CLAN	metric	[MLM96]
M. Rieger	Duploc	lexical	[DRD99]

Table 12.4: Participants

than 1000 with a weight larger than 20. These extremely high numbers stem from massive code duplication in that particular software: It contains a high amount of files, which just have been copied and changed a bit for slightly different purposes.

12.4 Comparison with other Tools

The presented approach has participated in a comparison study done by Bellon [Bel02]. The participants of this study are shown in table 12.4.

The reported duplicates (or *clones*) where categorized in *identical*, *parameterized* and *modified* clones. Each category has been compared individually for the six tools. The study showed:

- This technique (despite the mentioned recall problems) was *best* in detecting *modified* clones of which many were not discovered by any other technique. The results for the test case "cook" are shown in figure 12.9 on the following page: Our tool had a recall of 56% for modified clones—well above the recall for any other tool in that category.
- Our technique is slower than all others (which was expected because of the high complexity).
- This technique had worse precision than the others for identical and parameterized duplicates. This was also expected because of a very restricted definition of a clone: It had to be a continuous source code range. As this technique focuses on detection of similar code, it may report fragmented and distributed code. A projection onto a continuous source code range usually results in a large fragment that is not considered as a (precise) clone.
- This technique had worse recall than the others for identical and parameterized duplicates—which was *unexpected*. However, a closer examination revealed the following reasons:
  - To speed-up computation, a pair of clones was not allowed to be in the same procedure (the reference test data contains such clones).

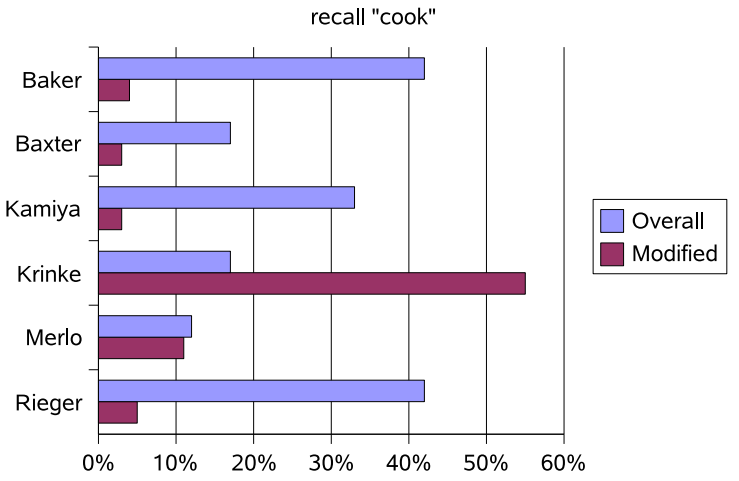


Figure 12.9: Results for test case `cook`

- Due to k-limiting, reported clones may be smaller than they actually are (the reference test data contains huge clones).
- The used implementation had a 'bug': entry nodes were not used as starting nodes. Therefore a lot of small cloned functions have not been identified.

This experience is shared with the other tools that are not based on lexical comparison: Our tool has a recall of 17%, Baxter's structure based tool also has 17% and Merlo's metric based tool has a recall of 12% only. The tools with higher recall numbers all use a similar approach based on lexical comparison.

Another interesting observation was that often similar code has not been cloned (or duplicated), because programs frequently contain repeating patterns of accesses to data structures.

12.5 Related Work

A very similar approach is [KH01], based on (traditional) program dependence graphs. Starting from *every* pair of matching nodes, they construct isomorphic subgraphs for ideal clones that can be replaced by function calls automatically. Different from the presented approach, their subgraphs are only subtrees that are not maximal, as they visit every node only once during subgraph construction. They cannot analyze big programs due to limitations of the underlying PDG generating infrastructure, too.

Another structure-comparing work is [BYM+98], where a program under observation is transformed to an AST first. For every subtree in the AST a hash value is computed and identical subtrees are identified via identical hash values. To also detect similar (not identical) subtrees, the subtrees have to be compared pairwise. The authors suggest improvements as future work that are similar to our approach.

An approach that obeys but that does not compare syntactical structure is [MLM96], where metrics are calculated from names, layout, expression and (simple) control flow of functions. Two functions are considered to be clones if their metrics are similar. This work can only identify similar functions but not similar pieces of code. A language-independent approach is [DRD99], which looks for specific patterns in a comparison of every line to each other. Other text-based approaches are [Bak95, KKI02]. These approaches can be used to analyze very large programs as they are not relying on pairwise comparison.

An application in the same setting is the detection of *plagiarism*: Given two programs, one has to detect whether one program is duplicated partly or completely in the other. Most plagiarism detecting systems are comparing the lexical structure of the programs [PMP00, Wis92]. Other systems are based on metrics again; however, studies show that metric-based systems are only partly successful because of the trade-off between recall and precision, both for detection of plagiarism [VW96] and detection of similar code [Kon97].

The opposite problem to identifying similar or identical parts of programs is identifying the differences between programs. [HPR89, HR92] is an approach to identify program differences based on program dependence graphs. However, that approach relies on the existence of a mapping ϕ that maps every node of one program to a node of the other if the representing program components are the same in both programs. The authors suggest a special program editor that keeps such a mapping. Instead, it can be found by the presented approach.

The matching of similar (attributed) graphs is used in other areas like computer vision [KO90] and graph visualization [Bac00], too.

The presented definition of similar paths and graphs have equivalences in process algebra [vG01] if the graphs are seen as process graphs where the transitions (s, a, t) are defined by the mapping Δ: $(s, \Delta(s, t), t)$. However, in its present state the presented algorithm does not even preserve *trace equivalence*. Future algorithms could be made more precise by only identifying *trace equivalent* nodes (or requiring even stronger equivalence classes).

Chapter 13

Path Conditions

Slicing can answer the question *which statements have an influence on statement X?*, chopping can answer the question *how does statement Y influence statement X?*, but neither slicing nor chopping can answer the question *why statement Y influences statement X*.

Example 13.1: Consider Figure 13.1 on the next page for an example: First, the question *which statements influence the output of variable y in line 8* is simply answered by computing a slice for this criterion. Figure 13.2 on the following page shows the result: Just one statement or node does not influence the criterion. Now, why is statement 1 included in the slice and how does statement 1 influence statement 8? This can be answered by a chop between statement 1 and statement 8. However, the result is a chop identical to the previous slice—which does not help at all.

Here, the computation of *path conditions* can assist. Path conditions give necessary conditions under which a transitive dependence between the source and target (criterion) node exists. These conditions give the answer to "why is this statement in the slice?" which Griswold [Gri01] categorized as a hard question.

This chapter will first introduce the basic concepts of path conditions and how they can be extended for array and pointer usage. The third section will show how path conditions can be computed for programs with procedures, followed by a section where path conditions are even adapted for programs with concurrently executing threads.

13.1 Simple Path Conditions

A simple approach to compute path conditions between two nodes x and y in a program dependence graph consists of the following steps:

1. Compute all paths p_i from x to y in the program dependence graph.

```
1    if (i > 0)
2      a = x;
3      j = b;
4    c = z;
5    if (j < 5)
6      if (i < 8)
7        y = a;
8    print(y);
```

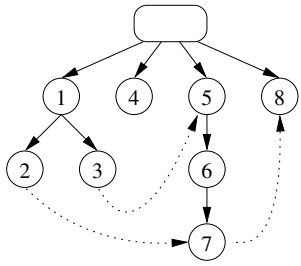


Figure 13.1: Simple fragment with program dependence graph

```
1    if (i > 0)
2      a = x;
3      j = b;
4
5    if (j < 5)
6      if (i < 8)
7        y = a;
8    print(y);
```

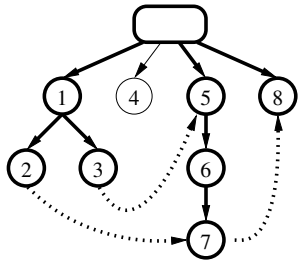


Figure 13.2: Slice of the fragment

2. For every node n, that is part of a path p_i, compute the *execution condition* $E(n)$.
3. Combine the execution conditions to compute the path condition $PC(x, y)$.

These three steps will be discussed next.

13.1.1 Execution Conditions

The execution condition $E(n)$ gives the conditions under which a node n may be executed. This can simply be computed by following the incoming control dependence edges and collecting the predicates of the ancestor nodes until the root (START) node is reached. In the example of Figure 13.1, the execution condition for statement 7 is $E(7) = (j < 5) \wedge (i < 8)$. More generally, an execution condition for a node n is computed by

$$E(n) = \bigwedge_{n \overset{cd}{\dashleftarrow} m \mid START \overset{cd}{\dashleftarrow}^\star n \overset{cd}{\dashleftarrow} m \overset{cd}{\dashleftarrow}^\star n} \gamma(n \overset{cd}{\dashleftarrow} m)$$

$$\gamma(n \overset{cd}{\dashleftarrow} m) = \begin{cases} \mu_E(n) & \text{if } \nu(n \overset{cd}{\dashleftarrow} m) = \mathsf{true} \\ \neg\mu_E(n) & \text{if } \nu(n \overset{cd}{\dashleftarrow} m) = \mathsf{false} \\ \mu_E(n) = \nu(n \overset{cd}{\dashleftarrow} m) & \text{otherwise} \end{cases}$$

$$
\begin{array}{lcl}
E(1) & = & \mathsf{true} \\
E(2) & = & \mathtt{i} > 0 \\
E(3) & = & \mathtt{i} > 0 \\
E(5) & = & \mathsf{true} \\
E(6) & = & \mathtt{j} < 5 \\
E(7) & = & (\mathtt{j} < 5) \wedge (\mathtt{i} < 8) \\
E(8) & = & \mathsf{true}
\end{array}
$$

Figure 13.3: Execution conditions

where $\mu_E(n)$ returns the (predicate) expression of a node n and $\nu(e)$ returns the label of edge e. Control dependence edges leaving predicates of if- and while-statements are labeled with either true or false and control dependence edges leaving expressions of switch statements are labeled with the constant of the target case.

Example 13.2: Figure 13.3 shows the execution conditions for each of the example's statements.

In the presence of unstructured control flow the control dependence subgraph may not be a tree and there may be more than a single path from the root to the node of interest. Under such circumstances, the execution conditions compute as follows:

$$
\begin{array}{lcl}
E(p) & = & \displaystyle\bigwedge_{n \xrightarrow{cd} m \mid p = \mathrm{START} \xrightarrow{cd}{}^{\star} n \xrightarrow{cd} m \xrightarrow{cd}{}^{\star} n} \gamma(n \xrightarrow{cd} m) \\
E(n) & = & \displaystyle\bigvee_{p = \mathrm{START} \xrightarrow{cd}{}^{\star} n} E(p)
\end{array}
$$

In presence of unstructured control flow, the control dependence subgraph may even be cyclic which causes the set of possible paths to be infinite. Snelting [Sne96] proved that cycles in execution conditions can be ignored and thus execution conditions are only computed over the finite set of cycle-free paths.

Other provisions to handle unstructured control flow are not necessary. Even approaches like the augmented control flow and program dependence graph from Section 3.4 are not needed because jump statements are predicateless.

13.1.2 Combining Execution Conditions

The execution conditions are used to form the path conditions. For a path p in a program dependence graph, the execution conditions are conjunctively combined to result in the conditions under which this path may be taken during execution:

```
1     x = a
2     while (x<7) {
3       x = y + x
4       if (x == 8)
5         p(x)
6     }
```

$$\begin{aligned} E(1) &= \mathsf{true} \\ E(2) &= \mathsf{true} \\ E(3) &= x < 7 \\ E(4) &= x < 7 \\ E(5) &= (x < 7) \wedge (x = 8) \end{aligned}$$

Figure 13.4: Example with multiple definitions for variable x

$$PC(p) = \bigwedge_{n|p=\langle \ldots,n,\ldots\rangle} E(n) \tag{13.1}$$

Usually more than one path exists between two nodes x and y and the path conditions for the single paths are combined disjunctively to form the path condition $PC(x, y)$:

$$PC(x, y) = \bigvee_{p=x \rightharpoonup^{\star} y} PC(p) \tag{13.2}$$

A program dependence graph is typically cyclic which may result in an infinite number of paths between two nodes. Again, cycles in paths can be ignored [Sne96] and only cycle-free paths are used.

Example 13.3: The example of figure 13.1 on page 194 contains two paths from statement 1 to 8: $p_1 = \langle 1, 2, 7, 8\rangle$ and $p_2 = \langle 1, 3, 5, 6, 7, 8\rangle$. Their path conditions are:

$$\begin{aligned} PC(p_1) &= \mathsf{true} \wedge (i > 0) \wedge (j < 5) \wedge (i < 8) \wedge \mathsf{true} \\ &= (i > 0) \wedge (j < 5) \wedge (i < 8) \\ PC(p_2) &= \mathsf{true} \wedge (i > 0) \wedge \mathsf{true} \wedge (j < 5) \wedge (j < 5) \wedge (i < 8) \wedge \mathsf{true} \\ &= (i > 0) \wedge (j < 5) \wedge (i < 8) \end{aligned}$$

Both path conditions can be combined disjunctively and simplified: The path condition from statement 1 to 8 is then $PC(1, 8) = (i > 0) \wedge (j < 5) \wedge (i < 8)$.

13.1.3 SSA Form

In presence of multiple definitions of a single variable, the presented approach does not succeed:

Example 13.4: Consider Figure 13.4, where the execution condition of each statement is shown to the right. The execution condition for statement 5 is $E(7) = (x < 7) \wedge (x = 8)$, which cannot be fulfilled.

The problem here is that the two usages of variable x are not the same instance, because x gets redefined at statement 3. A simple solution is to use SSA form [CFR+91], so that every variable has a single definition.

```
1     x1 = a
2     while (x2 = φ(x1,x3),        E(1) = true
              x2 < 7) {            E(2) = true
3       x3 = y1 + x2               E(3) = x2 < 7
4       if (x3 == 8)               E(4) = x2 < 7
5         p(x3)                    E(5) = (x2 < 7) ∧ (x3 = 8)
6     }
```

Figure 13.5: Example in SSA form

Example 13.5: Figure 13.5 shows the example program in SSA form. Because the usages of variable x in lines 2 and 3 may use the definitions of x in line 1 or 3; a new assignment to x has been inserted. This new assignment may use either the definition from line 1 or 3, modeled as a ϕ-function. Also, all variable definitions and uses have been replaced with a new (numbered) variable, so that for every usage of a variable a single definition exists.

But there is a problem with SSA form, too: Because SSA form just captures static behavior, it does not distinguish different instances due to loops.

Example 13.6: Consider the following code:

```
1   int n, a, b, c, x, y, z;
2   a = y;
3   while (n > 0) {
4     x = ...;
5     if (x > 0)
6       b = a;
7     else
8       c = b;
9   }
10  z = c;
```

This fragment is already in SSA form: Every variable has a single assignment. If the path condition $\mathrm{PC}(2, 10)$ is computed, the single path between statement 2 and 10 will be traversed:

$$\begin{aligned}
\mathrm{PC}(2,10) &= \mathrm{PC}(2 \rightharpoonup 6 \rightharpoonup 8 \rightharpoonup 10) \\
&= \mathsf{E}(2) \wedge \mathsf{E}(6) \wedge \mathsf{E}(8) \wedge \mathsf{E}(10) \\
&= \mathsf{true} \wedge (\mathtt{x} > 0) \wedge (\mathtt{n} > 0) \wedge \neg(\mathtt{x} > 0) \wedge (\mathtt{n} > 0) \wedge \mathsf{true} \\
&= \mathsf{false}
\end{aligned}$$

Of course, this result is incorrect: The execution conditions $\mathsf{E}(6)$ and $\mathsf{E}(8)$ are using different instances of variable x, dependent on the loop iteration.

The key to a correct solution is *loop-carried* data dependence: the data dependence between statement 6 and 8 is a loop-carried data dependence, because it only exists if the back edge of the loop is executed at least once. The

path conditions are decomposed at loop-carried data dependence edges: Let the analyzed path be $p = p_1 p_2 \dots p_n$, where all p_i are loop-carried dependence-free subpaths and all p_i are connected to p_{i+1} via a loop-carried dependence. The correct equation is now:

$$\mathrm{PC}(p) \quad = \quad \bigwedge_{1 \leqslant i \leqslant n} \mathrm{PC}(p_i, i)$$

where $\mathrm{PC}(p_x, x)$ is a path condition $\mathrm{PC}(p_x)$ for path p_x and every occurrence of a variable v has been replaced by v_x, a new instance of v.

Example 13.7: The computation of $\mathrm{PC}(2, 10)$ now results in:

$$\begin{aligned}
\mathrm{PC}(2,10) &= \mathrm{PC}(2 \rightharpoonup 6 \rightharpoonup 8 \rightharpoonup 10) \\
&= \mathrm{PC}(2 \rightharpoonup 6, 1) \wedge \mathrm{PC}(8 \rightharpoonup 10, 2) \\
&= \mathsf{true} \wedge (x_1 > 0) \wedge (n_1 > 0) \wedge \neg(x_2 > 0) \wedge (n_2 > 0) \wedge \mathsf{true} \\
&= (x_1 > 0) \wedge (n_1 > 0) \wedge (x_2 \leqslant 0) \wedge (n_2 > 0)
\end{aligned}$$

which can be interpreted as: *the loop has to be iterated at least twice, where in one iteration* $x > 0$ *holds and in a later iteration* $x \leqslant 0$ *holds.*

Such path conditions are not only correct conservative approximations, but they also contain more information. On the other hand, the execution condition may contain variables which must not be distinguished in different instances when a more complex embedding of loops and if-statements exists. A replacement of such different instances by a single one makes the path conditions smaller, but not more precise. How this is done is presented in [Rob].

From now on, SSA form is assumed—further examples will be free of multiple definitions of a single variable or loop-carried data dependence.

13.2 Complex Path Conditions

The presented path conditions can handle only intraprocedural programs limited to simple variables. The following will extend the path conditions stepwise to handle complex data structures like array and pointer usage, too.

13.2.1 Arrays

If array elements are distinguished, additional constraints for index expressions will be generated for data dependences concerning array elements.

Example 13.8: This fragment is an example of an array usage:

```
1    a[i + 3] = x
2    if (i > 10)
3      y = a[2*j - 42]
```

A path condition $PC(1,3)$ exists only if a path from 1 to 3 exists. This requires a data dependence between 1 and 3 which only exists if $i + 3 = 2j - 42$.

In general, any data dependence edge $n_1 \rightharpoonup n_2$ with $\mu_E(n_1) = a[E_1]$ and $\mu_E(n_2) = a[E_2]$ generates a constraint $E_1 = E_2$. For a path in the dependence graph, all such constraints along its edges are added conjunctively to the path condition. Thus, the general equation 13.1 to compute a path condition for a path becomes

$$PC(p) = \bigwedge_{n|p=\langle\ldots,n,\ldots\rangle} E(n) \wedge \bigwedge_{n\rightharpoonup m|p=\langle\ldots,n\rightharpoonup m,\ldots\rangle} \delta(n \rightharpoonup m) \tag{13.3}$$

where

$$\delta(n_1 \rightharpoonup n_2) = \begin{cases} E_1 = E_2 & \text{if } (\mu_E(n_1) = a[E_1]) \wedge (\mu_E(n_2) = a[E_2]) \\ \mathsf{true} & \text{otherwise} \end{cases} \tag{13.4}$$

The resulting path conditions may contain complex conditions for index values, and it is well known that arbitrary constraints over integers cannot be solved. But many solvers can deal with constant or linear index expressions or even Presburger arithmetic [PW98].

When several definitions of an array element reach the same program point, the situation becomes even more complex as the dependence edges themselves must be modified in order to take care of possible aliases.

Example 13.9: Consider

```
1    a[i] = x;
2    a[j] = y;
3    z = a[k];
```

The standard approach (shown in section 4.2.2) is to assume assignments to array elements to be *non-killing definitions* (i.e. no previous assignment is killed). This generates data dependence edges $1 \rightharpoonup 3$ and $2 \rightharpoonup 3$. However, this is problematic for path conditions, as knowledge of the execution order of statements 1 and 2 is not available from the control and data dependence alone. The knowledge that statement 2 may kill the definition at statement 1 is lost. With the additional constraint that `i` equals `j`, the path condition is too weak: $(i = j) \wedge PC(1,3) = (i = j) \wedge (i = k)$. However, this should evaluate to false.

Therefore, assignments to array elements are assumed to be *killing modifications*: An assignment to an array element uses the (complete) array before the specified element is defined. This approach does not change slices or chops for any criterion node x except where x is an assignment to an array element[1]. Now, the example has data dependence edges $1 \rightharpoonup 2$ and $2 \rightharpoonup 3$, where edge $1 \rightharpoonup 2$ is going from a definition to a definition and $2 \rightharpoonup 3$ is going from a definition to a use. The δ-constraints for edges going from a definition to a

[1] It does change distant slicing results from chapter 10.

definition are negations of the above equations, as the array element defined at the source is only used at the target if the array indices are *not* the same.

For the killing variant, the δ-constraints of equation 13.3 and 13.4 are more complex and only valid for a *specific* path under examination, because all data dependence edges between definitions of array elements must be followed backwards:

$$PC(p) = \bigwedge_{n|p=\langle\ldots,n,\ldots\rangle} E(n) \wedge \bigwedge_{n\rightharpoonup m|p=\langle\ldots,n\rightharpoonup m,\ldots\rangle} \delta(n \rightharpoonup m, p) \tag{13.5}$$

Let $\text{ddchain}(e, p)$ be a function that maps a (cycle-free) path $p = \langle e_1, \ldots, e_n\rangle$ to a maximal subpath $p' = \langle e_i, \ldots, e_j\rangle$, such that $e_j = e$ and all edges $e_i \ldots e_{j-1}$ are data dependence edges between two array element definitions. The edge $e = e_j$ must be a data dependence edge from a definition to a usage of an array element. Path p' is not maximal if edge e_{i-1} exists and that edge is a data dependence edge between two array element definitions. Also, let the function $A(n)$ return the expression E of a node n, where $\mu_E(n) = a[E]$.

$$\begin{aligned}\delta(n \rightharpoonup m, p) \quad = \quad & \text{let } p' = \text{ddchain}(n \rightharpoonup m, p) \\ & \text{with } p' = \langle n_1 \rightharpoonup n_2, n_2 \rightharpoonup n_3, \ldots, n_k \rightharpoonup m\rangle \wedge n_k = n \\ & \text{in } (A(n_1) = A(m)) \wedge \left(\textstyle\bigwedge_{1<i\leqslant k} A(n_i) \neq A(m)\right)\end{aligned} \tag{13.6}$$

If path p does not contain edges between two definitions of array elements, equations 13.5 and 13.6 reduce to equations 13.3 and 13.4.

Example 13.10: The example path condition $PC(1,3)$ is now computed with the new equations:

$$\begin{aligned} PC(1,3) &= PC(\langle 1 \rightharpoonup 2, 2 \rightharpoonup 3\rangle) \\ &= E(1) \wedge E(2) \wedge E(3) \\ &\quad \wedge \delta(1 \rightharpoonup 2, \langle 1 \rightharpoonup 2, 2 \rightharpoonup 3\rangle) \\ &\quad \wedge \delta(2 \rightharpoonup 3, \langle 1 \rightharpoonup 2, 2 \rightharpoonup 3\rangle) \\ &= \text{true} \wedge \text{true} \wedge \text{true} \\ &\quad \wedge \text{true} \\ &\quad \wedge (i = k) \wedge (j \neq k) \\ &= (i = k) \wedge (j \neq k) \end{aligned}$$

This result is stronger than the first path condition which was $PC(1,3) = (i = k)$ and makes the possible killing of i clear.

13.2.2 Pointers

Even more complex than array usage is the presence of pointers in the analyzed program.

Example 13.11: Consider the following fragment:

```
1    *q = x;
2    *p = y;
3    if (a > 7)
4      p = q;
5    z = *p;
```

Besides the data dependence $4 \rightharpoonup 5$, also the data dependences $1 \rightharpoonup 5$ and $2 \rightharpoonup 5$ exist due to pointer dereferencing. The path condition $PC(1,5)$ is simply true.

Again, the path conditions can be made stronger with additional δ-constraints that represent the additional requirements of aliasing. Equation 13.4 is extended for data dependence due to pointer usage:

$$\delta(n_1 \rightharpoonup n_2) = \begin{cases} E_1 = E_2 & \text{if } \mu_E(n_1) = *E_1 \wedge \mu_E(n_2) = *E_2 \\ \&E_1 = E_2 & \text{if } \mu_E(n_1) = E_1 \wedge \mu_E(n_2) = *E_2 \\ E_1 = \&E_2 & \text{if } \mu_E(n_1) = *E_1 \wedge \mu_E(n_2) = E_2 \\ E_1 = E_2 & \text{if } \mu_E(n_1) = a[E_1] \wedge \mu_E(n_2) = a[E_2] \\ \text{true} & \text{otherwise} \end{cases}$$

These equations handle only a subset of possible pointer expressions. More complex pointer expressions use the fall-through of true as a conservative approximation: The path conditions are correct but not as precise as possible. The path condition $PC(1,5)$ of the small example now computes to $PC(1,5) = (q = p)$. Of course, the pointer extensions can also be applied together with the more complex definition of δ in equation 13.6.

13.3 Increasing the Precision

One reason of the optimizations in chapter 11 was to increase the precision of path conditions.

Example 13.12: Consider the following fragment:

```
1    if(x > 0)
2      b = a;
3    y = x;
4    if (y < 0)
5      c = b;
```

Here, $PC(2,5)$ is computed as $PC(2,5) = (x > 0) \wedge (y < 0)$, whereas $PC(2,5) =$ false would be more precise (and still correct).

If copy propagation is applied first, both variables x and y are unified and the path condition will now be computed as $PC(2,5) = (x > 0) \wedge (y < 0) =$ false. Besides the optimizations from section 11.2, other optimizations techniques like global value numbering and value flow analysis [ASU85] could increase the accuracy of path conditions.

13.4 Interprocedural Path Conditions

So far, single procedure programs have been assumed. To generate path conditions for programs with procedures, the problem of unrealizable paths must be revisited. First, interprocedural path conditions are categorized:

Truncated same-level path conditions are path conditions $PC_{TS}(x, y)$ where x and y are nodes of the same procedure and the condition is built only from the procedure itself. Called procedures are obeyed without generating new conditions.

Non-truncated same-level path conditions are path conditions $PC_{NS}(x, y)$ where x and y are nodes of the same procedure and the condition is built from the procedure itself together with the called procedures.

Truncated non-same-level path conditions are path conditions $PC_{TN}(x, y)$ where x and y are nodes of not necessarily the same procedure and the condition is built only from the procedures of x and y together with *necessary* calling procedures.

Non-truncated non-same-level path conditions are path conditions $PC_{NN}(x, y)$ where x and y are nodes of not necessarily the same procedure and the condition is built from all involved procedures.

The concept of non-same-level path conditions will be made clear later in this section; first, the same-level variants are explained.

13.4.1 Truncated Same-Level Path Conditions

The computation of truncated same-level path conditions just needs a restriction on allowed paths for equation 13.2 on page 196: A path p in the system dependence graph has to be a *truncated same-level* path, i.e. it is not allowed to contain parameter or call edges.

$$PC_{TS}(x, y) = \bigvee_{p = x \rightharpoonup^{\star} y} PC_{TS}(p) \text{ and all } p \text{ are truncated same-level paths}$$

$$PC_{TS}(p) = \bigwedge_{n \mid p = \langle \ldots, n, \ldots \rangle} E(n) \wedge \bigwedge_{n \rightharpoonup m \mid p = \langle \ldots, n \rightharpoonup m, \ldots \rangle} \delta(n \rightharpoonup m)$$

Now, the only different case to the intraprocedural path condition is the presence of summary edges. Because effects of called procedures have to be ignored, no extension to the δ-constraint is necessary and summary edges just use the fall-through true for the δ-constraint.

13.4.2 Non-Truncated Same-Level Path Conditions

Non-truncated same-level path conditions are like their truncated counterparts, except that effects of called procedures should be used to make the path conditions stronger. This is achieved by extending the δ-constraints for summary edges. A summary edge $x \rightharpoonup y$ between actual parameter nodes shows the existence of a transitive dependence between the corresponding formal parameter nodes. The conditions under which this dependence exists can be computed by path condition $PC(x', y')$ between the formal parameter nodes x', y' corresponding to x, y. This path condition must be bound to the call site by binding together the variables of the actual and formal parameter nodes. But this is not enough, because the instance problem of section 13.1.3 exists here, too. Therefore, all variables in the path condition through the called procedure must be replaced by a new instance with $PC(x', y', i)$, where i has not been used before. Let v be the variable of the actual-in node x ($\mu_E(x) = v$), w the variable of the actual-out node ($\mu_E(y) = w$) and let v' and w' be the variables of the corresponding formal parameter nodes ($v' = \mu_E(x')$, $w' = \mu_E(y')$). Then

$$
\begin{aligned}
PC_{NS}(x, y) &= \bigvee_{p = x \rightharpoonup^* y} PC_{NS}(p) \text{ and all } p \text{ are truncated same-level paths} \\
PC_{NS}(p) &= \bigwedge_{n | p = \langle \ldots, n, \ldots \rangle} E(n) \wedge \bigwedge_{n \rightharpoonup m | p = \langle \ldots, n \rightharpoonup m, \ldots \rangle} \delta_{NS}(n \rightharpoonup m) \\
\delta_{NS}(x \rightharpoonup y) &= \begin{cases} (v = v'_i) \wedge (w = w'_i) \wedge PC_{NS}(x', y', i) \\ \qquad \text{if } x \rightharpoonup y \text{ is a summary edge} \\ \delta(x \rightharpoonup y) \quad \text{otherwise} \end{cases}
\end{aligned}
$$

In presence of recursion the generated path conditions can be recursive themselves: The previous equations are similar to inlining called procedures. To prevent unlimited unfolding, only k-truncated same-level path conditions are computed, where unfolding stops after k levels:

$$
\begin{aligned}
PC^k_{NS}(x, y) &= \bigvee_{p = x \rightharpoonup^* y} PC^k_{NS}(p) \text{ and all } p \text{ are truncated same-level paths} \\
PC^k_{NS}(p) &= \bigwedge_{n | p = \langle \ldots, n, \ldots \rangle} E(n) \wedge \bigwedge_{n \rightharpoonup m | p = \langle \ldots, n \rightharpoonup m, \ldots \rangle} \delta^k_{NS}(n \rightharpoonup m) \\
\delta^k_{NS}(x \rightharpoonup y) &= \begin{cases} (v = v'_i) \wedge (w = w'_i) \wedge PC^{k-1}_{NS}(x', y', i) \\ \qquad \text{if } k > 1 \wedge x \rightharpoonup y \text{ is a summary edge} \\ \delta(x \rightharpoonup y) \quad \text{otherwise} \end{cases}
\end{aligned}
$$

If unlimited unfolding ($k = \infty$) is allowed, $PC_{NS}(x, y) = PC^\infty_{NS}(x, y)$ holds.

13.4.3 Truncated Non-Same-Level Path Conditions

The restriction that both nodes x and y for a path condition must be in the same procedure is relaxed and x and y may now be in different procedures. However, the path condition is still truncated, i.e. summary edges are not expanded.

Therefore, the paths between x and y have to be *truncated non-same-level* paths. A truncated non-same-level path P can be split in two parts p_1 and p_2 with $p = p_1 p_2$, such that p_1 does not contain parameter-in and call edges and p_2 does not contain parameter-out edges (see section 10.2.7 for an explanation). This restriction will exclude automatically all unrealizable paths, because after following a parameter-in edge it is impossible to follow a parameter-out edge. The equations are similar to those of the truncated same-level path conditions:

$$\begin{aligned} PC_{TN}(x,y) &= \bigvee_{p=x\rightharpoonup^* y} PC_{TN}(p) \text{ and all } p \text{ are truncated non-same-level paths} \\ PC_{TN}(p) &= \bigwedge_{n|p=\langle\ldots,n,\ldots\rangle} E(n) \wedge \bigwedge_{n\rightharpoonup m|p=\langle\ldots,n\rightharpoonup m,\ldots\rangle} \delta(n \rightharpoonup m) \end{aligned}$$

13.4.4 Non-Truncated Non-Same-Level Path Conditions

The expansion of truncated non-same-level path conditions to non-truncated ones is not different to the expansion of same-level path conditions:

$$\begin{aligned} PC^k_{NN}(x,y) &= \bigvee_{p=x\rightharpoonup^* y} PC^k_{NN}(p) \text{ and all } p \text{ are truncated non-same-level paths} \\ PC^k_{NN}(p) &= \bigwedge_{n|p=\langle\ldots,n,\ldots\rangle} E(n) \wedge \bigwedge_{n\rightharpoonup m|p=\langle\ldots,n\rightharpoonup m,\ldots\rangle} \delta^k_{NN}(n \rightharpoonup m) \\ \delta^k_{NN}(x \rightharpoonup y) &= \begin{cases} (v = v'_i) \wedge (w = w'_i) \wedge PC^{k-1}_{NN}(x', y', i) \\ \qquad \text{if } k > 1 \wedge x \rightharpoonup y \text{ is a summary edge} \\ \delta(x \rightharpoonup y) \quad \text{otherwise} \end{cases} \end{aligned}$$

Again, $PC_{NN}(x,y) = PC^{\infty}_{NN}(x,y)$ holds.

13.4.5 Interprocedural Execution Conditions

The execution conditions of the presented path conditions are computed just intraprocedural, because only the control dependence subgraph of the enclosing procedure is traversed. This is almost sufficient for interprocedural path conditions as the execution of a called procedure is controlled by the execution condition for the call site.

Example 13.13: Figure 13.6 on the next page contains a program with two procedures p and q. The path condition for a dependence between statement 7 and 10 is computed as shown in figure 13.7 on the facing page. The resulting path condition $(x < 8) \wedge (x > 3)$ can be made stronger by including the calling context of the criterion statements: Statement 7 and statement 10 are only executed if procedure q is called.

```
01    proc p():
02      if (x > 3)
03        z = y
04    end
06    proc q():
07      y = 3
08      if (x < 8)
09        p()
10      print(z)
11    end
12
13    read(x)
14    if (x < 0)
15      q()
```

Figure 13.6: Example for an interprocedural path condition

$$
\begin{aligned}
\mathrm{PC}^2_{\mathrm{NN}}(7,10) &= \mathrm{PC}^2_{\mathrm{NN}}(7 \rightharpoonup 9_{y_{\mathrm{in}}} \rightharpoonup 9_{z_{\mathrm{out}}} \rightharpoonup 10) \\
&= \mathrm{E}(7) \wedge \mathrm{E}(9_{y_{\mathrm{in}}}) \wedge \mathrm{E}(9_{z_{\mathrm{out}}}) \wedge \mathrm{E}(10) \\
&\quad \wedge \delta^2_{\mathrm{NN}}(7 \rightharpoonup 9_{y_{\mathrm{in}}}) \\
&\quad \wedge \delta^2_{\mathrm{NN}}(9_{y_{\mathrm{in}}} \rightharpoonup 9_{z_{\mathrm{out}}}) \\
&\quad \wedge \delta^2_{\mathrm{NN}}(9_{z_{\mathrm{out}}} \rightharpoonup 10) \\
&= \mathsf{true} \wedge (x < 8) \wedge (x < 8) \wedge \mathsf{true} \\
&\quad \wedge \mathsf{true} \\
&\quad \wedge (x = x_1) \wedge (z = z_1) \wedge \mathrm{PC}^1_{\mathrm{NN}}(1_{x_{\mathrm{in}}}, 4_{z_{\mathrm{out}}}, 1) \\
&\quad \wedge \mathsf{true} \\
&= (x < 8) \wedge (x = x_1) \wedge (z = z_1) \wedge \mathrm{PC}^1_{\mathrm{NN}}(1_{x_{\mathrm{in}}}, 4_{z_{\mathrm{out}}}, 1)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{PC}^1_{\mathrm{NN}}(1_{x_{\mathrm{in}}}, 4_{z_{\mathrm{out}}}) &= \mathrm{PC}^1_{\mathrm{NN}}(1_{y_{\mathrm{in}}} \rightharpoonup 3 \rightharpoonup 4_{z_{\mathrm{out}}}) \\
&= \mathrm{E}(1_{y_{\mathrm{in}}}) \wedge \mathrm{E}(3) \wedge \mathrm{E}(4_{z_{\mathrm{out}}}) \\
&\quad \wedge \delta^1_{\mathrm{NN}}(1_{y_{\mathrm{in}}} \rightharpoonup 3) \wedge \delta^1_{\mathrm{NN}}(3 \rightharpoonup 4_{z_{\mathrm{out}}}) \\
&= \mathsf{true} \wedge (x > 3) \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \\
&= (x > 3)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{PC}^2_{\mathrm{NN}}(7,10) &= (x < 8) \wedge (x = x_1) \wedge (z = z_1) \wedge \mathrm{PC}^1_{\mathrm{NN}}(1_{x_{\mathrm{in}}}, 4_{z_{\mathrm{out}}}, 1) \\
&= (x < 8) \wedge (x = x_1) \wedge (z = z_1) \wedge (x_1 > 3)
\end{aligned}
$$

Figure 13.7: Computation of $\mathrm{PC}^2_{\mathrm{NN}}(7,10)$ for example 13.6

The calling context can induce additional execution conditions which can simply be computed by not only traversing control dependence but also call edges:

$$\begin{aligned} E_{IP}(p) &= \bigwedge_{n \overset{cd,cl}{\rightharpoonup} m \mid p=START \overset{cd}{\rightharpoonup}{}^{\star} n \overset{cd}{\rightharpoonup} m \overset{cd,cl}{\rightharpoonup}{}^{\star} n} \gamma(n \overset{cd}{\rightharpoonup} m) \\ E_{IP}(n) &= \bigvee_{p=START \overset{cd,cl}{\rightharpoonup}{}^{\star} n} E_{IP}(p) \end{aligned}$$

The traversal of call edges includes the execution conditions at the call sites, which control the execution of the called procedures. The equations for interprocedural path conditions are extended to:

$$PC_{IP}(x,y) = E_{IP}(x) \wedge E_{IP}(y) \wedge \bigvee_{p=x \rightharpoonup^{\star} y} PC_{IP}(p)$$

In this equation, PC_{IP} can be substituted by any of the four variants PC_{TS}, PC^k_{NS}, PC_{TN} or PC^k_{NN} to form the complete version.

Example 13.14: With this modified equations, the resulting example path condition is:

$$\begin{aligned} PC^2_{NN}(7,10) &= E_{IP}(7) \wedge E_{IP}(10) \wedge PC^2_{NN}(7 \rightharpoonup 9_{y_{in}} \rightharpoonup 9_{z_{out}} \rightharpoonup 10, 1) \\ &= (x < 0) \wedge (x < 0) \wedge PC^2_{NN}(7 \rightharpoonup 9_{y_{in}} \rightharpoonup 9_{z_{out}} \rightharpoonup 10, 1) \\ &= (x < 0) \wedge (x < 8) \wedge (x = x_1) \wedge (z = z_1) \wedge (x_1 > 3) \\ &= \mathsf{false} \end{aligned}$$

The outcome is as strong as it can become: In the example $PC(7,10) = \mathsf{false}$ holds—a dependence between 7 and 10 is impossible.

13.5 Multi-Threaded Programs

As described so far, path conditions can handle only sequential programs. In this section, the approach of chapter 5 is used to extend the path conditions for intraprocedural programs with concurrently executing threads. As yet, threaded interprocedural programs can only be handled by inlining called procedures at call sites.

The most simple approach to compute path conditions in the tPDG of chapter 5 is to replace all interference dependence edges with normal data dependence edges and compute the path conditions as usual. The resulting path conditions are always correct because the replacement of interference dependence is a conservative approximation. However, the resulting path conditions are imprecise as they allow impossible paths.

Example 13.15: Consider the following program fragment: It is impossible that statement 2 is executed *after* statement 3. However, due to the interference dependences $3 \rightharpoonup 6$ and $6 \rightharpoonup 2$, there exists a path from 3 to 2 and the path condition computes to $PC(3,2) = x > 0$.

```
thread 1:              thread 2:
  1    a = b;            5    if (x>0)
  2    c = d;            6      d = e;
  3    e = a;            7    if (y>0)
                         8      d = a;
```

Chapter 5 introduced the notion of a *threaded witness*, which is the witness of a possible program execution where its nodes execute in the same order. The same chapter showed how paths in the tPDG can be constructed so that they are always threaded witnesses. This can simply be used to refine the general equation 13.2 for path conditions:

$$PC(x, y) = \bigvee_{\substack{p = x \dashrightarrow^{\star} y \\ p \text{ is a threaded witness}}} PC(p)$$

The example path condition $PC(3, 2)$ is now simply false—no valid path from 3 to 2 being a threaded witness exists in the tPDG.

13.6 Related Work

Path conditions have been introduced by Snelting [Sne96] as a way to validate measurement software. The main problem to compute path conditions is scalability and efficiency. Therefore a divide-and-conquer strategy is applied: Paths and path conditions are decomposed before computation as shown in [Sne96, KS98] and [RS02, SRK03, Rob], which also contain case studies and evaluations.

Chapter 14

VALSOFT

Most of the techniques presented in the previous chapters have been implemented in the VALSOFT system. The system is a set of tools that are used to analyze C source code to help the engineer to understand and validate the code. This chapter will present the system in terms of its architecture and mention other slicing tools highlighting some of their features.

14.1 Overview

The overall architecture of the system is presented in figure 14.1 on the next page. Its parts will be discussed in the following. Table 14.1 on page 213 shows the sizes of the different parts[1] in terms of lines of code (LOC, counted by 'wc -l'), non-blank, non-comment lines (NLOC) and modules/classes (MOD) as counted by CCCC [Lit01].

14.1.1 C Frontend

The 'CParse' library is a frontend (scanner/parser etc.) for ANSI C. It reads (preprocessed) sources, constructs an attributed abstract syntax tree and a symbol table, and "links" all corresponding symbols of different sources together. This library has been implemented by LINEAS GmbH, Braunschweig. The only client of this library is the analyzer that constructs the program dependence graph for the set of a program's sources.

14.1.2 SDG Library

All tools of the VALSOFT system generate or use program (or system) dependence graphs. All functionality related to such graphs has been put into a 'SDG' library. It contains functionality

[1] The author's share of the implementation is around 45.000 LOC, 25.000 NLOC, 250 modules.

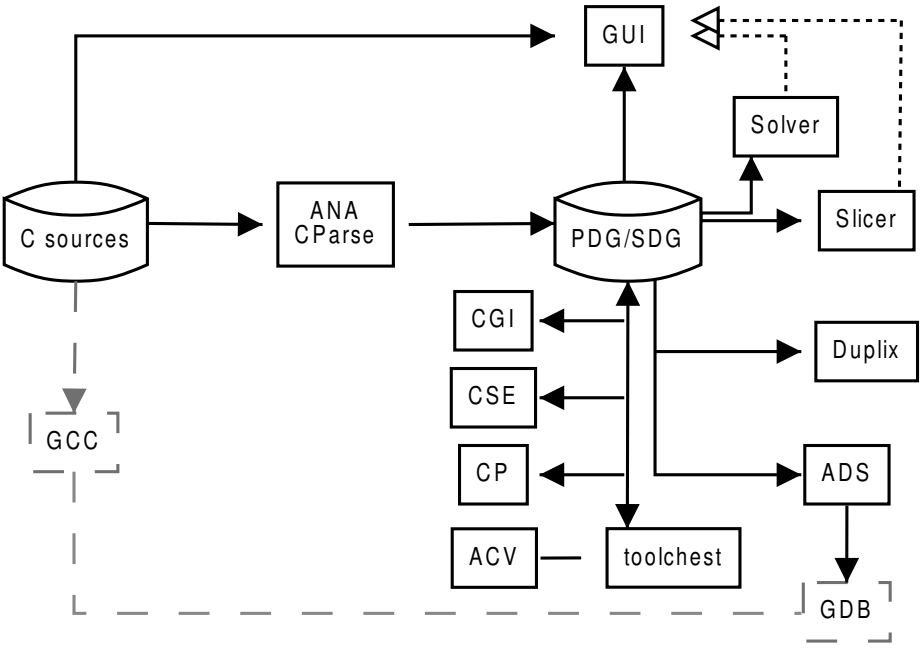


Figure 14.1: VALSOFT architecture

- to build, modify, save, restore and traverse general graphs, program and system dependence graphs,
- to generate summary edges in an interprocedural program dependence graph, and
- to compute forward and backward slices.

The only data that is exchanged between different tools are files containing program (or system) dependence graphs. Most tools use this library to read, write or work with program dependence graphs.

14.1.3 Analyzer

The 'ANA' tool is the analyzer and program dependence constructor. This tool consists of multiple passes:

1. A traversal of the AST (as built from 'CParse') creates a simple approximation of the call graph. At call sites, where function pointers are used, it is assumed that all procedures with a matching signature may be called. This is a crude but sufficient analysis for most C programs[2]. If a higher

[2]In only two of the used test-case programs this crude approach was imprecise.

precision is needed, the program dependence graph may be refined later on with the 'CG' tool.

2. Another traversal of the AST does a simple, flow-insensitive data flow analysis. It calculates the IMOD and IREF sets.

3. The alias (points-to) analysis is a further traversal of the AST. It is a flow-insensitive but context-sensitive alias analysis developed by Burke et al [BCCH95].[3]

4. The information of the first passes is now used to compute the later on needed GMOD and GREF sets in an interprocedural analysis (section 6.4).

5. The last traversal over the AST generates the control flow graph for every procedure. The control flow graphs will be expanded in later passes to fully-blown program dependence graphs. For well-structured procedures the control dependence edges are computed syntax-directed already during this pass.

6. For procedures with unstructured jumps, the control dependence edges are now computed traditionally according to section 2.3.1. For debugging or evaluation purposes, the conventionally computation can also be used for well-structured procedures, and the results can be compared to the syntax-directed approach.

7. As presented in section 6.4, a flow-sensitive computation of data dependence edges follows. For well-structured procedures, this can be done in two ways: A syntax-directed approach computes the reaching definitions and the data dependence edges through a one-pass traversal of the abstract syntax tree. The other approach is using a traditional iterative framework and iterates over the control flow graph. The results from both approaches can be used for debugging[4] and evaluation again. For procedures with unstructured jumps, only the iterative approach is used.

8. The last pass connects the program dependence graphs for all procedures to the interprocedural program dependence graph. This is achieved by inserting the call and parameter edges between the call sites and the called procedures. If wanted, summary edges can also be computed in this pass.

The created system dependence graph is persistently saved to disk and all other tools work with the saved graph. As presented in chapter 4, the program dependence graphs are fine-grained, such that client tools do not need other intermediate representations.

[3] A Steensgaard points-to analysis [Ste96] has also been implemented but not integrated.

[4] Through comparison of the results, some errors in implementation have been identified and removed. Because both approaches generate the same results in principle now, there is high confidence in both approaches.

14.1.4 The Slicer

The slicing and chopping algorithms of the SDG library have been made accessible by a slicer tool that can be used as a command-line tool or server to which other programs can connect. In server mode, the GUI connects to the server to compute the slices and chops, so that GUI and server may run on different machines. In command line mode, the slicer is able to compute slices or chops and visualize the results as source code or HTML files that can be displayed in any browser (as presented in section 9.2 and 10.1).

14.1.5 The Solver

Robschink [Rob] has developed the generation of path conditions as presented in chapter 13. The implementation is similar to the slicer: The solver can also be used as a command-line tool or server. The solver contains highly specialized techniques to decompose graphs and simplify path conditions.

14.1.6 The GUI

The graphical user interface mainly contains the visualization of dependence graphs as presented in section 9.1. It uses the external slicer and solver to compute slices, chops and path conditions.

The GUI is used for navigation in the PDGs and the source. It visualizes the program's call graph where the user may select any procedure node and the GUI visualizes the corresponding PDG. Every node is selectable and the corresponding parts of the source may be visualized in a textual manner. The user may select a set of nodes as a slicing criterion and let the GUI execute the slicer. The resulting set of nodes is visualized both in the graph based and textual presentations.

The implementation was done by Frank Ehrich [Ehr96] in cooperation with LINEAS GmbH.

14.1.7 The 'Tool Chest'

Various tools have been grouped together into a tool 'chest'. It mainly provides a test bed for experimentation with new approaches (e.g. the slicing and chopping algorithms from chapters 7 and 10). It also contains various tools for PDG manipulation, e.g. the three approaches to reduce the graphs' size from section 11.1. Some tools have moved from the tool chest into independent applications:

- The call graph improver ('CGI') eliminates redundant call edges in presence of function pointers as described in Section 11.2.1.
- The common subexpression eliminator ('CSE') identifies common subexpressions and replaces them by a shared subgraph (section 11.2.3).

	LOC	NLOC	MOD
CParse	17515	5912	36
SDG	11935	5992	103
ANA	16744	9285	105
Tool Chest	17792	11091	44
Slicer	1897	701	14
Solver	26314	13358	172
Duplix	1903	1313	18
ADS	1325	620	5
CP	4196	2706	20
CGI	1467	674	15
CSE	2188	1066	15
ACV	1094	588	24
GUI	59790	33702	175
Σ	170377	88419	613

Table 14.1: Code size of the VALSOFT components

- The constant propagator ('CP') does constant propagation over the SDGs and tries to simplify them (eliminating edges and nodes) as described in section 11.2.2.

- The chop visualizer ('ACV') provides a display of relationships between procedures or variables in terms of chop size (section 10.4).

14.1.8 An Approximate Dynamic Slicer

The 'ADS' tool provides a simple technique for approximate dynamic slicing [ADS93]: By an analysis of the program dependence graph a program is instrumented to trace entries to statement blocks. The set of visited statements and their corresponding nodes are inferred from such a trace. A static slice that only traverses visited nodes is an approximation of a dynamic slice. It is less precise than a real dynamic slice but has much smaller runtime complexity.

The ADS tool instruments the program by generating breakpoints for the GDB debugger. Every breakpoint, that is hit during execution emits a trace entry and deletes itself afterwards. Therefore every breakpoint is never executed more than once and the runtime overhead is only equivalent to the program size.

14.1.9 The Duplicated Code Detector

The last tool ('Duplix') is the implementation of the approach to identify similar code from chapter 12.

14.2 Other Systems

Instead of a section for related work, this chapter closes with a list of other slicing systems. Most of the slicing systems had been research prototypes and were not available or vanished in the last years. This list is not supposed to be complete. Some comparative studies have been done to subsets of the following list [HKF95].

Aristotle

The Aristotle Analysis System [HR97, HC98] is targeted at the development of software engineering tools and provides program analysis information. One part of the system is a control flow graph based slicer. It is able to analyze C programs.

CANTO

The CANTO environment [AFL+97] integrates fine-grained information for source code with architectural views. It has a slicer built in, together with a graph tool PROVIS based on dot which can visualize PDGs (besides other graphs). To slice a program, it is transformed into an intermediate control flow graph language first.

ChopShop

ChopShop was a reverse engineering tool to help programmers to understand unfamiliar C code. It accepts full ANSI C and generates program slices in textual and pictorial form. Chopping was introduced by this tool, though only intraprocedural chops could be computed [JR94b, JR94a].

CodeSurfer

The commercially available CodeSurfer [AT01] is the successor of the Wisconsin Program-Integration System. It is the most advanced, complete and stable slicing tool. Its primary target is program understanding of ANSI C programs.

It can visualize call graphs graphically, procedures are textually visualized. For better usability, other elements like variables or files can be browsed hierarchically. Programs can be sliced and chopped in various way.

Similar to our tool, the main data structure is the system dependence graph of a program. CodeSurfer can be programmed using its scripting language (Scheme). The scripting has access to the complete dependence graphs through an API. Thus, CodeSurfer can be used as an infrastructure for other program analyses, e.g. model checking [ERT01].

FOCUS

FOCUS has been developed by Lyle to evaluate the usability of slicing for debugging [Lyl84]. Although first built to slice FORTRAN programs, it was extended to support C programs. The implemented slicer solves data flow equations to compute intraprocedural slices.

Ghinsu

Ghinsu is a SDG-based environment to compute and visualize slices (both forward and backward), dices, and dependences. Similar to our approach, the SDG is not statement based, but based on tokens from the abstract syntax tree [LR92, LA93, LC94a, LC94b, LJ00]. Ghinsu is targeted at ANSI C. However, unstructured control flow (`goto`, `break`, `continue` and long jumps) is not allowed and pointers are not analyzed (only pointers to pass variables as call-by-reference).

Osaka

The Osaka slicer can do static, dynamic and *call-mark slicing* on a pascal subset (no pointers) [NJKI99].

PELAS

The PELAS debugging environment has been extended with an dynamic slicing tool [KR97a]. Besides normal dynamic slices, the tool can compute partial dynamic slices which are restricted to subsequences of traces.

Slash

This research prototype tool was implemented to evaluate Kamkar's slicing algorithms [Kam93]. Its main focus is dynamic slicing of Pascal programs (only a subset of the language can be used). The algorithms are based on program dependence graphs. A graphical view visualizes the execution tree and the resulting slice.

Sprite

Sprite is a slicing tool built on top of Icaria and Ponder [AG96, AG98, BAG00, AG01b, MACE02]. Ponder is a language independent infrastructure to build tools for performing syntactic and semantic analyses of large software systems. Icaria is the language dependent component for ANSI C and contains the Sprite tool, which is able to slice ANSI C. It contains a textual visualization of slices and is able to do typical set operations on slices. The implemented slicing tool uses a Weiser-style algorithm via data flow analysis iterating over a control flow graph. All three tools are freely available; a comparison to CodeSurfer has been done in [BAG00].

Spyder

Spyder [Agr91, ADS93] is a tool to do static and dynamic slicing (although the focus of the research prototype was dynamic slicing). It is built on top of the GNU compiler GCC and its debugger GDB and acts as a graphical user interface to the debugger. These tools have been extended to compute and use program dependence graphs. Static and dynamic slices can be highlighted in the source code.

Steindl's slicer

Steindl's slicer [Ste98, Ste99a, Ste99b] is a slicer for Oberon built on top of an Oberon infrastructure. The (textual) visualization includes graphical elements like pop-ups for visualization and navigation along data dependences and calls. It also features bidirectional feedback: The user of the slicer can disable potential aliases between variables, enabling more precise slices.

Surgeon's Assistant

The Surgeon's Assistant is a prototype implementation of decomposition slicing which allows selection of variables for slicing and then textually displays the decomposition slices. It contains a Decomposition Slice Display System (DSDS) which visualizes the relationships between the decomposition slices in a graphical display. An integrated editor can be restricted to allow changes only inside or outside slices [Gal90, GL91, Gal92, Gal96, GO97].

Unravel

Unravel [LW97] is a prototype tool that can be used to evaluate ANSI C source code statically by using program slicing. In its target to evaluate high integrity software it is most similar to the presented VALSOFT system. However, it is limited to computing forward and backward slices, which can be combined using set operations. The implemented Weiser-style algorithm is based on data flow equations and control flow graphs.

WPIS

The Wisconsin Program-Integration System [Rep93] is not only able to compute forward and backward slices (and visualizes them in source code and as dependence graphs), but also provides program differencing and integration based on system dependence graphs [HR92]. It is able to do various forward and backward, inter- and intraprocedural slices. The slices are visualized textually in source code. It has been superseded by CodeSurfer.

Chapter 15

Conclusions

Almost 25 years ago, Weiser invented slicing: Slices are program abstractions onto a subset of the original statements where the reduced program has the same behavior at a specified statement. Besides a thorough presentation of underlying theory of slicing and chopping, this thesis examined different algorithms in depth and compared them to newly developed variants. The implementation of the various slicing and chopping algorithms for the VALSOFT system led to several implications:

- The main problem for for precision and scalability is not slicing itself, but the data flow analysis that is needed for building the interprocedural program dependence graph. The handling of data structures, especially the pointer analysis, is responsible for imprecision and complexity.

- A syntax-directed approach to data flow analysis is appealing at first glance. However, an implementation is much more complex than a traditional data flow analysis. In the implemented system, the syntax-directed approach does not even have advantages in runtime.

- The decision to omit a transformation of C programs into an intermediate representation before performing data flow analysis is justified by the requirements of path condition generation. Without such demands, it is advisable to transform programs first: To analyze ANSI C directly, an implementation has to consider every (obscure) detail of this language.

Because both time and space complexity of slicing algorithms depend mainly on the size of the results in terms of nodes and edges in dependence graphs, this thesis investigated techniques to compress those graphs without any loss of precision. Folding nodes and edges provided a substantial size reduction of dependence graphs, resulting in a much smaller runtime for slicing and chopping.

In an digression, this thesis showed another new application of dependence graphs independent from slicing: the detection of duplicated or 'cloned'

code. An advantage of the presented complex approach is the ability to detect (highly) modified clones, confirmed in an external clone detection competition.

Precise slicing of concurrent programs is still a challenge: This thesis presented the first technique to slice concurrent (recursive procedural) programs accurately. This high-precision approach has a high complexity both in space and time but any cheaper approach would cause an unacceptable loss of precision. Future work should develop approximations to the presented solution with lower space and time requirements. Furthermore, the trade-off between approximated and precise solution should be evaluated with real programs.

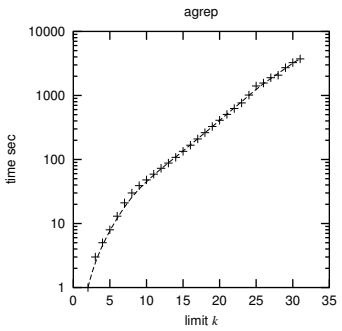
Visualization of dependence graphs together with slices is important for program understanding. Two main approaches have been discussed: textual and graphical representation. The presented specialized dependence graph layout has considerable advantages over general layout techniques and provides comprehensible diagrams for medium sized procedures. However, graphical layout of large procedures is not reasonable.

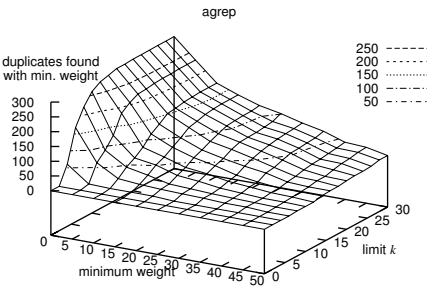
'Pure' slicing is an inadequate mean for a software-(re)engineer: A slice presented as source code or dependence graph does not contain (enough) information to explain why a statement or node affects a criterion. This thesis identified some new slicing and chopping extensions which generate slices and chops more focused to the user's problems. A more advanced 'slice explainer'—the generation of path conditions—has been extended for procedural and concurrent programs as well as complex data structures. Path conditions give sufficient conditions under which an influence from one statement to another may occur. Other work [RS02, SRK03] has already revealed their usefulness, and the thesis of Robschink [Rob] can be seen as a sequel to this work.

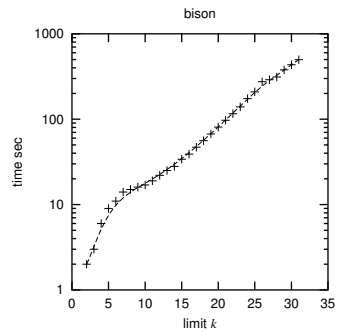
Appendix A

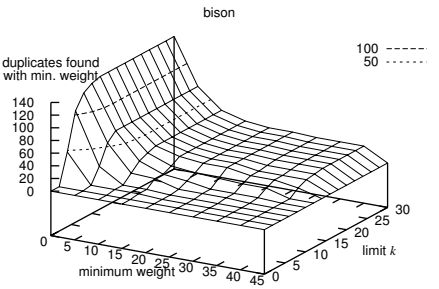
Additional Plots

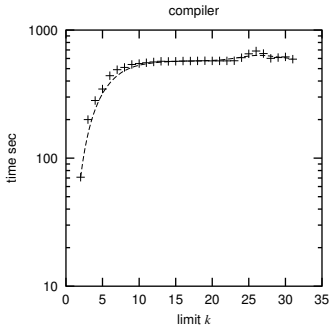
The following plots show data for some of the test cases in Table 12.1 on page 183 for chapter 12, where a technique to identify similar code has been presented. To the left, the running time of the algorithm is shown dependent of the chosen limit k for the maximal path length. To the right, the number of the identified duplicates is shown dependent on limit k and the chosen minimum weight, that a duplicate has to have before it gets reported.

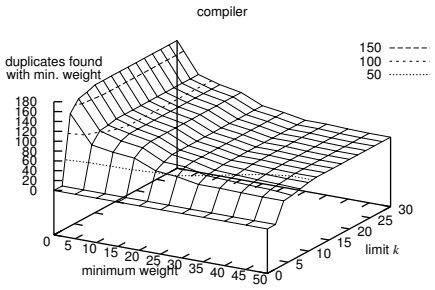


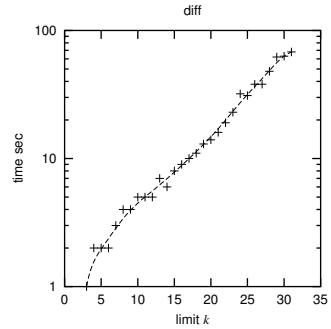


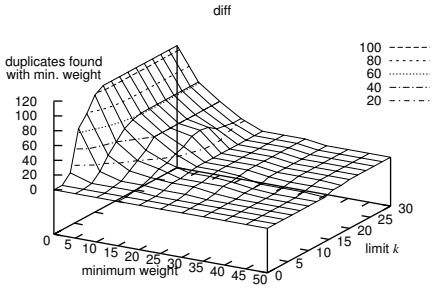


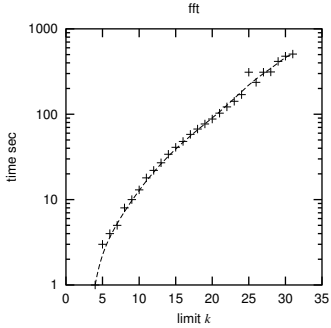


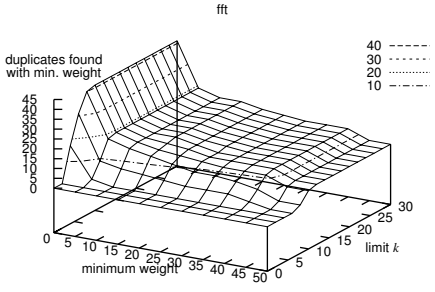

compiler
time sec
limit k

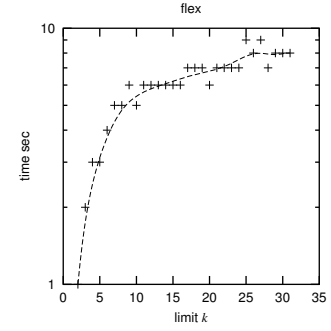

compiler
duplicates found with min. weight
minimum weight
limit k

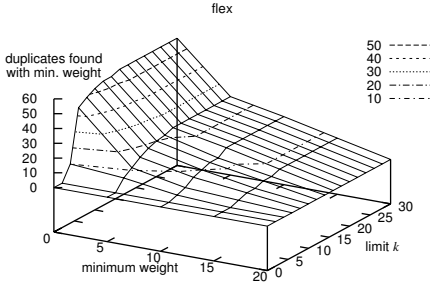

diff
time sec
limit k

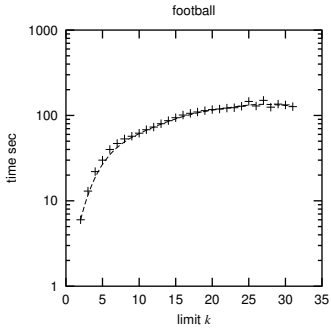

diff
duplicates found with min. weight
minimum weight
limit k

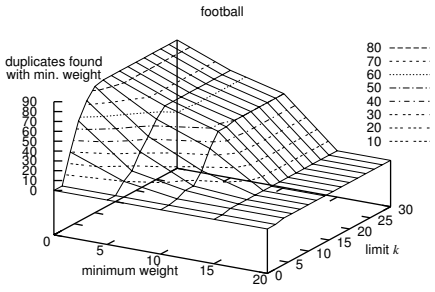

fft
time sec
limit k

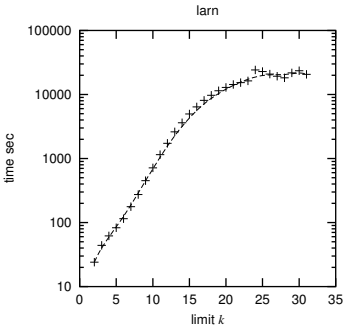

fft
duplicates found with min. weight
minimum weight
limit k

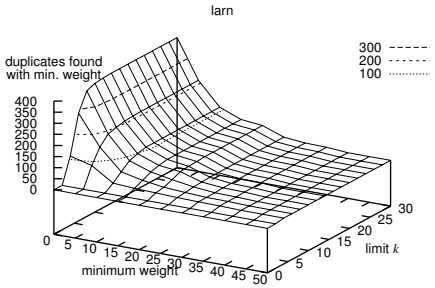

flex
time sec
limit k

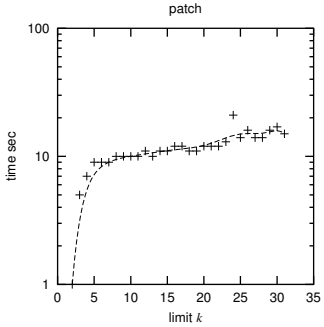

flex
duplicates found with min. weight
minimum weight
limit k

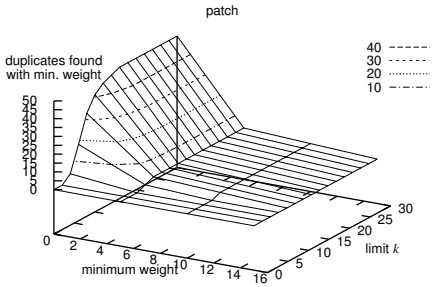

football
time sec
limit k

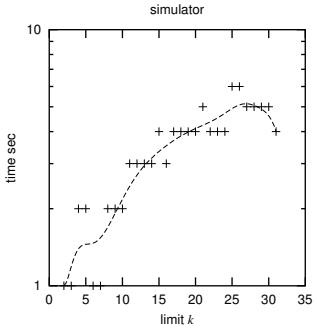

football
duplicates found with min. weight
minimum weight
limit k

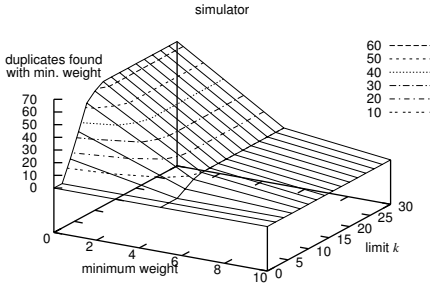

larn
time sec
limit k

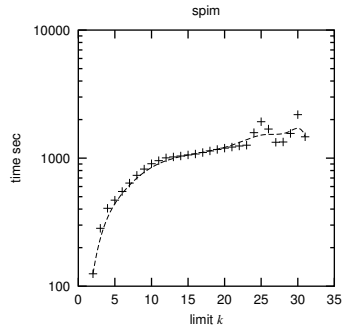

larn
duplicates found with min. weight
minimum weight
limit k

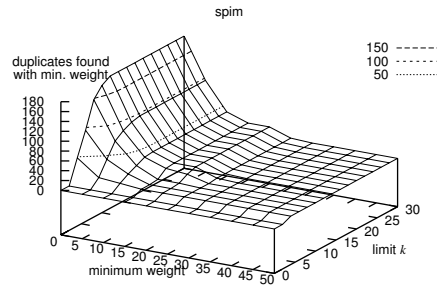

patch
time sec
limit k

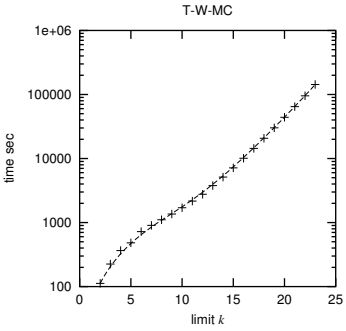

patch
duplicates found with min. weight
minimum weight
limit k

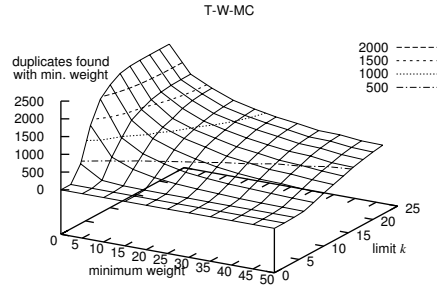

simulator
time sec
limit k


simulator
duplicates found with min. weight
minimum weight
limit k


spim
time sec
limit k
10000
1000
100
0 5 10 15 20 25 30 35


spim
duplicates found
with min. weight
150
100
50
minimum weight
limit k


T-W-MC
time sec
limit k
1e+06
100000
10000
1000
100
0 5 10 15 20 25


T-W-MC
duplicates found
with min. weight
2000
1500
1000
500
minimum weight
limit k

Bibliography

[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, June 1993.

[AFL+97] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo. Program understanding and maintenance with the CANTO environment. In *International Conference on Software Maintenance*, pages 72–81, 1997.

[AG96] Darren C. Atkinson and William G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, 1996.

[AG98] Darren C. Atkinson and William G. Griswold. Effective whole-program analysis in the presence of pointers. In *Foundations of Software Engineering*, pages 46–55, 1998.

[AG01a] Gagan Agrawal and Liang Guo. Evaluating explicitly context-sensitive program slicing. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 6–12, 2001.

[AG01b] Darren C. Atkinson and William G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proc. International Conference on Software Maintenance*, pages 52–61, 2001.

[Agr91] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1991.

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, 1994.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[AM95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symposium*, pages 33–50, 1995.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[AT01] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering (CAV 2001)*, 2001.

[Bac00] Sabine Bachl. *Erkennung isomorpher Subgraphen und deren Anwendung beim Zeichnen von Graphen*. Dissertation, Universität Passau, 2000. (In German).

[BAG00] L. Bent, D. Atkinson, and W. Griswold. A comparative study of two whole-program slicers for C. Technical Report CS2000-0643, Univer. of California at San Diego, 2000.

[Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings: Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[Bal93] Thomas Jaudon Ball. *The Use of Control-Flow and Control Dependence in Software Tools*. PhD thesis, University of Wisconsin-Madison, 1993.

[Bal01] Françoise Balmas. Displaying dependence graphs: a hierarchical approach. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 261–270, 2001.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 234–250. Springer, 1995.

[BE93] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518, 1993.

[BE94] Thomas Ball and Stephen G. Eick. Visualizing program slices. In *IEEE Symposium on Visual Languages*, pages 288–295, 1994.

[Bel02] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Diplomarbeit, Universität Stuttgart, 2002. (In German).

[BFS+02] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *International Conference on Software Maintenance (ICSM'02)*, pages 12–21, 2002.

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

[BG97] Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Conference on Programming Language Design and Implementation*, pages 159–170, 1997.

[BGS+01] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimothy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth Conference on Software Maintenance and Reengineering, CSMR 2001*, pages 105–113, 2001.

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

[BH93b] Samual Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396, 1993.

[Bin92] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 41–50, 1992.

[Bin93a] David Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2(1-4):31–45, 1993.

[Bin93b] David Binkley. Slicing in the presence of parameter aliasing. In *Proceedings of the 1993 Software Engineering Research Forum*, pages 261–268, 1993.

[Bin94] David Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In *Proceedings of the Fifth International Conference on Compiler Construction*, volume 786 of *LNCS*, pages 374–388, 1994.

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

[Bin99] David Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, 1999.

[BL98] David W. Binkley and James R. Lyle. Application of the pointer state subgraph to static program slicing. *The Journal of Systems and Software*, pages 17–27, 1998.

[BO94] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings; International Conference on Software Maintenance*, pages 368–378, 1998.

[CCD98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11–12):595–607, 1998.

[CF89] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989.

[CF94] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, 1994.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[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.

[Che93] Jingde Cheng. Slicing concurrent programs. In *Automated and Algorithmic Debugging, 1st International Workshop, AADEBUG'93*, volume 749 of *LNCS*, pages 223–240. Springer, 1993.

[Che97] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *International Conference on Advances in Parallel and Distributed Computing*, 1997.

[CMN91] J.-D. Choi, B. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[CW97] Jiun-Liang Chen and Feng-Jian Wang. Slicing object-oriented programs. In *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC'97/ICSC'97)*, 1997.

[CX01a] Zhenqiang Chen and Baowen Xu. Slicing concurrent java programs. *ACM SIGPLAN Notices*, 36(4):41–47, 2001.

[CX01b] Zhenqiang Chen and Baowen Xu. Slicing object-oriented java programs. *ACM SIGPLAN Notices*, 36(4):33–40, 2001.

[CXZY02] Zhenqiang Chen, Baowen Xu, Jianjun Zhao, and Hongji Yang. Static dependency analysis for concurrent ada 95 programs. In *7th Ada-Europe International Conference on Reliable Software Technologies*, volume 2361 of *LNCS*, pages 219–230, 2002.

[DCH+99] Matthew B. Dwyer, James C. Corbett, John Hatcliff, Stefan Sokolowski, and Hognjun Zheng. Slicing multi-threaded java programs: A case study. Technical Report KSU CIS TR 99-7, Department of Computing and Information Sciences, Kansas State University, 1999.

[DFHH00] S. Danicic, C. Fox, M. Harman, and R. Hierons. Consit: A conditioned program slicer. In *International Conference on Software Maintenance*, pages 216–226, 2000.

[DGS92] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *5th Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *LNCS*, pages 497–511. Springer, 1992.

[DHS95] Sebastian Danicic, Mark Harman, and Yoga Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313, 1995.

[DKN01] Yunbo Deng, Suraj Kothari, and Yogy Namara. Program slice browser. In *Ninth International Workshop on Program Comprehension (IWPC'01)*, pages 50–59, 2001.

[DL01] Andrea De Lucia. Program slicing: Methods and applications. In *IEEE workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001. Invited paper.

[DLFM96] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, 1996.

[DLHHK03] Andrea De Lucia, Mark Harman, Rob Hierons, and Jens Krinke. Unions of slices are not slices. In *7th European Conference on Software Maintenance and Reengineering*, 2003.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 109–118, 1999.

[DT97] T. B. Dinesh and F. Tip. A case-study of a slicing-based approach for locating type errors. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, 1997.

[EBP01] James Ezick, Gianfranco Bilardi, and Keshav Pingali. Efficient computation of interprocedural control dependence. Technical Report TR2001-1850, Cornell University, 2001.

[Ehr96] Frank Ehrich. Entwurf und Implementierung eines Werkzeugs zur Visualisierung von Programmabhängigkeitsgraphen. Diplomarbeit, TU Braunschweig, 1996. (In German).

[Ern94] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 1994.

[ERT01] James Ezick, David W. Richardson, and Tim Teitelbaum. Practical model checking and example generation for context-free processes. Technical Report TR2002-1851, Cornell University, 2001.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[FG97] Istvan Forgács and Tibor Gyimóthy. An efficient interprocedural slicing method for large programs. In *Proceedings of SEKE'97, the 9th International Conference on Software Engineering & Knowledge Engineering*, pages 279–287, 1997.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FR01] M. A. Francel and S. Rugaber. The value of slicing while debugging. In *Proceedings of the 7th International Workshop on Program Comprehension*, pages 151–169, 2001.

[FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*, pages 379–392, 1995.

[FT98] John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. *Information and Software Technology*, 40(11–12):609–636, 1998.

[FTAM99] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. Points-to analysis for program understanding. *The Journal of Systems and Software*, 44(3):213–227, 1999.

[FW95] M. Fröhlich and M. Werner. Demonstration of the interactive graph visualization system davinci. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing, DIMACS International Workshop GD'94*, volume 894 of *LNCS*. Springer, 1995.

[Gal90] Keith B. Gallagher. Surgeon's assistant limits side effects. *IEEE Software*, 7(64), 1990.

[Gal92] Keith Brian Gallagher. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance*, pages 236–244, 1992.

[Gal96] Keith B. Gallagher. Visual impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 52–58, 1996.

[GH97] Richard Gerber and Seongsoo Hong. Slicing real-time programs for enhanced schedulability. *ACM Transactions on Programming Languages and Systems*, 13(3):525–555, 1997.

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

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[GM99] Valerie Gouranton and Daniel Le Metayer. Dynamic slicing: a generic analysis based on a natural semantics format. *Journal of Logic and Computation*, 9(6):835–871, 1999.

[GM00] D. Goswami and R. Mall. Dynamic slicing of concurrent programs. In *High Performance Computing - HiPC 2000, 7th International Conference*, volume 1970 of *LNCS*, pages 15–26, 2000.

[GM02] D. Goswami and R. Mall. An efficient method for computing dynamic program slices. *Information Processing Letters*, pages 111–117, 2002.

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. DAG - A program that draws directed graphs. *Software, Practice and Experience*, 18(11):1047–1062, 1988.

[GO97] Keith Gallagher and Liam O'Brien. Reducing visualization complexity using decomposition slices. In *Software Visualization Workshop*, pages 113–118, 1997.

[Gop91] Rajiv Gopal. Dynamic program slicing based on dependence relations. In *Conference on Software Maintenance*, pages 191–200, 1991.

[Gri01] William G. Griswold. Making slicing practical: The final mile, 2001. Invited Talk, PASTE'01.

[GS93] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[GS95] Rajiv Gupta and Mary Lou Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 29–40, 1995.

[Hal95] R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, March 1995.

[Hau89] Philip A. Hausler. Denotational program slicing. In 22^{nd} *Annual Hawaii International Conference on System Sciences*, pages 486–495, 1989.

[HC98] Mary Jean Harrold and Ning Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, 1998.

[HCD+99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symposium*, volume 1694 of *LNCS*, pages 1–18. Springer, 1999.

[HD94] M. Harman and S. Danicic. A new approach to program slicing. In *7th International Quality Week*, 1994.

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

[HD97] Mark Harman and Sebastian Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, 1997.

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

[HDC88] J. C. Hwang, M. W. Du, and C. R. Chou. Finding program slices for recursive procedures. In *Proceedings COMPSAC 88: The Twelfth International Computer Software and Applications Conference*, pages 220–227, 1988.

[HDZ00] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library: Programming Language Series. North-Holland, New York, 1977.

[HG98] Mark Harman and Keith Brian Gallagher. Program slicing. *Information and Software Technology*, 40(11–12):577–581, 1998.

[HH01] Mark Harman and Rob Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[HHD99] Robert M. Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[HHD+01] Mark Harman, Robert M. Hierons, Sebastian Danicic, John Howroyd, Mike Laurence, and Chris Fox. Node coarsening calculi for program slicing. In *Working Conference on Reverse Engineering*, 2001.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001.

[HKF95] Tommy Hoffner, Mariam Kamkar, and Peter Fritzson. Evaluation of program slicing tools. In *2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 51–69, 1995.

[HMR93] Mary Jean Harrold, Brian A. Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. In *International Symposium on Software Testing and Analysis*, pages 160–170, 1993.

[Hor90] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[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, 1988.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(4):345–387, July 1989.

[HR91] S. Horwitz and T. Reps. Efficient comparison of program slices. *Acta Informatica*, 28:713–732, 1991.

[HR92] Susan B. Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, 1992.

[HR96] Mary Jean Harrold and Gregg Rothermel. Syntax-directed construction of program dependence graphs. Technical Report OSU-CISRC-5/96-TR32, The Ohio State University, May 1996.

[HR97] M. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, Department of Computer and Information Science, The Ohio State University, 1997.

[HRB88] Susan B. Horwitz, Thomas W. 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(7) of *ACM SIGPLAN Notices*, pages 35–46, 1988.

[HRB90] Susan B. Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HRS98] Mary Jean Harrold, Gregg Rothermel, and Saurabh Sinha. Computation of interprocedural control dependence. In *International Symposium on Software Testing and Analysis*, pages 11–20, 1998.

[HS97] D. Huynh and Y. Song. Forward computation of dynamic slicing in the presence of structured jump statements. In *Proceedings of ISACC'97*, pages 73–81, 1997.

[HSD96] Mark Harman, Dan Simpson, and Sebastian Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8(4):490–497, 1996.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[HW97] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 450–467, 1997.

[INIY96] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on vhdl descriptions and its applications. In *Asian Pacific Conference on Hardware Description Languages (APCHDL)*, pages 132–139, 1996.

[Int90] International Organization for Standardization. *ISO/IEC 9899:1990: Programming languages – C*. International Organization for Standardization, 1990.

[JR94a] Daniel Jackson and Eugene J. Rollins. Abstraction mechanisms for pictorial slicing. In *Proceedings of the IEEE Workshop on Program Comprehension*, pages 82–88, 1994.

[JR94b] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 2–10, 1994.

[JZR91] J. Jiang, X. Zhou, and D. J. Robson. Program slicing for C – The problems in implementation. In *International Conference on Software Maintenance*, pages 182–190, 1991.

[Kam93] Mariam Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993.

[Kam98] Mariam Kamkar. Application of program slicing in algorithmic debugging. *Information and Software Technology*, 40(11–12):637–645, 1998.

[KF92] Bogdan Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science Journal*, 2(2):199–215, 1992.

[KFS93a] Mariam Kamkar, P. Fritzson, and N. Shahmerhi. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming*, 38:625–636, 1993.

[KFS93b] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Interprocedural dynamic slicing applied to interprocedural data how testing. In *Proceedings of the Conference on Software Maintenance*, pages 386–395, 1993.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Eigth International Static Analysis Symposium (SAS)*, volume 2126 of *LNCS*, 2001.

[KH02] Sumit Kumar and Susan Horwitz. Better slicing of programs with jumps and switches. In *Proceedings of FASE 2002: Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 96–112. Springer, 2002.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[KK95] Mariam Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *International Conference on Software Maintenance*, pages 222–231, 1995.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[KL90] Bogdan Korel and Janusz Laski. Dynamic slicing in computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[KN96] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot*. Murray Hill, NJ, 1996.

[Kno98] Jens Knoop. *Optimal Interprocedural Program Optimization*, volume 1428 of *LNCS*. Springer, 1998.

[KO90] Heikki Kälviäinen and Erkki Oja. Comparisons of attributed graph matching algorithms for computer vision. Technical report, Lappeenranta University Of Technology, Finland, 1990.

[Kon97] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings Fourth Working Conference on Reverse Engineering*, pages 44–54, 1997.

[Kor85] Richard E. Korf. Depth-first iterative deepening: An optimal admissable tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Kor95] Bogdan Korel. Computation of dynamic slices for programs with arbitrary control flow. In 2^{nd} *International Workshop on Automated Algorithmic Debugging (AADEBUG'95)*, 1995.

[Kor97] Bogdan Korel. Computation of dynamic slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, 1997.

[KR97a] Bogdan Korel and Juergen Rilling. Application of dynamic slicing in program debugging. In *Automated and Algorithmic Debugging*, pages 43–58, 1997.

[KR97b] Bogdan Korel and Juergen Rilling. Dynamic program slicing in understanding of program execution. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 80–89, 1997.

[KR98] Bogdan Korel and Juergen Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11–12):647–659, 1998.

[Kri98] Jens Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42. ACM Press, 1998. ACM SIGPLAN Notices 33(7).

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[Kri02] Jens Krinke. Evaluating context-sensitive slicing and chopping. In *International Conference on Software Maintenance*, pages 22–31, 2002.

[KS92] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *International Conference on Compiler Construction*, pages 125–140, 1992.

[KS98] Jens Krinke and Gregor Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11-12):661–675, December 1998.

[KSF92] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of the* 4^{th} *Conference on Programming Language Implementation and Logic Programming*, pages 370–384, 1992.

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[KY94] Bogdan Korel and S. Yalamanchili. Forward derivation of dynamic slices. In *Symposium on Testing, Analysis, and Verification*, pages 66–79, 1994.

[LA93] Panos E. Livadas and Scott D. Alden. A toolset for program understanding. In Bruno Fadini and Vaclav Rajlich, editors, *Proceedings of the IEEE Second Workshop on Program Comprehension*, 1993.

[Lak92] Arun Lakhotia. Improved interprocedural slicing algorithm. Technical Report CACS TR-92-5-8, University of Southwestern Louisiana, 1992.

[Lak93] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, pages 273–284, 1993.

[LB93] James R. Lyle and David Binkley. Program slicing in the presence of pointers. In *Proceedings of the 1993 Software Engineering Research Forum*, pages 255–260, 1993.

[LC94a] P. Livadas and S. Croll. A new algorithm for the calculation of transitive dependences. *Journal of Software Maintenance*, 6:100–127, 1994.

[LC94b] 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.

[LD98] Arun Lakhotia and Jean-Christophe Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11–12):677–690, 1998.

[LD99] Arun Lakhotia and Jean-Christophe Deprez. Restructuring functions with low cohesion. In *Working Conference on Reverse Engineering*, pages 36–46, 1999.

[LGB99] Steve Lawrence, C. Lee Giles, and Kurt Bollacker. Digital libraries and Autonomous Citation Indexing. *IEEE Computer*, 32(6):67–71, 1999.

[LH96] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *18th International Conference on Software Engineering*, pages 495–505, 1996.

[LH98] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference On Software Maintanence*, pages 358–367, 1998.

[LH99a] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Foundations of Software Engineering*, pages 199–215, 1999.

[LH99b] Donglin Liang and Mary Jean Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the International Conference On Software Maintanence*, 1999.

[Lit01] Tim Littlefair. *An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Faculty of Communications, Health and Science, Edith Cowan University, 2001.

[LJ00] Panos E. Livadas and Theodore Johnson. An optimal algorithm for the construction of the system dependence graph. *Information Sciences*, 125(1-4):99–131, 2000.

[LPP70] D. C. Luckham, D. M. R. Park, and M. S. Paterson. On formalised computer programs. *Journal of Computer and System Sciences*, 4(3):220–249, June 1970.

[LR92] Panos E. Livadas and Prabal K. Roy. Program dependence analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 356–365. IEEE Computer Society Press, 1992.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[LV93] F. Lanubile and G. Visaggio. Function recovery based on program slicing. In *Proceedings of the International Conference on Software Maintenance*, pages 396–405, 1993.

[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.

[LW87] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *2 International Conference on Computers and Applications*, pages 877–882, 1987.

[LW97] J. Lyle and D. Wallace. Using the unravel program slicing tool to evaluate high integrity software. In *Proceedings of Software Quality Week*, 1997.

[Lyl84] James R. Lyle. *Evaluating Variations of Program Slicing for Debugging*. PhD thesis, University of Maryland, 1984.

[MACE02] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, 2002.

[Mar99] Florian Martin. *Generating Program Analyzers*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1999.

[MC88] B. P. Miller and J.-D. Choi. A mechanism for effiecient debugging of parallel programs. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 135–144, 1988.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[MLM96] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–254, 1996.

[MMKM94] B. A. Malloy, J. D. Mcgregor, A. Krishnaswamy, and M. Medlkonda. An extensible program representation for object-oriented software. *ACM SIGPLAN Notices*, 29(12):38–47, 1994.

[MMS02] G.B. Mund, R. Mall, and S. Sarkar. An efficient dynamic program slicing technique. *Information and Software Technology*, 44(2):123–132, 2002.

[MOS01] Markus Müller-Olm and Helmut Seidl. On optimal slicing of parallel programs. In *STOC 2001 (33th ACM Symposium on Theory of Computing)*, pages 647–656, 2001.

[MT98] L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking. In *Proceedings of the 4th International SPIN Workshop*, 1998.

[MT00] Lynette I. Millett and Tim Teitelbaum. Issues in slicing promela and its applications to model checking, protocol understanding, and simulation. *International Journal on Software Tools for Technology Transfer*, 2(4):343–349, 2000.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NA98] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of 6th International Symposium on the Foundations of Software Engineering*, pages 24–34, 1998.

[NAC99] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In O. Nierstrasz and M. Lemoine, editors, *7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 338–354. Springer, 1999.

[NJKI99] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. Call-mark slicing: An efficient and economical way of reducing slice. In *International Conference of Software Engineering*, pages 422–431, 1999.

[NNH99] F. Nielson, H. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[NR00] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *International Conference on Software Testing and Analysis (ISSTA 2000)*, pages 180–190, 2000.

[OB98] Linda M. Ott and James M. Bieman. Program slices as an abstraction for cohesion measurement. *Information and Software Technology*, 40(11-12):691–700, 1998.

[OHFI01] Fumiaki Ohata, Kouya Hirose, Masato Fujii, and Katsuro Inoue. A slicing method for object-oriented programs using lightweight dynamic information. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, 2001.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19(5) of *ACM SIGPLAN Notices*, pages 177–184, 1984.

[OSH01a] A. Orso, S. Sinha, and M. Harrold. Effects of pointers on data dependences. In *Proc. of the 9th International Workshop on Program Comprehension*, 2001.

[OSH01b] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *International Conference on Software Maintenance*, 2001.

[OT89] Linda M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM conference on Software Engineering*, pages 198–204, 1989.

[OT93] Linda M. Ott and Jeff J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, 1993.

[Ott92] Linda M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium*, pages 16–23, 1992.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[PMP00] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultät für Informatik, Universität Karlsruhe, Germany, 2000.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

[RC00] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

[Rep93] Thomas Reps. *The Wisconsin Program-Integration System 2.0 Reference Manual*. University of Wisconsin, Madison, 1993.

[Rep98] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 1998.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[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.

[RLP+01] Barbara G. Ryder, W. Landi, B. Philip, A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, March 2001.

[Rob] Torsten Robschink. Phd thesis. In preparation.

[ROMA92] Debra J. Richardson, T. Owen O'Malley, Cynthia Tittle Moore, and Stephanie Leif Aha. Developing and integrating prodag into the arcadia environment. In *Proceedings of the Fifth Symposium on Software Development Environments*, pages 109–119, 1992.

[RR95] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 41–52, 1995.

[RRB99] James Reichwein, Gregg Rothermel, and Margaret M. Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *Conference on Domain Specific Languages*, pages 25–38, 1999.

[RS02] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *Proceedings of the 24th International Conference of Software Engineering (ICSE)*, pages 478–488, 2002.

[RT96] Thomas Reps and Todd Turnidge. Program specialization via program slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 409–429, 1996.

[RT01] Filippo Ricca and Paolo Tonella. Web application slicing. In *International Conference on Software Maintenance*, pages 148–157, 2001.

[Rus02] Jeffry Russel. Program slicing for codesign. In *Tenth International Symposium on Hardware/Software Codesign*, 2002.

[RY89] Thomas Reps and Wuu Yang. The semantics of program slicing and program integration. In *Proceedings of the Colloquium on Current Issues in Programming Languages*, volume 352 of *LNCS*, pages 360–374. Springer, 1989.

[San95] Georg Sander. Graph layout through the VCG tool. In Roberto Tamassia and Ioannis G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD'94*, number 894 in LNCS, pages 194–205. Springer, 1995.

[SD96] Stephane Schoenig and Mireille Ducasse. A backward slicing algorithm for prolog. In *Static Analysis Symposium*, pages 317–331, 1996.

[Sel89] Rebecca Parsons Selke. A rewriting semantics for program dependence graphs. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 12–24, 1989.

[SH96] Anthony M. Sloane and Jason Holdsworth. Beyond traditional program slicing. In *International Symposium on Software Testing and Analysis*, pages 180–186, 1996.

[SH97] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings from the 4th International Static Analysis Symposium*, volume 1302 of *LNCS*, pages 16–34, 1997.

[SHD97] Yoga Sivagurunathan, Mark Harman, and Sebastian Danicic. licing, I/O and the implicit state. In 3rd *International Workshop on Automated Debugging (AADEBUG'97)*, pages 59–65, 1997.

[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.

[SHR01] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. Interprocedural control dependence. *ACM Transactions on Software Engineering and Methodology*, 10(2):209–254, 2001.

[Sne96] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis Symposium*, volume 1145 of *LNCS*, pages 332–348. Springer, 1996.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[SRK03] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. Submitted for publication, 2003.

[SS93] V. Sarkar and B. Simons. Parallel program graphs and their classification. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *LNCS*, pages 633–655. Springer, 1993.

[SS00] Helmut Seidl and Bernhard Steffen. Constraint-based interprocedural analysis of parallel programs. In *Proceedings of ESOP'00, 9th European Symposium on Programming*, volume 1782 of *LNCS*, 2000.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Ste98] Christoph Steindl. Intermodular slicing of object-oriented programs. In *International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 264–278. Springer, 1998.

[Ste99a] Christoph Steindl. Benefits of a data flow-aware programming environment. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, 1999.

[Ste99b] Christoph Steindl. *Program Slicing for Object-Oriented Programming Languages*. PhD thesis, Johannes Kepler University Linz, 1999.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, February 1981.

[TAFM97] Paolo Tonella, Giuliano Antoniol, Roberto Fiutem, and Ettore Merlo. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 433–444, 1997.

[TCFR96] F. Tip, J-D Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in C++. In *Conference on Object-oriented Programming Systems, Languages and Applications*, pages 179–197, 1996.

[TD01] Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, January 2001.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3), September 1995.

[TOI01] T. Takada, F. Ohata, and K. Inoue. Dependence-cache slicing: A program slicing method using lightweight dynamic information. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 169–177, 2001.

[UHS97] N. Uchihira, S. Honiden, and T. Seki. Hypersequential programming. *IEEE Concurrency*, pages 44–54, 1997.

[VA00] Wamberto Weber Vasconcelos and Marcelo A. T. Aragao. Slicing knowledge-based systems: Techniques and applications. *Knowledge Based Systems*, 13(4), 2000.

[Vas99] Wamberto Weber Vasconcelos. A flexible framework for dynamic and static slicing of logic programs. In *Proceedings of PADL'99*, volume 1551 of *LNCS*, pages 259–274, 1999.

[Ven91] Guda A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 26–28, 1991.

[Ven95] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, March 1995.

[vG01] R. van Glabbeek. *Handbook of Process Algebra*, chapter The Linear Time - Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes. Elsevier, 2001.

[VW96] K. L. Verco and M. J. Wise. Plagiarism à la mode: a comparison of automated systems for detecting suspected plagiarism. *The Computer Journal*, 39(9):741–750, 1996.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–310, 1994.

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

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

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

[Wis92] Michael J. Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. In *Proceedings of the 23rd Technical Symposium on Computer Science Education*, SIGSCE Bulletin, 1992.

[ZCU96] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Static slicing of concurrent object-oriented programs. In *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pages 312–320, 1996.

[ZCU98] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. A dependence-based representation for concurrent object-oriented software maintenance. In *Procedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 60–66, 1998.

[Zha98] Jianjun Zhao. Applying slicing technique to software architectures. In *Proceedings of 4th IEEE International Conference on Engineering of Complex Computer Systems*, pages 87–98, 1998.

[Zha99a] Jianjun Zhao. Multithreaded dependence graphs for concurrent java programs. In *Proceedings of 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 13–23, 1999.

[Zha99b] Jianjun Zhao. Slicing concurrent Java programs. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, pages 126–133, 1999.