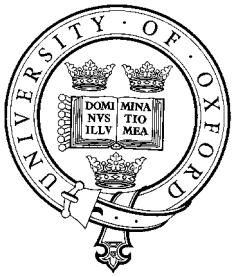
# Program Slicing for Refactoring

Mathieu Verbaere

Lady Margaret Hall




Oxford University
Computing Laboratory

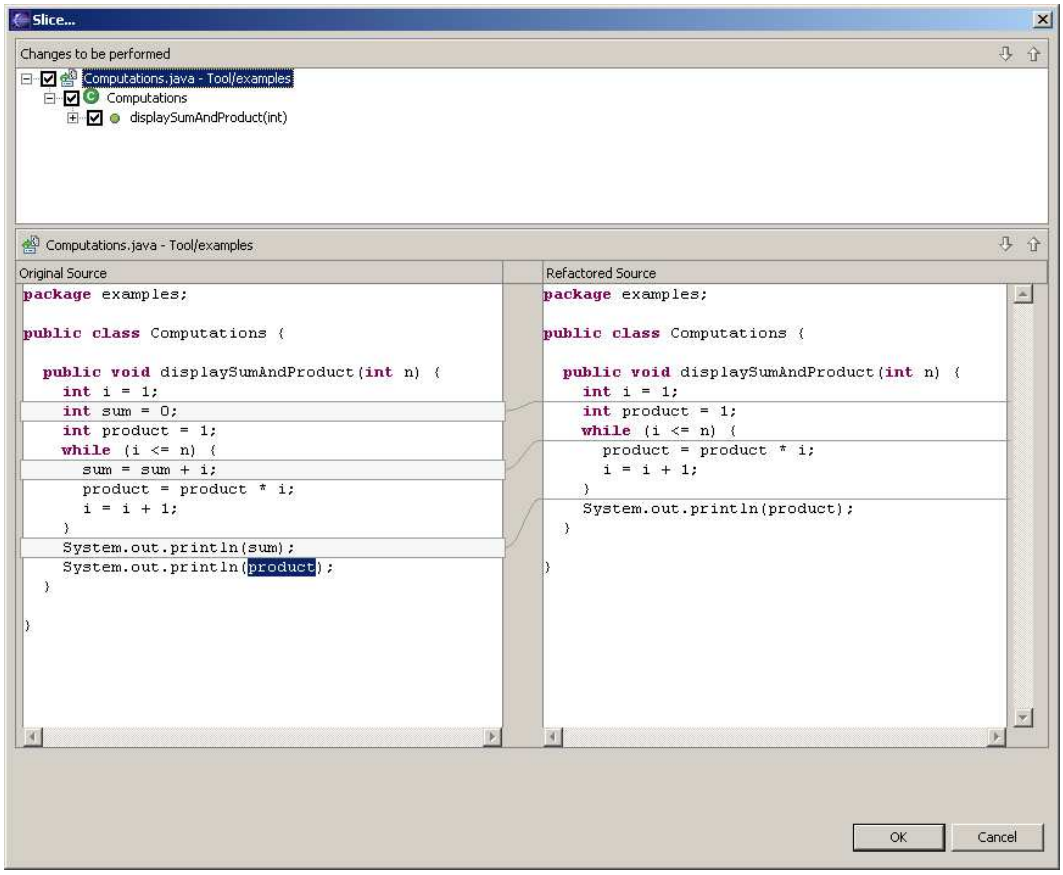
September 2003

# Abstract

A *program slice* singles out all statements that may have affected the value of a given variable at a specific program point. The screenshot below gives an example. According to Mark Weiser, who introduced the concept of *program slicing* in the early eighties, a slice is the mental abstraction people make when they are debugging a program. Some other software engineering tasks, such as *refactoring*, can also be improved with the use of automatic slicing.

Several algorithms have been proposed to compute slices, from slicing as an iterative dataflow equations problem to slicing as a dependence graph reachability problem. Although the latter one is the most common, building a graph representation of a whole program appears to be inappropriate for refactoring.

This thesis draws up a variant of the former: an iterative context-sensitive analysis using *inference rules* which are computed on demand and cached for possible reuse. The new algorithm has been implemented as a tool for the *Eclipse* project in order to experiment with the slicing of programs written in the *Java* language.




Statements that contributed to the value of *product*.

# Acknowledgments

I thank my supervisor, Professor Oege de Moor, for getting me involved with the project of the Programming Tools Group about refactoring into aspects, and for his advices and guidance.

I am also deeply indebted to Ran Ettinger, who works on the project. He has introduced me to the program slicing research domain and has encouraged me a lot in drawing up a novel slicing approach for refactoring. I have enjoyed our discussions on refactoring, aspect-oriented programming and was really amazed on many occasions by his views on software engineering in general.

# Contents


**1 Introduction** **1**

1.1 Motivation . . . . . 1

1.2 Contribution . . . . . 3

1.3 Outline . . . . . 3

**2 Background on program analysis** **5**

2.1 Overview . . . . . 5

2.2 Control flow analysis . . . . . 6

2.3 Data flow analysis . . . . . 7

**3 Program slicing** **10**

3.1 An overview of slicing properties . . . . . 10

3.1.1 Static or dynamic slicing . . . . . 10

3.1.2 Backward or forward computation . . . . . 11

3.1.3 Amorphous slicing . . . . . 11

3.1.4 Intraprocedural and interprocedural slicing . . . . . 12

3.2 Current slicing algorithms . . . . . 12

3.2.1 Data flow equations . . . . . 12

3.2.2 Dependence graphs . . . . . 14

3.3 Context sensitivity . . . . . 18

3.3.1 The calling context problem . . . . . 18

3.3.2 A solution for dependence graphs . . . . . 18

**4 Slicing using inference rules** **20**

4.1 Motivation . . . . . 20

4.2 Definition of a WHILE language . . . . . 21

4.3 The inference rules approach . . . . . 22

4.3.1 The slicing context . . . . . 22

4.3.2 Basic inference rules . . . . . 23

4.3.3 Slicing with a derivation tree . . . 26
4.4 Slicing interprocedurally . . . 27
4.4.1 Extension of the WHILE language . . . 27
4.4.2 Interprocedural rules . . . 29
4.4.3 Speeding up slicing . . . 32
4.4.4 Building the rules . . . 34
4.4.5 Recursive calls . . . 35
4.5 More features . . . 36
4.5.1 Complex expressions . . . 36
4.5.2 Array accesses . . . 37
4.5.3 Variables declarations . . . 38
4.5.4 Structured jumps . . . 38
4.5.5 General slicing criteria . . . 39
4.5.6 Aliasing . . . 40
4.5.7 Object-oriented slicing . . . 40

**5 Implementation** **42**
5.1 The Eclipse project . . . 42
5.1.1 Presentation . . . 42
5.1.2 Plug-in development . . . 44
5.1.3 Java development tools . . . 45
5.2 Design . . . 46
5.2.1 The slicing context . . . 46
5.2.2 Inference rules . . . 46
5.2.3 Intensive use of the Visitor pattern . . . 47
5.3 Experimentation . . . 48
5.3.1 Implemented features . . . 48
5.3.2 A novel refactoring tool . . . 49

**6 Conclusion** **50**
6.1 Summary . . . 50
6.2 Future work . . . 50

**A The calling context problem** **54**

**B Screenshots** **57**

# Chapter 1

# Introduction

## 1.1 Motivation

*Refactoring* is a technique to gradually improve the design of an existing software by performing source code transformations. Some new agile methodologies of software development such as *eXtreme Programming* and *Test-Driven Development* rely mostly on this technique to help in achieving a safe and reusable design.

Object-Oriented Programming is already well known to ease the design of reusable software. Yet, due to naming, typing and scoping dependences, changes of an object-oriented software are more complex than they seem to be a priori. Therefore, even if refactoring can be done manually, automatic refactoring tools are likely to be helpful.

Martin Fowler, who contributed a lot in making refactoring popular [1], has been watching in the past few years for the refactoring's Rubicon to be crossed, which he defines as the ability to automatically extract a method. He considers this refactoring as a key one since it requires the analysis and the manipulation of the abstract syntax tree of a program. Indeed, if you have a code fragment that can be grouped together and want to turn this fragment into a new dedicated method like in Figure 1.1, you need to analyse the original method, find any temporary variables, then figure out what to do with them.

Some Integrated Development Environments have recently crossed the Rubicon and provide refactoring features which are not obvious anymore. In the case of the *Extract Method* refactoring, you have to highlight the whole block you want to extract before calling the automatic tool. The tool checks of course whether the refactoring is possible before proceeding to any source transformations. For example, if several primitive type variables are modified in the highlighted block and used afterwards then the block can not be turned into a single method as only one value can be returned from it. In such a case, the refactoring is automatically rejected.

| Original method | Refactored code |
|---|---|
| `void printMedium(Segment s) {`<br>`  Point m = new Point();`<br>`  m.x = ( s.p1.x + s.p2.x ) / 2;`<br>`  m.y = ( s.p1.y + s.p2.y ) / 2;`<br>`  print("medium("+s+")="+m);`<br>`}` | `void printMedium(Segment s) {`<br>`  Point m = getMedium(s);`<br>`  print("medium("+s+")="+m);`<br>`}`<br><br>`Point getMedium(Segment s) {`<br>`  Point m = new Point();`<br>`  m.x = ( s.p1.x + s.p2.x ) / 2;`<br>`  m.y = ( s.p1.y + s.p2.y ) / 2;`<br>`  return m;`<br>`}` |

Figure 1.1: The *Extract Method* refactoring on a Java piece of code.

Nevertheless, the *Extract Method* refactoring could be even smarter. Indeed, you could highlight only the variable you are interested in and the tool would look for all the statements which must be included in the extracted method and decide if those statements need to be kept or removed from the original method. In order to detect relevant statements, this kind of refactoring requires a complex analysis of the source code, which is known as *program slicing*.

The concept of *program slicing* was originally introduced by Mark Weiser in 1981. He claimed a slice to be the mental abstraction people make when they are debugging a program. A slice consists of all the statements of a program that may affect the values of some variables in a set $V$ at some point of interest $p$. The pair $(p, V)$ is referred to as a *slicing criterion*. Figure 1.2 shows a traditional example of a slice in pseudo-code. Many applications have already been foreseen: debugging, code understanding, reverse engineering, program testing and metrics analysis [Wei84].

| Original program | Slice |
|---|---|
| `1 read(n);`<br>`2 i := 1;`<br>`3 sum := 0;`<br>`4 product := 1;`<br>`5 while (i <= n) {`<br>`6  sum := sum + i;`<br>`7  product := product * i;`<br>`8  i := i + 1;`<br>`  }`<br>`9 print(sum);`<br>`10 print(product);` | `1 read(n);`<br>`2 i := 1;`<br>`3`<br>`4 product := 1;`<br>`5 while (i <= n) {`<br>`6`<br>`7  product := product * i;`<br>`8  i := i + 1;`<br>`  }`<br>`9`<br>`10 print(product);` |

Figure 1.2: A slice for the criterion $(10, \{product\})$.

In addition to these applications, program slicing opens the way with some new advanced refactoring capabilities. Far beyond the current automatic extraction of methods, it could also help taking advantage of new paradigms such as Aspect-Oriented Programming. AOP is a new technology for separation of concerns in software development [2]. It makes possible the modularization of crosscutting aspects of a system (like debugging or logging) which are usually tangled at many places in a program. You could think of an aspect as an observer of a base program. When certain events occur in the base program, some additional code that belongs to the aspect is executed. Existing software would benefit in readability and reusability if some of their aspects were extracted from the original source. This task, which is again very fastidious to do manually, could be automated. An *Extract Aspect* refactoring tool would use program slicing to identify the code that needs to be turned into an aspect.

## 1.2 Contribution

However, the aim of this thesis is not to explain how these new refactoring tools might benefit from program slicing. We take it for granted that advanced refactoring tools *will* somehow need program slicing techniques, and therefore we shall design a novel variant of common slicing approaches to fit refactoring's constraints.

Indeed, most of the applications of program slicing require different properties of slices. For example, some will require that the computed slice is still executable. Others will only need relevant statements to be highlighted among the original program without checking whether those statements form a valid stand-alone program or not. Various notions of program slices have been proposed and, a fortiori, different algorithms to compute them. Yet, no algorithm seems to be appropriate for refactoring.

This work is part of a project of the Programming Tools Group whose final aim is to refactor source code into aspects. Therefore, the new algorithm has been implemented to slice a subset of the *Java* programming language and allow a future implementation of an *Extract Aspect* refactoring tool. Why *Java*? There are two main reasons for this choice. Firstly, the aspect-oriented programming extension for *Java* is already available, in particular within the AspectJ project [3]. Secondly, we will be able to develop over the Eclipse project [4]. It provides a full open-source programming environment written in *Java* for the *Java* programming language itself. Any developer can contribute to the platform and add features through the development of plug-ins.

## 1.3 Outline

Chapter 2 gives a brief overview on program analysis. In particular, we introduce notions about control and data flow that will be used throughout the thesis.

Chapter 3 first describes the different concepts of program slices that have been proposed so far and characterizes the kind of slicing we are interested in for refactoring, that is mainly static interprocedural slicing. Then, we present the two most common slicing algorithms, respectively based on data flow equations and on dependence graphs. Last, we focus on the problem of context sensitivity that occurs in the presence of multiple call sites to the same procedure.

Chapter 4 presents our new approach based on inference rules. We first justify the need of a new approach for refactoring. Next, we introduce a simple imperative language in order to formalize our approach. We focus on slicing a single procedure, then on interprocedural slicing. Moreover, we explain how interprocedural rules are built and how they can be computed and cached on demand. Last, we show that our approach is easily extensible to support more features such as complex expressions, variables declarations, structured jumps, aliasing and object-oriented concepts.

Chapter 5 describes the *Eclipse* project and the useful features it provides. We discuss briefly the main data structures of our experimental implementation and we mention the related work that was done thanks to this implementation.

Finally, Chapter 6 contains the conclusion of this thesis presenting results and future work.

# Chapter 2

# Background on program analysis

## 2.1 Overview

Program analysis offers static techniques for predicting, from the code of a program, some properties about its behaviour that arise dynamically at run-time. Practically, it is used in many software engineering tasks. A main application is to allow compilers to generate code with no redundant, superfluous computation. We can cite also debugging, program understanding and any applications that need to combine information from different places of a program.

In fact, among the many analyses that require collecting information about data manipulation within a program, similarities have been identified. In consequence, an underlying framework has been proposed to provide a single formal model that describes all these data flow analyses. This framework exploits lattice theory and considers the program as a transition system. Because monotonicity of the transfer functions from one state to another is required, the framework is commonly known as *monotone framework*. In this thesis, we will not talk more about it but we should keep in mind that what we describe can or could certainly be expressed with the monotone framework notations [LDK99]. Concerning our new slicing approach described in Chapter 4, it could be even an interesting future work and would allow us to prove its correctness (or reveal its limitations) more easily.

However, this chapter, whose material is partly based on the book *Principles of Program Analysis* [NNH99], focuses on basic concepts of program flow analysis, that are relevant for program slicing and useful in the sequel of the thesis.

## 2.2 Control flow analysis

The goal of control flow analysis is to discover the hierarchical flow of control within a procedure. Some special statements of a program, known as control flow statements, are used to conditionally execute other statements (**if**), to repeatedly execute one or more statements (**while**), and to otherwise change the normal sequential flow through structured jumps (**return**, **break**, **continue**) or unstructured jumps (**goto**).

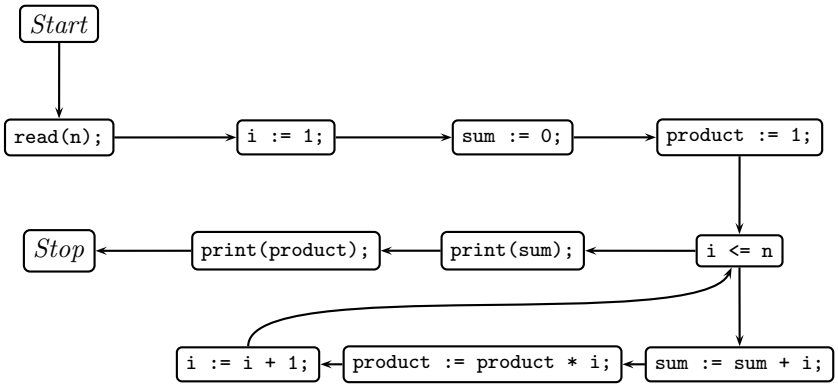



Figure 2.1: CFG of the program of Figure 1.2

The most common representation used to model the control flow of a program is the *Control Flow Graph*. A CFG is a graph whose nodes represent the statements and control predicates of the program and whose directed edges represent control flow: an edge from node $i$ to node $j$ indicates the possible flow of control from $i$ to $j$. A CFG also contains a unique entry node *Start* and a unique exit node *Stop*. Figure 2.1 shows the CFG of the original program of Figure 1.2. Alternatively, the CFG nodes may represent *basic blocks*, which are sequences of instructions that can be entered only at the beginning and exited only at the end. For instance, in Figure 1.2, the three statements of the while loop body form a basic block.

We note $Pred(i)$ and $Succ(i)$ the predecessors and successors of a node $i$. $i \rightarrow_{CFG} j$ denotes that $j$ is an immediate successor of $i$. In addition, the notion of control dependences has been introduced to represent the relations between program statements due to control flow.

**Definition 2.1** *In a CFG, a node* $j$ post-dominates *a node* $i$ *if and only if* $j$ *is a member of any path from* $i$ *to* Stop.

As an example referring to Figure 2.1, node `print(sum);` post-dominates node `sum := 0;`.

**Definition 2.2** *In a CFG, a node $j$ is* control dependent *on a node $i$ if and only if:*

1. *there exists a directed path $P$ from $i$ to $j$ ;*
2. *$j$ post-dominates any $k$ in $P$ such that $k \neq i$, $j$ ;*
3. *$j$ does not post-dominate $i$.*

**Definition 2.3** *The* range of influence *$Infl(i)$ of a node $i$ is the set of nodes that are control dependent on $i$.*

For instance, still in our sample program, only the three statements in the **while** loop are control dependent on node `i <= n`, and hence represent the range of influence of this node.

In our case, although the notions of control flow and control dependences will be largely used in the description of slicing algorithms, we will not need to perform any control flow analysis. Indeed, we will work on the source code of programs written in modern languages that do not allow any unstructured jumps (**goto**). The *Abstract Syntax Tree* of a program, which consists of the data structure obtained after parsing the source and reflecting the structured control flow of such modern languages, will provide us enough information. For example, the statements in the bodies of both the *then* branch and the *else* branch of an **if** are control dependent on its control predicate. Yet, in future work, we may wish to work on the binary code of libraries in order to retrieve accurate information about the data that are manipulated during a library call. In such a case, a control flow analysis will precede any data flow analysis.

## 2.3 Data flow analysis

Data flow analysis provides global information of data manipulation within a program. Among the various data flow analysis problems, we can cite the reaching definitions problem, constant propagation or liveness analysis.

The *reaching definitions* problem is interested in finding, for each program point, which assignment *may* have been made and not overwritten when the program execution reaches this point along the path. One application of reaching definitions is the constructions of direct links between basic blocks that produce values and blocks that use them.

*Constant propagation* aims at propagating any constant assigned to a variable through the flow graph to substitute it at the use of the related variables for optimization purposes.

At last, a variable is said to be live at a particular point in the program if its value at that point will be used in the future. Otherwise, it is dead. Liveness analysis aims at pointing out the live variables of a program.

The data flow describes the flow of values of variables from the point of their definitions to the points where their values are used. It is defined over the CFG of a program as well. Let $i$ be a node of a CFG. The sets $Def(i)$ and $Ref(i)$ denote respectively the sets of variables *defined* (or *modified*) and *referenced* (or *used*) at node $i$.

Most problems can be solved with data flow equations. Two sets are attributed to each statement (or each node in the CFG) of the program. The *In* set of a node $i$ contains relevant information of the problem just when $i$ is reached, whereas the *Out* set contains relevant information when $i$ is exited. The equations concern these two sets for each node of the CFG. For instance, briefly, the equations for liveness analysis are:

$$In(i) = Ref(i) \cup (Out(i) - Def(i))$$
$$Out(i) = \bigcup_{s \in Succ(i)} In(s)$$

Generally, two more sets need to be introduced for each node in order to describe the equations: the *Gen* and the *Kill* sets. $Gen(i)$ contains the new information that is added at node $i$. On the other hand, $Kill(i)$ contains the old information that is removed at node $i$. Problems that are solved with data flow equations and these two sets are known as *Kill/Gen* problems. In fact, liveness analysis is one of them since we can say that a reference of a variable generates liveness and that a definition of variable kills liveness: for any node $i$, $Gen(i) = Ref(i)$ and $Kill(i) = Def(i)$.

Then, an *iterative* data flow analysis is performed until the series of *In*s and *Out*s converge for every node. The algorithm for liveness analysis is given as an example in Figure 2.2. The iterative algorithm is usually improved using a *work list*. When the *In* or *Out* sets change for a single node, a work-list algorithm keeps track of the nodes that must be recalculated instead of computing all the equations again, including those that may not be affected by the change.

Furthermore, the notion of data dependences has been introduced once again.

**Definition 2.4** *In a CFG, a node $j$ is* data dependent *(also called* flow dependent*) on a node $i$ if and only if there exist a variable $x$ such that:*

1. $x \in Def(i)$ ;
2. $x \in Ref(j)$ ;
3. *there exists a path from $i$ to $j$ with no intervening definitions of $x$.*

In fact, data flow analysis is inevitably unprecise when it is performed statically for any possible inputs. Indeed the analysis takes into account some paths that may be unrealizable at run-time. For this reason, we can distinguish *may* and *must* problems. They reveal

foreach node $i$ in the CFG
  $In[i] = \varnothing$
  $Out[i] = \varnothing$
repeat
  foreach node $i$ in the CFG
    $In'[i] = In[i]$
    $Out'[i] = Out[i]$
    $In(i) = Ref(i) \cup (Out(i) - Def(i))$
    $Out(i) = \bigcup_{s \in Succ(i)} In(s)$
until $In'[i] = In[i]$ and $Out'[i] = Out[i]$ for all $i$

Figure 2.2: The iterative data flow analysis for liveness analysis.

respectively what may and must occur on some paths through a flow graph. Depending on the problem you want to solve, you have to consider either a may or a must approach in order to get a safe *conservative* result, i.e. a result which is not the best solution but a valid approximation.

Having introduced the basic notions of data flow analysis, we shall now focus on the particular program analysis that is program slicing.

# Chapter 3

# Program slicing

## 3.1 An overview of slicing properties

As we said in the introduction when defining Weiser's original concept of slicing, various notions of program slices and different algorithms to compute them have been proposed. Hence, many papers in the literature aim at providing a comparison of program slicing techniques. Tip's survey give a complete and very useful overview of both static and dynamic slicing [Tip95].

### 3.1.1 Static or dynamic slicing

In *static* slicing, only statically available information is used for computing slices. No assumption is made on the input of the program. Hence, the computation can not be perfectly accurate. By contrast, *dynamic* slicing aims at providing an accurate slice for a specific execution of a program. The analysis assumes a fixed input that is included in the slicing criterion. Figure 3.1 shows an example of dynamic slice. For $n = -3$, the statement of the *then* branch is not relevant and can be removed. The semicolon is kept in order to preserve the syntax correctness.

| Original program | Slice |
|---|---|
| 1 read(n);<br>2 if (n >= 0)<br>3 sign := +1;<br>else<br>4 sign := -1;<br>5 print(sign); | 1 read(n);<br>2 if (n >= 0)<br>3 ;<br>else<br>4 sign := -1;<br>5 print(sign); |

Figure 3.1: A dynamic slice for the criterion *(n=-3, 5, {sign})*.

For refactoring, we are interested in static slicing because the slices have to be valid for any possible execution of the program. Some further static analyses, such as constant propagation or alias analysis (see 4.5.6), can be performed in order to produce considerably smaller conservative slices. Yet, the computation is then more time-consuming. In the case of a refactoring tool, we need to find a compromise to perform an analysis neither too slow nor too inaccurate. Not too slow because refactoring is performed while editing the source code, and not too inaccurate because a refactoring tool aims at being as efficient as a manual intervention.

### 3.1.2 Backward or forward computation

All the slices we have mentioned so far were computed backwards, that is using a backward traversal of the program flow starting from the point of interest given in the slicing criterion. *Backward* slicing is the one which was introduced originally by Weiser: a slice contains all statements and control predicates that may have influenced a given variable at a given point of interest. By contrast, forward slices contain all statements and control predicates that may be influenced by the variable. *Forward* slicing is used in debugging and program understanding to highlight the downstream code that depends on a specific variable. Nevertheless, backward and forward slices are computed in a similar way. The only difference is the way the flow is traversed.

In this thesis, we focus on backward slicing. It seems indeed more intuitive to extract code on the upstream side of the selected point of interest. Yet, special refactoring tools may need forward slicing and future work could consist in adapting the new algorithm to forward slicing as well. In the sequel, we talk only about backward slicing.

### 3.1.3 Amorphous slicing

In order to be able to rewrite the program in such a way that it can be more easily read, *amorphous* slicing allows any semantics preserving transformations [HD03]. Hence, in the example of Figure 3.1, `if (n >= 0) ; else sign := -1;` could be replaced by `if (n < 0) sign := -1;`.

Moreover, several assignments could be digested into one. For instance, let us consider the following sequel of statements: `t := x * 2; y := t + 1; z:= t + 2;` . The variable $t$ is defined for optimization as we do not want to compute twice the double of $x$. Let us assume now that, for some reasons, the second statement `y := t + 1;` is sliced away. Then, the slice `t := x * 2; z:= t + 2;` should (or at least could) be rewritten `z := x * 2 + 1;`.

Currently, we are not interested in this kind of slicing since we believe the result might be very confusing for a developer in the case of refactoring. Nevertheless, it could be an

option in future refactoring tools based on program slicing. These transformations could also be performed afterwards by an independent tool using a distinct analysis.

### 3.1.4 Intraprocedural and interprocedural slicing

*Intraprocedural* slicing computes slices within one procedure. Calls to other methods, if supported, are handled in a conservative way, assuming the worst possible case that might occur during the call. On the other hand, *interprocedural* slicing can compute slices which span procedures and even different classes and packages when slicing object-oriented programs.

There are two ways in which slicing can cross a procedure boundary. Indeed, it can occur either when the procedure sliced is called by another procedure, or, more intuitively, when a procedure that is sliced contains a call to another procedure. We can think of slicing going respectively *up* or *down*.

Interprocedural slicing raises an important problem: when the same procedure is called at different places in the program, the context in which the call occurs is certainly different. Not taking into account this difference of context can lead to very inaccurate slices. Because program slicing for refactoring must be sufficiently accurate to be useful in a developing process, it has to be interprocedural and account for this difference of context. Solutions that handle this problem are said *context sensitive*. We discuss the issue of context sensitivity in section 3.3.

## 3.2 Current slicing algorithms

In summary, we are interested in static interprocedural backward slicing. We shall now describe the two main approaches that already exist to compute this kind of slices.

### 3.2.1 Data flow equations

When Weiser introduced the concept of slicing [Wei84], he proposed to find slices using data flow equations, in a similar way to liveness analysis. His algorithm uses a CFG as a representation of the program to be sliced and it computes sets of relevant variables for each node.

Let $C = (p, V)$ be a slicing criterion. $R^0_C(i)$, the set of directly relevant variables at statement i, can be determined by only taking into account data dependences and is defined as follows:

1. $R^0_C(i) = V$ when $i = p$
2. $R^0_C(i)$ contains all variables $v$ such that, for any $j$ verifying $i \rightarrow_{CFG} j$, either:

   (i) $v \in \mathit{Ref}(i)$ and $\mathit{Def}(i) \cap R^0_C(j) \neq \varnothing$

or (ii), $v \notin Def(i)$ and $v \in R_C^0(j)$

A set of directly relevant statements, $S_C^0$, is then defined as:

$$S_C^0 = \{i \mid Def(i) \cap R_C^0(j) \neq \emptyset, i \rightarrow_{CFG} j\}$$

In addition, variables referenced in the predicate of a control statement (**if**, **while**) may be indirectly relevant if at least one of the statement in its body is relevant. The resulting relevant control statements are:

$$B_C^0 = \{b \mid i \in Infl(b), i \in S_C^0\}$$

If $b$ is a control statement, $Ref(b)$ is the set of variables which can influence the choice of paths from $b$. The variables at a statement i which directly influence $b$ are $R^0_{(b,Ref(b))}(i)$. The next levels of influence are defined recursively:

$$R_C^{k+1}(i) = R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{(b,Ref(b))}(i)$$
$$S_C^{k+1} = \{i \mid Def(i) \cap R_C^{k+1}(j) \neq \emptyset, i \rightarrow_{CFG} j\} \cup B_C^k$$
$$B_C^{k+1} = \{b \mid i \in Infl(b), i \in S_C^{k+1}\}$$

The function $S_C^{\cdot}$ is non-decreasing. Its least fixed point, $S_C = \bigcup_{k \geq 0} S_C^k$, contains the final relevant statements of the slice for the criterion $C$.

As an example, we shall slice the program of Figure 1.2 for the criterion $(10, \{product\})$. Results are given in the table below. The fixed point is reach at $k = 1$. The relevant statements are the statements in $S_C^1 = \{1, 2, 4, 5, 7, 8\}$.

| Node | *Def* | *Ref* | *Infl* | $R_C^0$ | in $S_C^0$ | in $B_C^0$ | $R_C^1$ | in $S_C^1$ |
|---|---|---|---|---|---|---|---|---|
| 1 | $\{n\}$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | | | $\emptyset$ | $\surd$ |
| 2 | $\{i\}$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\surd$ | | $\{n\}$ | $\surd$ |
| 3 | $\{sum\}$ | $\emptyset$ | $\emptyset$ | $\{i\}$ | | | $\{i, n\}$ | |
| 4 | $\{product\}$ | $\emptyset$ | $\emptyset$ | $\{i\}$ | $\surd$ | | $\{i, n\}$ | $\surd$ |
| 5 | $\emptyset$ | $\{i, n\}$ | $\{6, 7, 8\}$ | $\{product, i\}$ | | $\surd$ | $\{product, i, n\}$ | $\surd$ |
| 6 | $\{sum\}$ | $\{sum, i\}$ | $\emptyset$ | $\{product, i\}$ | | | $\{product, i, n\}$ | |
| 7 | $\{product\}$ | $\{product, i\}$ | $\emptyset$ | $\{product, i\}$ | $\surd$ | | $\{product, i, n\}$ | $\surd$ |
| 8 | $\{i\}$ | $\{i\}$ | $\emptyset$ | $\{product, i\}$ | $\surd$ | | $\{product, i, n\}$ | $\surd$ |
| 9 | $\emptyset$ | $\{sum\}$ | $\emptyset$ | $\{product\}$ | | | $\{product\}$ | |
| 10 | $\emptyset$ | $\{product\}$ | $\emptyset$ | $\{product\}$ | | | $\{product\}$ | |

Figure 3.2: Results of Weiser's algorithm for the slice introduced in Figure 1.2.

We may observe that the tenth statement related to the output of variable *product* is not in the slice. In fact, as an output statement does not define any variable and has an

empty range of influence, it will never be contained in any slice. To overcome this problem, a possible trick is to consider the output of a variable $x$ as the concatenation of $x$ to an *out* variable: `out := out + x;`. Then, if you include *out* in the set of variables of the slicing criterion, all output statements will be contained in the slice.

In addition, Weiser proposes a solution to interprocedural slicing. For each criterion $C$ for a procedure $P$, he defines a set of criteria $Up_0(C)$ which are those needed to slice callers of $P$, and a set of criteria $Down_0(C)$ which are those needed to slice procedures called by $P$. The criteria of the $Down_0(C)$ set (or $Up_0(C)$ set) are computed using the set of relevant variables at each call site, by changing actual arguments of the call site to the parameters of the called procedure (or the opposite) and removing the variables which are not in the scope of the called (or calling) procedure. $Up_0(C)$ and $Down_0(C)$ can then be extended to the functions $Up$ and $Down$ which map sets of criteria to sets of criteria. Let $CC$ be a set of criteria:

$$Up(CC) = \bigcup_{C \in CC} Up_0(C)$$
$$Down(CC) = \bigcup_{C \in CC} Down_0(C)$$

The union and transitive closure of $Up$ and $Down$, denoted $(Up \cup Down)^*$, maps any set of criteria into the criteria necessary to complete all the slices of calling and called procedures. So, the final interprocedural slice for $C$ is the union of the intraprocedural slices (computed with the method outlined in the first part of this subsection) for every criterion in $(Up \cup Down)^*(C)$. We will see in the next section that this method is too inaccurate. But let us first present the other approach, which is the most commonly used.

### 3.2.2 Dependence graphs

Ottenstein and Ottenstein proposed to compute intraprocedural slices as a graph reachability problem using *Program Dependence Graph* [OO84]. A PDG is a graph whose nodes represent the statements and control predicates of the program like the CFG, but whose directed edges represent control dependences and data dependences. It includes also an *entry* node whose range of influence is the set of all the nodes that are not already control dependent on a control predicate. Figure 3.3 shows the PDG of the sample program of Figure 1.2. Thick solid arrows represent control dependences, whereas thin solid ones represent data dependences.

Once this graph has been computed for a single procedure, slicing is straightforwardly solved like any graph reachability problem in linear time. Starting from a set of nodes $V$ we are interested in and using a work-list algorithm, we mark iteratively all the nodes $i$ such that there is an oriented path from $i$ to any node of $V$. The slice with respect to $V$ consists of the resulting set of marked nodes.

We remark that the criteria of PDG-based slicing method are less general than those of slicing based on data flow equations. Indeed, a program can not be sliced with respect to a point of interest $p$ and an arbitrary set of variables $V$. Instead $V$ has be to the set of variables that are defined or referenced at $p$. But we can slice with multiple criteria at the same time, that is more precisely from various point of interests.

Anyway, we can see at a glance that the slice is correct when we slice, like we did before, from node `print(product);` and that, using this method, there is no problem concerning the output statements.

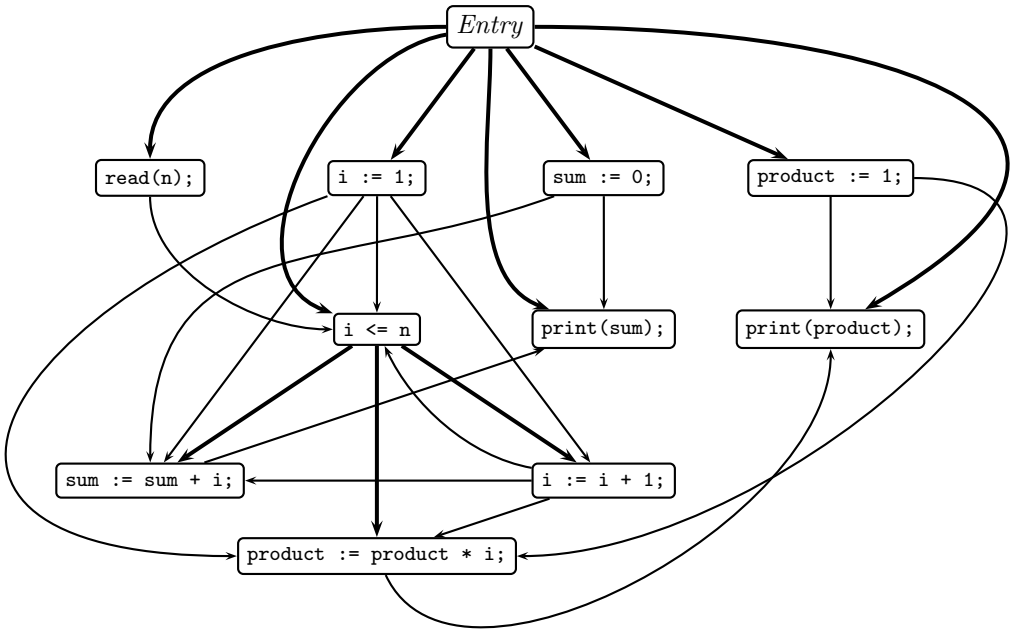



Figure 3.3: PDG of the program of Figure 1.2

Nevertheless, the PDG handles one procedure only. To overcome this limitation, Horwitz, Reps and Binkley introduced an interprocedural dependence graph representation, known as the *System Dependence Graph*, which models procedure calls and parameters passing [HRB90].

The SDG consists of several PDGs (one for each procedure) with some additional nodes and edges. Indeed, it uses a new kind of vertices to represent call sites and procedure entry points and some *call edges* to connect the call sites in the PDGs of the calling procedures to the related entry points in the PDGs of the called procedures. Moreover, it introduces two other kinds of vertices for each *actual* argument in the procedure calls: an *actual-in* vertex to represent the assignment of the actual argument in the calling procedure to an intermediate variable (used to send the input to the called procedure) and an *actual-out*

vertex to represent the assignment of the intermediate variable (used this time to retrieve the final value of the parameter in the called procedure) back to the actual argument. In a similar way, it introduces two new vertices for each *formal* parameter in the called procedures: a *formal-in* vertex and a *formal-out* vertex. Then there are some *parameter-in* and *parameter-out* edges to connect the pairs of intermediate variables from the PDGs of the call sites to the PDGs of the called procedures.

To give an example of SDG, we shall first transform our traditional example into a multiprocedural program. The figure below shows both this new program and the expected slice with the same criterion as before, $(10, \{product\})$.

| Original program | Slice |
|---|---|
| `procedure main {`<br>`1 read(n);`<br>`2 i := 1;`<br>`3 sum := 0;`<br>`4 product := 1;`<br>`5 while (i <= n) {`<br>`6 call add(sum,i);`<br>`7 call multiply(product,i);`<br>`8 call add(i,1);`<br>`}`<br>`9 print(sum);`<br>`10 print(product);`<br>`}`<br><br>`procedure add(a,b) {`<br>`11 a := a + b;`<br>`}`<br><br>`procedure multiply(c,d) {`<br>`12 j := 1;`<br>`13 k := 0;`<br>`14 while (j <= d) {`<br>`15 call add(k,c);`<br>`16 call add(j,1);`<br>`}`<br>`17 c := k;`<br>`}` | `procedure main {`<br>`1 read(n);`<br>`2 i := 1;`<br>`3`<br>`4 product := 1;`<br>`5 while (i <= n) {`<br>`6`<br>`7 call multiply(product,i);`<br>`8 call add(i,1);`<br>`}`<br>`9`<br>`10 print(product);`<br>`}`<br><br>`procedure add(a,b) {`<br>`11 a := a + b;`<br>`}`<br><br>`procedure multiply(c,d) {`<br>`12 j := 1;`<br>`13 k := 0;`<br>`14 while (j <= d) {`<br>`15 call add(k,c);`<br>`16 call add(j,1);`<br>`}`<br>`17 c := k;`<br>`}` |

Figure 3.4: An interprocedural slice for the criterion *(10, {product})*.

The SDG is then drawn in Figure 3.5. In addition to the previous conventions introduced for the PDG, thick and thin dashed arrows represent call and parameter edges respectively.

Computing an interprocedural slice seems still straightforward since it is a graph reachability problem, once again. Yet, like the Weiser's interprocedural solution, it yields inaccurate slices. Let us reveal this lack of accuracy due, in both approaches, to a failure in accounting for the calling context of procedures.

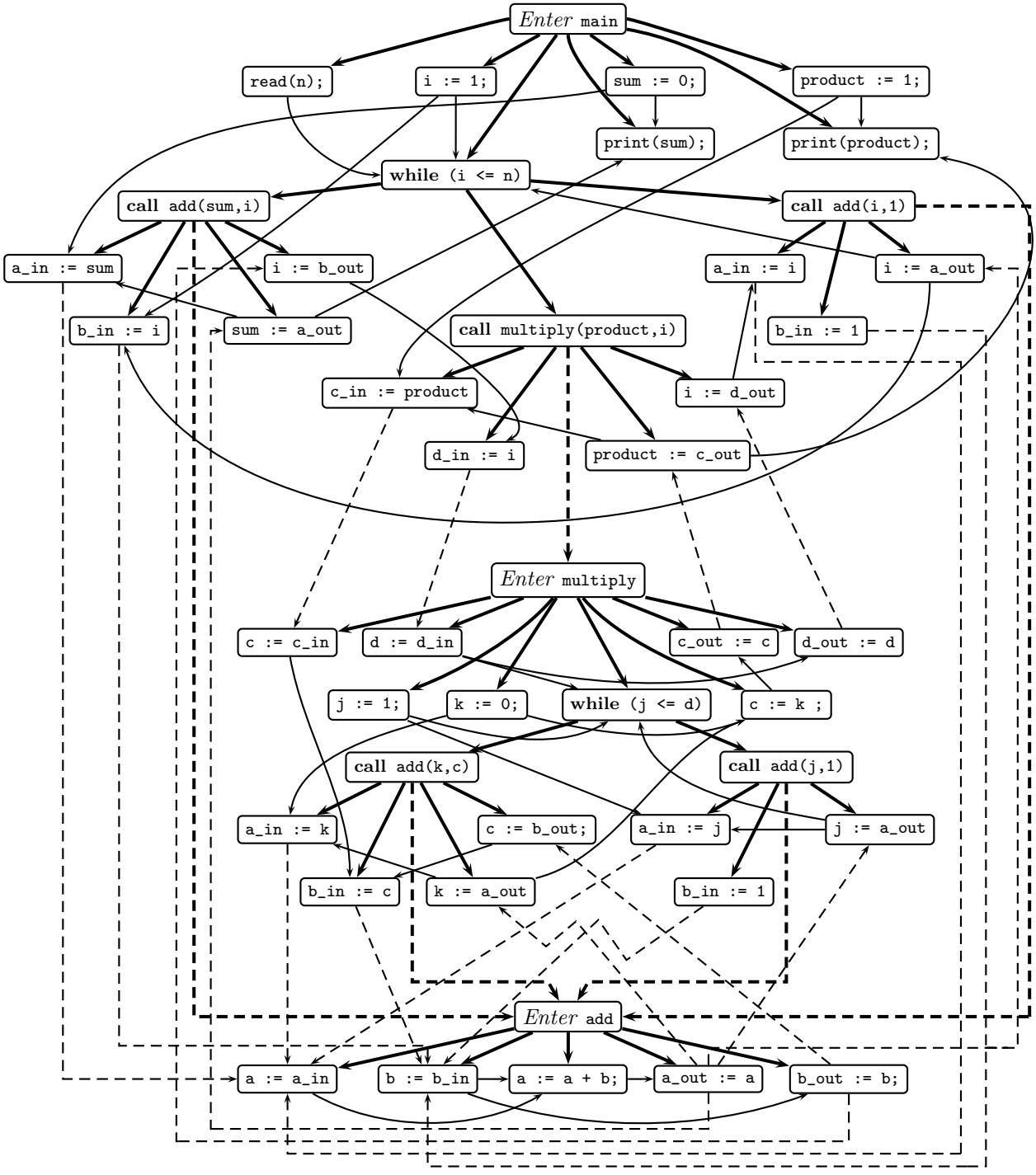



Figure 3.5: SDG of the multiprocedural program of Figure 3.4.

## 3.3 Context sensitivity

### 3.3.1 The calling context problem

If we slice our interprocedural sample with respect to $(10, \{product\})$ using either the equations approach or the current SDG approach, we obtain almost the whole program where statements related to the computation of *sum* are kept as relevant (except for the output of sum which is sliced away).

Indeed, in Weiser's approach, the criterion $< 17, \{k\} >$ is mapped *down* to $< 11, \{a\} >$, which is then mapped *up* to several criteria corresponding to all call sites of *add*. In particular, this includes the irrelevant criterion $< 6, \{sum\} >$. So the statement **call** `add(sum,i);` and consequently the statement `sum := 0;` are considered relevant although they should not be.

In SDG-based slicing, the same problem occurs and is much easier to understand. Let us consider the node `print(product);`. We observe that there is a directed path from all nodes of the graph (excluding `print(sum);`) to this node. In particular, the path from `sum := 0;` is given in Appendix A. What happens is that several actual-in nodes of different call sites can be reached from a formal-in node that was indirectly reached from an actual-out node of only one of those call sites (the others being irrelevant for instance). Yet, some of these execution paths are infeasible.

Thus, both approaches lead to unnecessarily inaccurate slices. Taking into account the calling context of a called procedure is the main difficulty in interprocedural slicing.

### 3.3.2 A solution for dependence graphs

Nevertheless, Horwitz *et al.* were aware of this problem and proposed a solution to it. An additional kind of edges is introduced to represent *transitive dependences* between actual-in and actual-out vertices. These edges are referred to as *summary edges*.

In the original paper of Horwitz *et al.*, summary edges were computed with an attribute grammar called *linkage grammar*. Later, some papers, including [RHSR94], proposed new algorithms which are more efficient.

As an example, we give in Figure 3.6 the summary edges of the first call to *add* in *main*. Thick dotted arrows represent transitive dependences. For instance, the value of *sum* after the call depends on both the value of *sum* and the value of $i$ before the call, whereas the value of $i$ after the call depends only on its value before since $i$ is not modified. The global SDG with summary edges added to every call site is given in Appendix A.

Of course, slicing does not consist of a single-pass reachability problem anymore. The computation of each slice is now performed in two phases. Both phases are performed like in the intraprocedural slicing, i.e. the graph is traversed to find the set of nodes that can reach the nodes of the criterion. Yet, the two traversals do not follow the same edges. In the

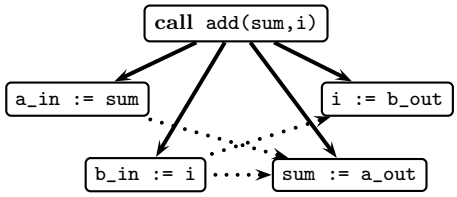



Figure 3.6: Example of summary edges for the call site `add(sum,i);`

first phase, all edges except for parameter-out edges are followed. Hence, the traversal does not descend into called procedures. In opposition, during the second phase, all edges except for parameter-in edges are followed. The traversal does not ascend into calling procedures. The calling context of each procedure call site is thus respected.

# Chapter 4

# Slicing using inference rules

## 4.1 Motivation

In the previous chapter, we explained that we were interested, for a refactoring tool, in static backward interprocedural slices. We then described two approaches of this kind of program slicing. One, introduced by Weiser, is based on data flow equations. The other one, more famous and used in engineering tasks, is based on a dependence graph called SDG. We outlined as well the issue of context sensitivity in interprocedural slicing. Weiser's approach is inaccurate because it is not context sensitive. On the other hand, the SDG-based approach can be improved to overcome the problem by means of summary edges.

We shall recall that slicing interprocedurally is necessary if we want refactoring tools, based on program slicing, to be really helpful to developers. Indeed, in modern code design, procedures are often very short (and consequently call sites very numerous) in order to obtain better readability, reusability, and structural decomposition. Hence, if we sliced intraprocedurally and handled call sites in a conservative approximative way, the benefits of program slicing for refactoring would be very small. We need to compute accurate interprocedural slices.

Thus, Weiser's approach is not appropriate for refactoring, and we may focus on the SDG-based approach. Due to the various kinds of dependences (including the summary edges that are needed to account for the calling context problem), the SDG of a program is complex to build. Much time and memory are required to compute and store the dependence graph. Moreover, all the program has to be analysed and represented in the SDG. Yet, once it is built, many slices of the same program, at different point of interest, can be computed in a linear time as a graph reachability problem.

This ability of quickly retrieving many slices after an expensive pre-computation is certainly useful in many software engineering tasks such as debugging or program understand-

ing but appears to be inappropriate in the case of refactoring. Indeed, refactoring is intended to be used during the development process. The program to be sliced is continually modified either directly by the developer or indirectly through the use of refactoring tools. Building the SDG each time the developer wishes to perform a refactoring would be very expensive. We could use incremental building but maintaining a SDG representation of the program is still expensive because changes in the program are not always atomic when using refactoring tools. Besides, the SDG of a program does not handle explicitly features such as jump statements or object-oriented concepts. Handling such features requires computing more dependences and adding new edges to the SDG, which results in an even more complex graph.

These are the reasons why we propose a new approach based on inference rules. Basic rules are given to account for the program flow. Dynamic interprocedural rules are dedicated to each procedure that might be relevant. They are context sensitive, computed on demand (when reaching a call site) and then cached for eventual reuse (at other call sites). Moreover, the system of rules is easily extended to support more features.

## 4.2 Definition of a WHILE language

In order to simplify the description of the inference rules algorithm, we shall consider a simple imperative language called WHILE. This language supports sequences of three kinds of statements: *assignment*, *if* and *while* statements. We shall use the following sets and syntactic conventions:

| | | | |
|---|---|---|---|
| $x \in \mathbf{Var}$ | variables | $n \in \mathbf{Num}$ | numerals |
| $\ell \in \mathbf{Lab}$ | labels | $op_a \in \mathbf{Op_a}$ | arithmetic operators |
| $S \in \mathbf{Stmt}$ | statements | $op_r \in \mathbf{Op_r}$ | relational operators |
| $a \in \mathbf{AExp}$ | arithmetic expressions | $op_b^b \in \mathbf{Op_b^b}$ | binary boolean operators |
| $b \in \mathbf{BExp}$ | boolean expressions | $op_b^u \in \mathbf{Op_b^u}$ | unary boolean operators |

In the sequel, we will often use $e \in \mathbf{Exp}$ to denote all kinds of expressions, either arithmetic or boolean. The abstract syntax of the language is:

$$a ::= x \mid n \mid a_1\ op_a\ a_2$$
$$b ::= \mathbf{true} \mid \mathbf{false} \mid op_b^u\ b \mid b_1\ op_b^b\ b_2 \mid a_1\ op_r\ a_2$$
$$S ::= \epsilon \mid [x := a]^\ell \mid \mathbf{if}\ [b]^\ell\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \mid \mathbf{while}\ [b]^\ell\ \mathbf{do}\ S_1 \mid S_1; S_2$$

We consider that each statement is labelled, either directly like $[x := a]^\ell$ or indirectly through the label given to the control predicate $[b]^\ell$. We assume that distinct statements have distinct labels. Furthermore, $\epsilon$ denotes the empty statement which verifies $\epsilon; S = S; \epsilon = S$ and brackets will be used when needed to resolve ambiguities.

As an example, we can express part of the program of Figure 1.2 in the WHILE language. Declarations of variables and calls to procedures (such as *read* or *print*) are ignored.

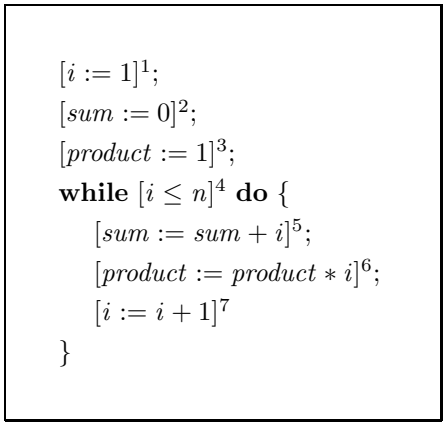



Figure 4.1: The example of Figure 1.2 in WHILE.

Moreover, we introduce the function $Ref_{Exp} : \mathbf{Exp} \rightarrow \mathbb{P}\,\mathbf{Var}$ which gives the set of variables that are used for the evaluation of a given arithmetic or boolean expression. For example, $Ref_{Exp}((x + y) < 10) = \{x, y\}$. In fact a label of a statement refers to its node in the CFG. Hence, using the notations introduced in 2.3, for any statement $[x := a]^\ell$, $Def(\ell) = \{x\}$ and $Ref(\ell) = Ref_{Exp}(a)$. Similarly, for any statement $[b]^\ell$, $Def(\ell) = \varnothing$ and $Ref(\ell) = Ref_{Exp}(b)$.

## 4.3 The inference rules approach

The concept of our approach is to use inference rules when traversing backwards the flow of the program, in order to determine which statements are relevant.

Inference rules are applied on a specific *configuration*, i.e. a triplet $< S, \Gamma, \mathcal{R} >$ where:

- $S$ is the statement or sequence of statements that have been analysed ;
- $\Gamma$ is the current slicing *context* that we introduce below ;
- $\mathcal{R}$ is the set of statements that are relevant so far.

### 4.3.1 The slicing context

A slicing context of a configuration handles information about the variables which were used in the statements that have been analysed so far. In a context $\Gamma$, a variable $x$ is either *referenced* or *modified*. It is referenced if, when looking forwards at the statements which have already been analysed, $x$ *may* be referenced before being modified. By contrast, it is

modified if, when looking at the analysed statements, $x$ *is* modified in any possible flows before being referenced.

Let us use the $Z$ notation [WD96] to formally define those notions. The set of possible states is:

$$\mathbf{State} ::= ref \mid mod$$

A context is a partial function which maps some variables to particular states:

$$\Gamma \in \mathbf{Context} = \mathbf{Var} \pfun \mathbf{State}$$

We need an operator for *overriding* a context with another one:

$$\_ \oplus \_ : \mathbf{Context} \times \mathbf{Context} \longrightarrow \mathbf{Context}$$

$$\forall A, B : \mathbf{Context} \bullet A \oplus B = (\dom B \ndres A) \cup B$$

In addition to overriding, we define a *merging* operator both for states and contexts. Merging two states yields the 'strongest' one where *ref* is 'stronger' than *mod*. Indeed, if a variable $x$ is referenced first in a flow and modified first in another flow, looking from the point where the two flows split, $x$ *may* be referenced first:

$$\_ \uplus_{State} \_ : \mathbf{State} \times \mathbf{State} \longrightarrow \mathbf{State}$$

$$\forall s : \mathbf{State} \bullet ref \uplus_{State} s = ref$$
$$mod \uplus_{State} ref = ref$$
$$mod \uplus_{State} mod = mod$$

Merging two contexts consists in merging the states of the common variables and keeping the other non-ambiguous mappings:

$$\_ \uplus \_ : \mathbf{Context} \times \mathbf{Context} \longrightarrow \mathbf{Context}$$

$$\forall A, B : \mathbf{Context} \bullet$$
$$A \uplus B = (\dom B \ndres A) \cup (\dom A \ndres B)$$
$$\cup \{v : \mathbf{Var} \mid v \in (\dom A \cap \dom B) \bullet$$
$$v \mapsto A\ v \uplus_{State} B\ v\}$$

### 4.3.2 Basic inference rules

We may now describe the inference rules. Assuming an initial configuration where no statement has been analysed, the aim is to reach a configuration where all statements to be sliced have been analysed. To achieve this, we apply some inference rules to a current configuration which is updated iteratively. A rule has the following structure:

$$\frac{premisses}{resulting\ configuration}\ [name] \quad side\ condition$$

It is applicable if and only if the premisses hold and the side condition is true. We may assume temporarily particular configurations and use them to reach specific premisses of a rule. In such cases, the rule is labeled $[name^{[i]}]$ and $\lceil < S, \Gamma, \mathcal{R} > \rceil^{[i]}$ denotes a related assumption, which is valid in the scope represented by the vertical ellipsis and discharged when the rule is applied. We have at our disposal *relevant* rules where the currently analysed statement is kept as relevant and *irrelevant* rules where, in opposition, the statement is not kept as relevant. Names of relevant rules are preceded with '$R-$' and names of non relevant ones with '$NR-$'. Moreover, to clarify the writing, we introduce a new function $\gamma_{ref} : \mathbf{Exp} \rightarrow \mathbf{Context}$ which gives the minimal context where all the variables of a given expression are mapped to *ref*:

$$\gamma_{ref}(e) = \{x : Ref_{Exp}(e) \cdot x \mapsto ref\}$$

For an *assignment* statement, we have two rules. Either the assigned variable is referenced in the current context, in which case the statement is relevant and the context is overridden with the local changes, or the assigned variable is not referenced in the context, in which case the statement is not relevant. The assigned variable might be referenced on the right-side expression of the assignment as well. Hence, if the assignment is relevant, the order in which we override the context is important and really reveals that we slice backwards.

$$\frac{< S, \Gamma, \mathcal{R} >}{< [x := a]^{\ell}; S, \Gamma \oplus \{x \mapsto mod\} \oplus \gamma_{ref}(a), \mathcal{R} \cup \{\ell\} >} \; [R - assign] \quad \Gamma(x) = ref$$

$$\frac{< S, \Gamma, \mathcal{R} >}{< [x := a]^{\ell}; S, \Gamma, \mathcal{R} >} \; [NR - assign] \quad x \notin \Gamma \rhd \{ref\}$$

Concerning the *if* statement, it is relevant if analysing the *then* branch $S_1$ and the *else* branch $S_2$, a statement appears to be relevant. In such a case, the two contexts of the two flows are merged and the local changes due to the control predicate are added to the resulting context. Of course, the set of relevant statements is updated as well. On the other hand, if no statement is relevant in the branches, the context and the set of relevant statements remain the same.

$$\frac{\begin{array}{ccc} & \lceil < \epsilon, \Gamma, \varnothing > \rceil^{[i]} \quad \lceil < \epsilon, \Gamma, \varnothing > \rceil^{[i]} & \\ & \vdots \qquad\qquad \vdots & \\ < S, \Gamma, \mathcal{R} > & < S_1, \Gamma_1, \mathcal{R}_1 > \quad < S_2, \Gamma_2, \mathcal{R}_2 > & \end{array}}{< \textbf{if } [b]^{\ell} \textbf{ then } S_1 \textbf{ else } S_2; S, (\Gamma_1 \uplus \Gamma_2) \oplus \gamma_{ref}(b), \mathcal{R} \cup \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{\ell\} >} \; [R - if^{[i]}] \quad \mathcal{R}_1 \cup \mathcal{R}_2 \neq \varnothing$$

$$\frac{\begin{array}{ccc} & \lceil < \epsilon, \Gamma, \varnothing > \rceil^{[i]} \quad \lceil < \epsilon, \Gamma, \varnothing > \rceil^{[i]} & \\ & \vdots \qquad\qquad \vdots & \\ < S, \Gamma, \mathcal{R} > & < S_1, \Gamma_1, \mathcal{R}_1 > \quad < S_2, \Gamma_2, \mathcal{R}_2 > & \end{array}}{< \textbf{if } [b]^{\ell} \textbf{ then } S_1 \textbf{ else } S_2; S, \Gamma, \mathcal{R} >} \; [NR - if^{[i]}] \quad \mathcal{R}_1 \cup \mathcal{R}_2 = \varnothing$$

At last, analysing the *while* statement is slightly more complex. It requires to look for a fixed point if the first analysis of the loop body reveals some relevant statements. A fixed point is reached when the entry context is equal to the exit context after having analysed the loop body plus the control predicate. For more clarity, we give two relevant rules in such a case. The first one is applicable if the fixed point is reached after the first iteration. The second one is for general cases. In both of them, the different flows of the program are taken into account when merging the initial context and the context resulting of the iterations. Besides, a third rule is given if the loop body is irrelevant.

$$\frac{\begin{array}{ccc} & \lceil < \epsilon, \Gamma, \varnothing > \rceil^{[i]} & \\ & \vdots & \\ < S, \Gamma, \mathcal{R} > & < S_1, \Gamma_1, \mathcal{R}_1 > & \end{array}}{< \textbf{while } [b]^{\ell} \textbf{ do } S_1; S, (\Gamma \uplus \Gamma_1) \oplus \gamma_{ref}(b), \mathcal{R} \cup \mathcal{R}_1 \cup \{\ell\} >} \; [R - while1^{[i]}] \quad \begin{array}{c} \mathcal{R}_1 \neq \varnothing \\ \Gamma = \Gamma_1 \oplus \gamma_{ref}(b) \end{array}$$

$$\frac{\begin{array}{ccccc} & & & \lceil < \epsilon, \Gamma_{n-1} \oplus \gamma_{ref}(b), \mathcal{R}_{n-1} \cup \{\ell\} > \rceil^{[i]} & \\ & & & \vdots & \\ & & & < S_1, \Gamma_n, \mathcal{R}_n > & \\ & & \lceil < \epsilon, \Gamma_1 \oplus \gamma_{ref}(b), \mathcal{R}_1 \cup \{\ell\} > \rceil^{[i]} & \vdots & \\ & & \vdots & & \\ & & < S_1, \Gamma_2, \mathcal{R}_2 > & & \\ & \lceil < \epsilon, \Gamma, \varnothing > \rceil^{[i]} & \vdots & & \\ & \vdots & & & \\ < S, \Gamma, \mathcal{R} > & < S_1, \Gamma_1, \mathcal{R}_1 > & \cdots & & \end{array}}{< \textbf{while } [b]^{\ell} \textbf{ do } S_1; S, (\Gamma \uplus \Gamma_n) \oplus \gamma_{ref}(b), \mathcal{R} \cup \mathcal{R}_n \cup \{\ell\} >} \; [R - while2^{[i]}] \quad \begin{array}{c} n \geq 2 \\ \mathcal{R}_1 \neq \varnothing \\ \Gamma_{n-1} \oplus \gamma_{ref}(b) = \Gamma_n \end{array}$$

$$\dfrac{\begin{array}{c}\lceil < \epsilon, \Gamma, \varnothing > \rceil^{[i]} \\ \vdots \\ < S, \Gamma, \mathcal{R} > \quad < S_1, \Gamma_1, \mathcal{R}_1 >\end{array}}{< \textbf{while } [b]^{\ell} \textbf{ do } S_1; S, \Gamma, \mathcal{R} >} \; [NR - while^{[i]}] \quad \mathcal{R}_1 = \varnothing$$

### 4.3.3 Slicing with a derivation tree

Let the slicing criterion be to slice from the last statement of a program $\mathcal{P}$ with respect to a set of variables $V$. Deriving the slice consists in reaching a configuration $< \mathcal{P}, \Gamma_{final}, \mathcal{R}_{final} >$ from the initial configuration $< \epsilon, \{x : V \cdot x \mapsto ref\}, \varnothing >$. As the inference rules are defined so that, at each step, one (and at most one) of them matches the configuration, there is always a unique derivation tree and, a fortiori, a unique final set of relevant statements $\mathcal{R}_{final}$.

As an example, we shall describe how the rules are applied to the simple program given in Figure 4.1. Let us slice from the end of the program for the variable *product*. The initial configuration is $< \epsilon, \{product \mapsto ref\}, \varnothing >$. and the unique derivation tree is shown in Figure 4.2. To make it short, statements are abbreviated with their label: 1; 2; 3; **while** 4 **do** {5; 6; 7} denotes the whole program.

$$\dfrac{\dfrac{\dfrac{\dfrac{\lceil < \epsilon, \{product \mapsto ref, i \mapsto ref, n \mapsto ref\}, \{6,4\} > \rceil^{[1]}}{< 7, \{product \mapsto ref, i \mapsto ref, n \mapsto ref\}, \{6,4,7\} >}\,[R-assign]}{< 6; 7, \{product \mapsto ref, i \mapsto ref, n \mapsto ref\}, \{6,4,7\} >}\,[R-assign]}{< 5; 6; 7, \{product \mapsto ref, i \mapsto ref, n \mapsto ref\}, \{6,4,7\} >}\,[NR-assign]}$$

$$\dfrac{\dfrac{\dfrac{\lceil < \epsilon, \{product \mapsto ref\}, \varnothing > \rceil^{[1]}}{< 7, \{product \mapsto ref\}, \varnothing >}\,[NR-assign]}{< 6; 7, \{product \mapsto ref, i \mapsto ref\}, \{6\} >}\,[R-assign]}{< 5; 6; 7, \{product \mapsto ref, i \mapsto ref\}, \{6\} >}\,[NR-assign]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{< \epsilon, \{product \mapsto ref\}, \varnothing >} \quad \dfrac{}{< \textbf{while } 4 \textbf{ do } \{5;6;7\}, \{product \mapsto ref, i \mapsto ref, n \mapsto ref\}, \{6,4,7\} >}}{< \textbf{while } 4 \textbf{ do } \{5;6;7\}, \{product \mapsto ref, i \mapsto ref, n \mapsto ref\}, \{6,4,7\} >}\,[R-while2^{[1]}]}{< 3; \textbf{while } 4 \textbf{ do } \{5;6;7\}, \{product \mapsto mod, i \mapsto ref, n \mapsto ref\}, \{6,4,7,3\} >}\,[R-assign]}{< 2; 3; \textbf{while } 4 \textbf{ do } \{5;6;7\}, \{product \mapsto mod, i \mapsto ref, n \mapsto ref\}, \{6,4,7,3\} >}\,[NR-assign]}{< 1; 2; 3; \textbf{while } 4 \textbf{ do } \{5;6;7\}, \{product \mapsto mod, i \mapsto mod, n \mapsto ref\}, \{6,4,7,3,1\} >}\,[R-assign]$$

Figure 4.2: The derivation tree to slice the program of Figure 4.1 with respect to *product*.

The relevant statements consists of the final set $\{1, 3, 4, 6, 7\}$. As expected, the state-

ments where *sum* is computed are not relevant. We said before that the control statements were labelled indirectly through their control predicate. Therefore, marking the control predicate $[i \leq n]^4$ as relevant is equivalent to marking the whole control statement structure **while** $[i \leq n]^4$ **do** $\cdots$. In an *if* statement, it may be possible that one of the two branches contains no relevant statement. In such a case, the control statement should be rewritten in order to omit the irrelevant branch. Those transformations are straightforward and dependent on the language you want to support. In fact, we will see in 4.5.1 that, in modern languages, a control predicate can be relevant by itself even if there is no relevant statement in the possible branches. To preserve the syntactic correctness of the slice, the control statement needs to be kept anyway.

We have assumed so far that we were slicing from the end of the program. Slicing from any point of interest, including from inside a body loop, requires a slight adaptation that we discuss in 4.5.5.

## 4.4 Slicing interprocedurally

Interprocedural slicing is required to achieve a good accuracy of slices. We give, in this section, inference rules to support interprocedural context-sensitive slicing.

### 4.4.1 Extension of the While language

We need first to extend our definition of the While language to consider a program as a sequence of procedures $p \in \mathbf{Proc}$. Sequences $param, arg \in \mathbf{Param}$ of variables are used to denote the parameters (or arguments with respect to a call site), passed by value-result. The abstract syntax of this extension is:

$$P ::= \mathbf{proc}\ [p]^\ell(param)\ \mathbf{begin}\ S\ \mathbf{end}\ \mid P_1; P_2$$
$$param ::= x \mid param_1; param_2$$

In addition, to support calls to procedures, the syntax of statements is extended with:

$$S ::= \cdots \mid [\mathbf{call}\ p(arg)]^\ell$$

Since global variables can be simulated with some additional arguments for each procedure, we assume variables to be only local variables or parameters. Let us now transform the While example of Figure 4.1 in a multiprocedural program, like we did in Chapter 3. For simplicity, we still ignore the calls to procedures *read* and *print*. If we wanted to include them, we would only need to define the procedure *read* with one parameter set to some value of an input stream, and the procedure *write* with one parameter used to modify an output stream. For the reasons we gave in section 3.2.1, the output stream should be mapped to *ref* so that the parameter of *write* appears to be relevant in every case. However, in the

multiprocedural program supporting the extended syntax, $n$ is passed as a parameter to the main procedure.

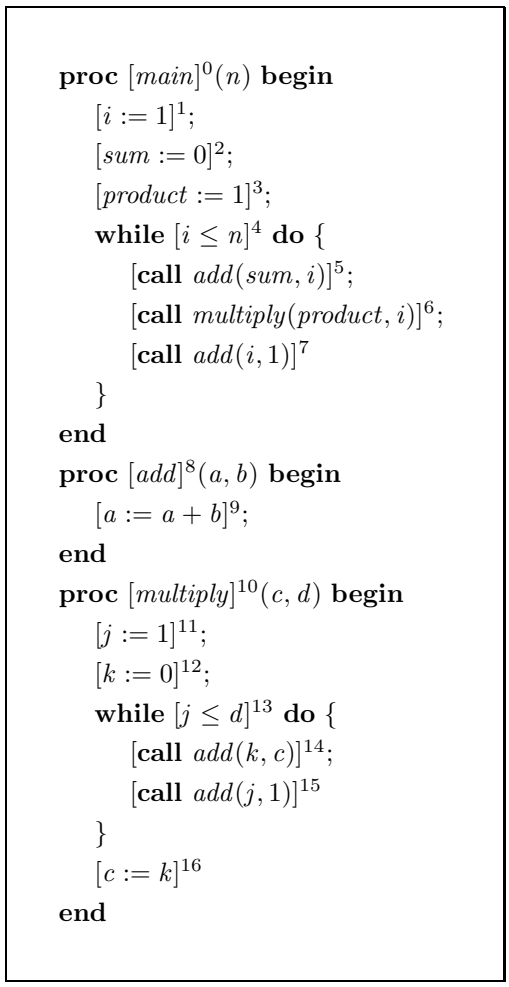



Figure 4.3: The example of Figure 1.2 in WHILE.

We introduce the function $Def_{Proc} : \mathbf{Proc} \rightarrow \mathbb{P}\,\mathbf{Var}$ which gives the set of non-local variables (i.e. the parameters since we do not consider any global variables) that may be modified in a given procedure. For instance, in the above example, $Def_{Proc}(add) = \{a\}$ and $Def_{Proc}(multiply) = \{c\}$.

Besides, we use the usual Z notations for sequences: $\sharp param$ denotes the numbers of parameters of $param$ and $param(i)$ denotes the $i^{th}$ parameter of $param$.

### 4.4.2 Interprocedural rules

We give now the following rules to introduce the declaration of a procedure in the statements that have been analysed. If some statements in the body of the procedure are relevant then the declaration of the procedure itself is relevant. Otherwise, it is not.

$$\frac{< S, \Gamma, \mathcal{R} >}{< \textbf{proc } [p]^{\ell}(param) \textbf{ begin } S \textbf{ end}, \Gamma, \mathcal{R} \cup \{\ell\} >} \; [R - proc] \quad \mathcal{R} \neq \varnothing$$

$$\frac{< S, \Gamma, \mathcal{R} >}{< \textbf{proc } [p]^{\ell}(param) \textbf{ begin } S \textbf{ end}, \Gamma, \mathcal{R} >} \; [NR - proc] \quad \mathcal{R} = \varnothing$$

We do not specify all the conditions on which these rules are applicable since they are intended to be used with two more rules, that we shall define now, to account for a call site. We need a function to adapt and restrict a context before and after slicing a specific procedure like in Weiser's approach. We first recall that, as we slice backwards, the entry and exit contexts of a procedure are respectively the contexts at the end and at the beginning of a procedure.

Thus, the entry context of the procedure to be sliced is the calling context where arguments of the call have been changed to the parameters of the called procedure and where any other variables that are not used in the scope of the called procedure have been removed. Conversely, the context in the calling procedure after having analysed the call site is the exit context of the sliced procedure where parameters have been changed back to the arguments and where local variables have been removed.

Therefore, we introduce the function $\delta$ : $\textbf{Var} \times \textbf{Var} \times \textbf{Context} \rightarrow \textbf{Context}$ which maps a sequence of variables to another in a given context and restricts this context to the variables present in the latter sequence. The two sequences must have the same length. The formal definition is:

$$\delta\_\rightarrow\_(\_) : \textbf{Var} \times \textbf{Var} \times \textbf{Context} \longrightarrow \textbf{Context}$$

$$\forall\, p, q : \textbf{Param}; \Gamma : \textbf{Context} \mid \sharp p = \sharp q \bullet$$
$$\delta_{p \rightarrow q}(\Gamma) = \{i : 1..\sharp\, p \mid p(i) \in \operatorname{dom} \Gamma \bullet q(i) \mapsto \Gamma p(i)\}$$

For instance, $\delta_{a,b,c \rightarrow x,y,z}(\{a \mapsto ref, b \mapsto mod, d \mapsto ref\}) = \{x \mapsto ref, y \mapsto mod\}$.

We shall make two remarks. Firstly, the entry context of a procedure to be sliced does not need to include variables that are mapped to *mod*. Indeed, when a variable is modified,

no statement analysed afterwards will be relevant because of this modification. In fact, concerning the whole algorithm, we may wonder why we map a variable to *mod* instead of removing its mapping to *ref* from the context. We will discuss this question in Subsection 4.5.3. Secondly, the entry context of a procedure does not need to include variables that can not be modified in the procedure. These variables are initially useless since they will no be required for any rules while slicing the procedure. We will discuss more about the modified variables of a procedure in Subsection 4.4.4.

$$\frac{\begin{array}{c}\lceil < \epsilon, Def_{proc}(p) \lhd \delta_{arg \to param}(\Gamma) \rhd \{ref\}, \varnothing > \rceil^{[i]} \\ \vdots \\ < S, \Gamma, \mathcal{R} > \quad < \textbf{proc } [p]^m(param) \textbf{ begin } S_{proc} \textbf{ end}, \Gamma_{exit}, \mathcal{R}_{exit} >\end{array}}{< [\textbf{call } p(arg)]^{\ell}; S, \Gamma \oplus \delta_{param \to arg}(\Gamma_{exit}), \mathcal{R} \cup \mathcal{R}_{exit} \cup \{\ell\} >} \; [R - call^{[i]}] \quad \mathcal{R}_{exit} \neq \varnothing$$

$$\frac{\begin{array}{c}\lceil < \epsilon, Def_{proc}(p) \lhd \delta_{arg \to param}(\Gamma) \rhd \{ref\}, \varnothing > \rceil^{[i]} \\ \vdots \\ < S, \Gamma, \mathcal{R} > \quad < \textbf{proc } [p]^m(param) \textbf{ begin } S_{proc} \textbf{ end}, \Gamma_{exit}, \mathcal{R}_{exit} >\end{array}}{< [\textbf{call } p(arg)]^{\ell}; S, \Gamma, \mathcal{R} >} \; [NR - call^{[i]}] \quad \mathcal{R}_{exit} = \varnothing$$

To experiment with these new rules, let us slice the WHILE multiprocedural sample program from the end of the procedure *main* for the variable *product*, by building the derivation tree.

For more clarity, we split the entire tree in four subtrees. $\lceil \textbf{addNR} \rceil^{[i]}$ represents the tree to slice the procedure *add* when assuming that the entry context is empty. Hence, the procedure is of course not relevant.

$$\frac{\dfrac{\lceil < \epsilon, \varnothing, \varnothing > \rceil^{[i]}}{< 9, \varnothing, \varnothing >} \; [NR - assign]}{< \textbf{proc } [add]^8(a, b) \textbf{ begin } 9 \textbf{ end}, \varnothing, \varnothing >} \; [NR - proc]$$

Figure 4.4: $\lceil \textbf{addNR} \rceil^{[i]}$

$\lceil \textbf{addR} \rceil^{[i]}$ is the tree for slicing *add* when assuming the entry context consists of $\{a \mapsto ref\}$. Besides, for convenience, we write the mappings of the context in a shorter way, by

regrouping the variables that are mapped to the same state. For instance, $\{(a, b) \mapsto ref\}$ is equivalent to $\{a \mapsto ref, b \mapsto ref\}$.

$$\dfrac{\dfrac{\lceil < \epsilon, \{a \mapsto ref\}, \varnothing > \rceil^{[i]}}{< 9, \{(a, b) \mapsto ref\}, \{9\} >}\ [R - assign]}{< \textbf{proc}\ [add]^8(a, b)\ \textbf{begin}\ 9\ \textbf{end}, \{(a, b) \mapsto ref\}, \{9, 8\} >}\ [R - proc]$$

Figure 4.5: $\lceil \textbf{addR} \rceil^{[i]}$

Next, $\lceil \textbf{multiplyR} \rceil^{[i]}$ is the tree to slice *multiply* with the assumption that the entry context is restricted to $\{c \mapsto ref\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\lceil < \epsilon, \{c \mapsto mod, k \mapsto ref\}, \varnothing > \rceil^{[1]} \quad \lceil \textbf{addNR} \rceil^{[2]}}{< 15, \{\epsilon, \{c \mapsto mod, k \mapsto ref\}, \varnothing >}\ [NR - call^{[2]}] \quad \lceil \textbf{addR} \rceil^{[3]}}{< 14; 15, \{(k, c) \mapsto ref\}, \{14, 8, 9\} >}\ [R - call^{[3]}] \quad \dfrac{\lceil < \epsilon, \{c \mapsto ref\}, \varnothing > \rceil^{[i]}}{< 16, \{c \mapsto mod, k \mapsto ref\}, \{16\} >}\ [R - assign]}{< \textbf{while}\ 13\ \textbf{do}\ \{14; 15\}; 16, \{(k, c, j, d) \mapsto ref\}, \{16, 14, 8, 9, 15, 13\} >}\ [R - while2^{[1]}]}{< 12; \textbf{while}\ 13\ \textbf{do}\ \{14; 15\}; 16, \{(c, j, d) \mapsto ref, k \mapsto mod\}, \{16, 14, 8, 9, 13, 15, 12\} >}\ [R - assign]}{< 11; 12; \textbf{while}\ 13\ \textbf{do}\ \{14; 15\}; 16, \{(c, d) \mapsto ref, (k, j) \mapsto mod\}, \{16, 14, 8, 9, 13, 15, 12, 11\} >}\ [R - assign]}{< \textbf{proc}\ [multiply]^{10}(c, d)\ \textbf{begin}\ \cdots\ \textbf{end}, \{(c, d) \mapsto ref, (k, j) \mapsto mod\}, \{8..16\} >}\ [R - proc]$$

$$\dfrac{\dfrac{\lceil < \epsilon, \{(k, c, j, d) \mapsto ref\}, \{14, 8, 9, 13\} > \rceil^{[1]} \quad \lceil \textbf{addR} \rceil^{[4]}}{< 15, \{\epsilon, \{(k, c, j, d) \mapsto ref\}, \{14, 8, 9, 13, 15\} >}\ [R - call^{[4]}] \quad \lceil \textbf{addR} \rceil^{[5]}}{< 14; 15, \{(k, c, j, d) \mapsto ref\}, \{14, 8, 9, 13, 15\} >}\ [R - call^{[5]}]$$

Figure 4.6: $\lceil \textbf{multiplyR} \rceil^{[i]}$

Last, Figure 4.7 shows the tree of the slice from the end of the procedure *main* with respect to *product*. Thanks to the fact the algorithm we propose is context sensitive, the call site related to *sum* is sliced away and the definition of *sum* as well. Thus, we obtain the slice we expected.

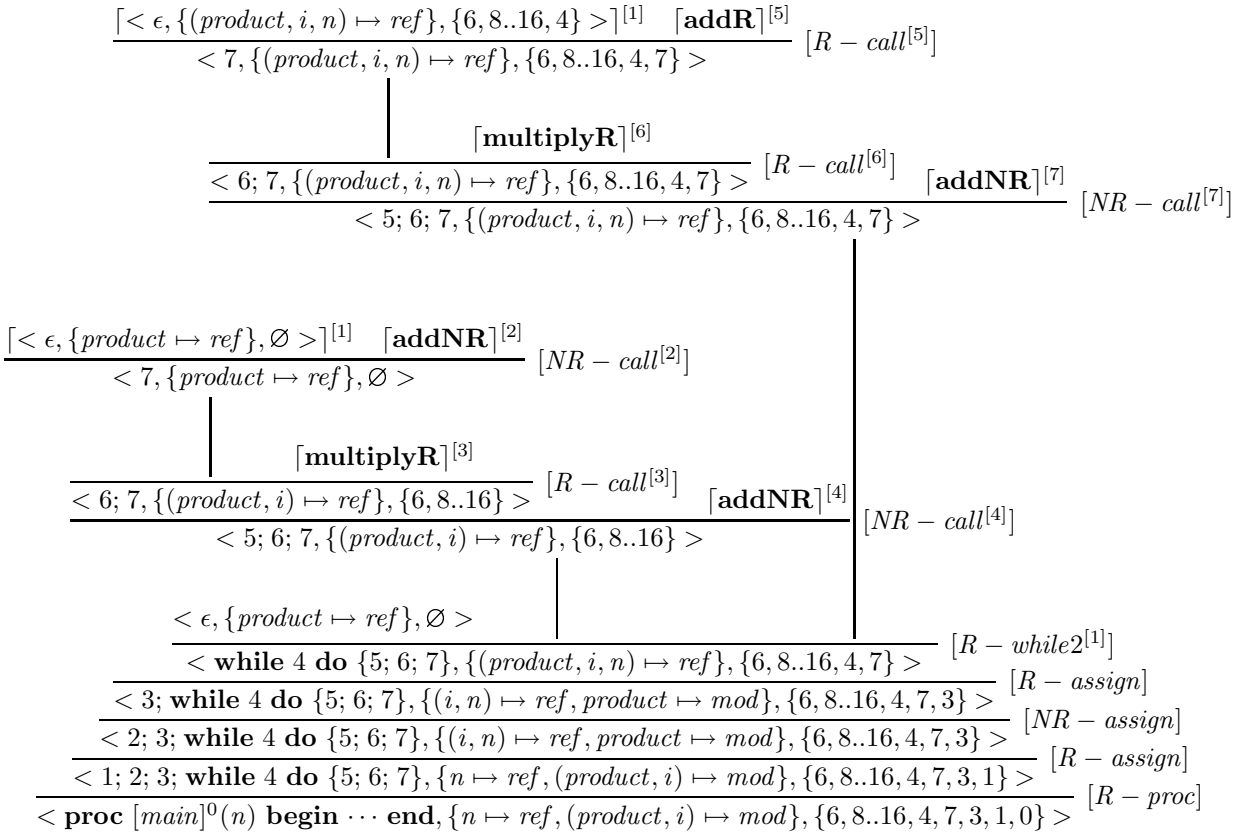



Figure 4.7: The derivation tree to slice the WHILE multiprocedural sample program from the end of the procedure *main* for the variable *product*.

### 4.4.3 Speeding up slicing

In the previous example, we may note that, at many call sites of the procedure *add*, we needed to slice *add* with respect to the same entry context. This is mostly because we restricted the entry context to what may be really useful. In fact, the call site **call** $p(arg)$ of a procedure **proc** $[p](param)$ **begin** $S$ **end** will be relevant if and only if we have for the calling context $\Gamma$:

$$Def_{proc}(p) \lhd \delta_{arg \to param}(\Gamma) \rhd \{ref\} \neq \varnothing$$

Therefore, we could obviously improve our non relevant call rule $NR - call$ to avoid computing a slice when, in opposition, the entry context is empty:

$$\frac{< S, \Gamma, \mathcal{R} >}{< [\textbf{call } p(arg)]^{\ell}; S, \Gamma, \mathcal{R} >}\ [NR - call] \qquad \begin{array}{c} Def_{proc}(p) \lhd \delta_{arg \to param}(\Gamma) \rhd \{ref\} = \varnothing \\ \text{where } param \text{ are the parameters of } p \end{array}$$

The subtree *addNR* which was given in the previous subsection is not useful anymore and we could even define a *new dedicated rule* for a call to *add* that appears not to be relevant:

$$\frac{< S, \Gamma, \mathcal{R} >}{< [\textbf{call } add(x,y)]^{\ell}; S, \Gamma, \mathcal{R} >} \; [NR - add] \quad x \notin \Gamma \rhd \{ref\}$$

Moreover, it is inefficient to compute several slices of the same procedure with respect to the same entry context. A natural and efficient solution is to compute the slice *on demand* when it is needed (i.e. when the first occurrence of a related call site has to be analysed) and to *cache* the result for a potential reuse. Hence, future occurrences of calls to the same procedure with the same entry context will be analysed straightforwardly.

Precisely, we define again a new dedicated rule for the given procedure with a side condition on the calling context. For instance, after *addR* has been computed once, we can add the following dedicated rule to our system of rules:

$$\frac{< S, \Gamma, \mathcal{R} >}{< [\textbf{call } add(x,y)]^{\ell}; S, \Gamma \oplus \{x \mapsto ref, y \mapsto ref\}), \mathcal{R} \cup \{8,9,\ell\} >} \; [R - add] \quad \Gamma(x) = ref$$

Future calls to add are then treated like any assignments, which speeds up slicing a lot.

In the case of *add* (or even *multiply*), there is only one possible dedicated relevant rule, since one parameter only is susceptible of being modified in the body of the procedure. We may wonder how to handle other cases. Figure 4.8 below shows a new procedure with two parameters that are both modified.

```
proc [swap]^0(a, b) begin
    [t := a]^1;
    [a := b]^2;
    [b := t]^3
end
```

Figure 4.8: A procedure in WHILE with two parameters being modified.

As there are three possible non-empty entry contexts, we could define three dedicated relevant rules in addition to the irrelevant one:

$$\frac{< S, \Gamma, \mathcal{R} >}{< [\textbf{call } swap(x,y)]^{\ell}; S, \Gamma, \mathcal{R} >} \; [NR - swap] \quad \begin{array}{l} x \notin \Gamma \rhd \{ref\} \\ y \notin \Gamma \rhd \{ref\} \end{array}$$

$$\frac{< S, \Gamma, \mathcal{R} >}{< [\textbf{call } swap(x,y)]^{\ell}; S, \Gamma \oplus \{x \mapsto mod, y \mapsto ref\}), \mathcal{R} \cup \{0,2,\ell\} >} [R - swap1] \quad \begin{array}{c} \Gamma(x) = ref \\ y \notin \Gamma \rhd \{ref\} \end{array}$$

$$\frac{< S, \Gamma, \mathcal{R} >}{< [\textbf{call } swap(x,y)]^{\ell}; S, \Gamma \oplus \{y \mapsto mod, x \mapsto ref\}), \mathcal{R} \cup \{0,1,3,\ell\} >} [R - swap2] \quad \begin{array}{c} \Gamma(y) = ref \\ x \notin \Gamma \rhd \{ref\} \end{array}$$

$$\frac{< S, \Gamma, \mathcal{R} >}{< [\textbf{call } swap(x,y)]^{\ell}; S, \Gamma \oplus \{x \mapsto ref, y \mapsto ref\}), \mathcal{R} \cup \{0,1,2,3,\ell\} >} [R - swap3] \quad \begin{array}{c} \Gamma(x) = ref \\ \Gamma(y) = ref \end{array}$$

In fact, building the last rule is not necessary and can be directly deduced from the two first relevant ones. In general, relevant rules where more than one parameter are in the entry context can be obtained from the rules where the entry context consists of one parameter only. Indeed, we can merge the corresponding final contexts and retrieve the union of the statements that have been relevant. This is faster than to compute a slice for another entry context. In our example:

$$\{x \mapsto ref, y \mapsto ref\} = \{x \mapsto mod, y \mapsto ref\} \uplus \{x \mapsto ref, y \mapsto mod\}$$

$$\{0,1,2,3\} = \{0,2\} \cup \{0,1,3\}$$

### 4.4.4 Building the rules

We noticed that the entry context's domain of a procedure should be restricted to the variables that are modified in this procedure and visible from the calling site, so that we get the minimal relevant context and are able to compute (and cache for reuse) the smallest amount of dedicated rules.

This means we need to retrieve, before slicing, which are the modified non-local variables of each procedure that may be sliced. With this information, we can deduce the condition on which potential dedicated rules of each procedure are applicable. When slicing, we are then able to say directly, with no additional computation, whether a call site is relevant or not for a given calling context. If it appears to be relevant, there are two cases. Either we do not have any dedicated rule to apply with respect to the calling context and we need to slice the called procedure to get one. Or, in opposition, we have already computed such a rule and we can apply it straightforwardly.

Yet, we do not want to analyse the whole program to look for the modified non-local variables of each procedure. Indeed, some of the procedures might not be relevant at all for a particular slice and it is hence useless to analyse them.

In fact, if we want to slice from the end a procedure $p$, we start by analysing $p$ looking both for the set of non-local variables *Mod* that may be modified and for the set of procedures $P_{Called}$ that may be called. To get the complete set *Mod*, we need first to analyse the procedures in $P_{Called}$. Indeed, some called procedures may modify non-local variables that are used in $p$ and hence have to be in *Mod*. Therefore, each time we find, in a procedure $p$, a call to a procedure $q$, we analyse first $q$ and then go on with the analysis of $p$. Recursive calls are problematic. We explain how they can be handled in the next subsection. All this analysis can be done forwards, since we do no take into account any flow of the program.

At the same time, we may gather some more information like the number of call sites of a specific procedure. What for? Let us consider again the *swap* procedure example of the previous subsection. It might be the case that only the third relevant rule ($R - swap3$) is needed during the whole slicing process. Then, it is faster to compute the slice with respect to entry context $\{x \mapsto ref, y \mapsto ref\}$ instead of computing two slices for both $\{x \mapsto mod, y \mapsto ref\}$ and $\{y \mapsto mod, x \mapsto ref\}$ and combining the results, like we proposed. Of course, it is hard to know while slicing if we are in such a case or not. Nevertheless, if we are aware that we have to analyse a single call to *swap* thanks to some summary information, and if we realize that the entry context consists of more than one variables, it is better to slice directly for this bigger entry context than to slice several times with the hope to reuse these additional computations later.

In fact, this choice is part of a bigger concern of our algorithm: reaching a good compromise between speed and memory consumption for caching the rules. We believe that the strategy to use really depends on the programs to be sliced and that these optimizations concern more the implementation than the algorithm itself.

We may notice that we did not differentiate slicing up to calling procedures or down to called procedures so far. Although the rules mechanism is similar in both cases, slicing up requires us to be aware of all the call sites of each procedure. Hence, in addition to procedures that can be reached from the point of interest of the slicing criterion, we need to analyse procedures that can reach it. A call graph of the program is thus required but it does not have to be complete if we take into account scoping concepts in object-oriented programming for instance.

### 4.4.5 Recursive calls

Recursive calls pose two problems. Firstly, it is more delicate to find the modified non-local variables of the procedures. If a procedure $p$ calls itself, there is no problem obviously. Yet, let us assume that $p$ calls another procedure $q$ that calls back $p$. In this case, to obtain the modified variables of $p$, we need the set of modified variables of $q$ and vice versa. The solution is to use a *chaotic iteration* algorithm [NNH99]. The set of modified variables of $p$ and $q$ are complete when a fixed point is reached. For instance, we start analysing $p$, find

a call to $q$. So, we start analysing $q$, find a call site to $p$ which is currently being analysed. We report the current modified variables of $p$ to $q$ and mark $q$ as dependent on $p$. We go on and finish analysing $q$. Then we report the current modified variables of $q$ to $p$ and mark $p$ as dependent on $q$. We finish analysing $p$, and we report to $q$, which is dependent on $p$, the eventual additional relevant variables. If there are some, we report to $p$, which is dependent on $q$, the changes in $q$ and observe that a fixed point is reached. The algorithm can be generalized to a recursion with any number of intervening procedures.

Secondly and for a similar reason, we can not compute definitive rules for such recursive calls. We use a chaotic iteration algorithm as well but in a slightly more complex way. For a given entry context, we start slicing from the end of $p$. When we reach the call to $q$, we start slicing from the end of $q$ with respect to the entry context computed from the calling context we had when $q$ was called. We then reach the call to $p$ and start slicing it probably with a different entry context from before. We repeat this until we reach a fixed point, i.e. until we realize we are about to slice one of the procedure for the same entry context twice. So, we get a temporary rule for this procedure, say $q$. We can finish slicing $p$ and obtain a temporary rule that we use to slice again the end of $q$ and so on until we obtain the definitive rule for $p$. Once again, this can be generalized to any recursion with multiple call sites of the same procedure as well.

## 4.5 More features

Slicing using inference rules is just another approach, and can certainly be expressed as a well-known dataflow analysis algorithm. Showing this similarity and proving the correctness of this approach are beyond the scope of this thesis and we hope to address these issues in future work. Nevertheless, we have shown that the system rules could be extensible to support context-sensitive interprocedural slicing. More generally, a strength of the inference rules approach is that it is easily and efficiently extensible to support more language features. We describe in this section some important ones and explain briefly how to support them.

### 4.5.1 Complex expressions

For simplicity, we have considered in the WHILE language simple expressions where variables could be referenced only. A variable was modified if and only if it was on the left side of an assignment which was considered as a statement. Yet, in modern languages, assignments are just like other expressions which are evaluated to the value of the left-side variable. Therefore you may have in Java `x = y = z;` which results in `x` and `y` having the same value as `z`. Modern languages also introduce prefix and postfix operators that allow modifying a variable before or after evaluating it. For example, `y = x++;` is equivalent to `y = x; x = x + 1;`, whereas `y = ++x;` is equivalent to `x = x + 1; y = x;`.

Hence, variables may be modified at any evaluation of an expression, including in a control predicate (e.g. **if** `(x++ == (y = 4)) ...;` **else** `...;`). Weiser dataflow equations based solution did not handle this explicitly, whereas our slicing rules can be adapted straightforwardly.

We need to substitute the function $\gamma_{ref} : \textbf{Exp} \rightarrow \textbf{Context}$ with a more general function $\gamma : \textbf{Exp} \rightarrow \textbf{Context}$ which gives the *local* context of a given expression, i.e. not only the referenced variables but also the variables that are modified. We must remember that we slice backwards and care about the order with which this local context is computed. For instance:

$$\gamma(\texttt{x++ == (y = 4)}) = \{x \mapsto ref, y \mapsto mod\}$$

Therefore, a control predicate can now be relevant by itself even if there is no relevant statement in its possible branches. So, we need to modify the previous irrelevant rules of the control statements **if** and **while**. These are now applicable only with an additional side condition saying that there is no variable that is both modified in the control predicate and referenced in the current context. Of course, we introduce new rules where the branches are still irrelevant but with an additional opposite side condition saying that there is a variable that is both modified in the control predicate and referenced in the current context. In such a case, we update the context with the local context of the predicate, and mark the control statement as relevant, even if none of its branches is relevant.

Last but not least, an expression may include a call to function. Functions can be considered as procedures with an additional parameter for the returned value. Hence, return statements must be transformed to an assignment of the returned value to this specific parameter before exiting the function. Then, if a function appears in an expression, we can specify that the parameter corresponding to the returned value is referenced and slice the function like we explained in the previous section.

We can think of another solution, which is not based on any additional parameters and transformations. Indeed, we could retrieve the set of returned variables while computing the summary information of a function. Then, when we evaluate the complex expression that calls a function, we include the returned variables in the referenced variables of the expression. Yet, we need also to compute a slice of the function with respect to the returned variables and include the result in the calling context. This could consist of a new dedicated rule.

### 4.5.2 Array accesses

We need to handle accesses to arrays both for reference in the evaluation of an expression and for modification in an assignment. The following considerations are general and would be

valid for other approaches. Let us consider the following piece of code: `a[i] = 1; r = a[j];`. As we do not slice dynamically, we do not have any information on the value of `i` or `j`. Thus, we need to be conservative although we may obtain bigger slices than expected.

A read access to an array must be considered as a use of the whole array. On the other hand, a write access is relevant if the array is used in the current context, but the array must not be mapped to modified in the updated context. Indeed, we may have to slice `a[i] = x; a[j] = y;`. If analysing `a[j] = y;` yields to a context where `a` is modified and not used any more, the statement `a[i] = x;` will never be relevant and we may obtain an incorrect slice. In fact, the assignment of a part of an array such as `a[i] = x;` shall behave like the `a = a + i + x;` (where a is not an array any more).

Yet, if we assign the entire array during an initialization for instance, we have to map the array to *mod* in our updated context so that, in `a[i] = x; a =` **new int**`[10];`, the statement `a[i] = x;` is never kept as relevant.

### 4.5.3 Variables declarations

In our previous explanations about interprocedural slicing, we outlined the fact that mapping a variable to *mod* did not appear to be useful and that we could instead only remove its mapping to *ref* from the context. In fact, it is useful if we want to handle variable declarations.

Let us consider the two statements **int** `a; a = 1;` and let the second one be relevant. If we did not keep a trace in our context about $a$ when it is modified, we would not keep its declaration as relevant and we would obtain a slice that is not executable. By contrast, when a variable is declared, we can remove the mapping related to the declared variable from the context. In fact, for coherence purpose, an implementation can choose to introduce another possible state, *dec*, which is used to mark variables as *declared* in a context:

$$\textbf{State} ::= \mathit{ref} \mid \mathit{mod} \mid \mathit{dec}$$

Then, the context can be cleaned frequently by removing the mappings to *dec*, when the analysis reaches the beginning of a block of statements for instance.

### 4.5.4 Structured jumps

Traditional slicing algorithms do not work explicitly on program that contains slicing with jump statements. Some papers proposed to modify the program dependence graph or to use an additional graph to handle them [Agr94].

The problem with jump statements is that they modify the flow of the program and hence special care has to be taken when manipulating the current slicing context. For instance, concerning a **break** statement of a **while** loop, there is direct path from the exit

of the loop to the occurrence of a break. In case of nested loops, breaks can be labelled in order to know which loop is concerned.

Therefore a solution consists of saving the slicing context before entering the loop. Then, when we reach a break, if some statements are relevant so far in the loop (between the break and the end of the loop), we mark the break as relevant and merge the current context with the saved context to account for the merging of the flows.

Let us extend our WHILE language and give some more rules to formalize this. The syntax of statements is extended with ($m \in$ **Lab** refers to the label of a **while**'s control predicate):

$$S ::= \cdots \mid [\textbf{break } m]^{\ell}$$

Then we modify the rules related to the **while** statement we gave in section 4.3.2 so that an additional special assumption is generated, $\lceil < \varnothing, \Gamma, \{\ell\} > \rceil^{[i]}$, where $\Gamma$ is the context before analysing the **while** which is labelled $\ell$. This assumption is of course discharged when the related rules are applied.

We can now give the rules to analyse a break and use the tricky previous assumption. The first one is applicable when some statements are relevant so far (between the point of application and the end of the loop). The second one is applicable in the opposite case.

$$\frac{< S, \Gamma, \mathcal{R} > \quad < \varnothing, \Gamma_{exit}, \{m\} >}{< [\textbf{break } m]^{\ell}; S, \Gamma \uplus \Gamma_{exit}, \mathcal{R} \cup \{\ell\} >} \; [R - break^{[i]}]$$

$$\frac{< S, \Gamma, \mathcal{R} > \quad < \varnothing, \Gamma_{exit}, \{m\} >}{< [\textbf{break } m]^{\ell}; S, \Gamma, \mathcal{R} >} \; [NR - break^{[i]}]$$

Handling **continue** statements in loops can be performed with a slightly similar trick. Last, we can define similar rules to handle **return** statements. Special assumptions are build with the entry context of the procedure and are valid during the entire analysis of the procedure. The difference is that a **return** statement is always relevant if the procedure is relevant.

### 4.5.5 General slicing criteria

So far, we have always sliced from the end of a program or the end of a procedure. If we want to slice from any point of interest $p$ with respect to a set of variables $V$, we can start slicing from the end of the wrapping procedure of $p$ with an empty slicing context. Then, when we reach $p$, we update our empty context with $\{x : V \cdot x \mapsto ref\}$ and continue slicing normally. Since the context was empty before analysing from $p$, no statement, after $p$ in the program flow, was kept as relevant.

In fact, with this trick, we can also slice for a set of criteria $\{(p_1, V_1), \cdots, (p_n, V_n)\}$, at the same time. When $p_i$ is reached, we update our current context with $\{x : V_i \cdot x \mapsto ref\}$. This process allows computing the union of several slices in a single analysis, and can be useful for refactoring tools as we explain in section 5.3.2.

### 4.5.6 Aliasing

Another difficulty in program slicing is to account for *aliasing*. Aliasing is the ability to have two variables sharing the same memory location, which frequently happens when manipulating pointers.

In fact, finding the dependences between variables sharing the same memory location is a problem independent from program slicing, for which various solutions have been proposed. The analysis required is often called *points to* analysis since it aims at gathering information about what each variable points to. Points to analysis can be flow-sensitive to obtain accurate results or flow-insensitive to compute, more quickly, safe but approximative results. The kind of analysis to perform depends on the intended application.

We shall now give an example to explain the issue of aliasing in program slicing:

```
1  x = y;
2  x.a = 1;
3  y.a = 2;
4  print(x.a);
```

Let us slice from the end with respect to `x.a`. Slicing backwards as usual, we do not have any information on `y.a` when we reach the third statement. Hence, we consider it as irrelevant, continue slicing and consider the second statement as relevant. Yet, `x` and `y` share the same location. So, the third statement is in fact relevant and the second one is not.

To overcome this problem, we need to perform a points to analysis before slicing. In the case of refactoring, we believe a flow-sensitive approach is needed since accuracy is particularly important as we have already explained. Thus a solution consists in gathering points-to information by traversing the flow forwards. This could be done at the same time we compute the summary information of each procedure.

### 4.5.7 Object-oriented slicing

Among the most important object-oriented concepts, we can cite *classes*, *inheritance*, *polymorphism* and *scoping*. Many extensions of the SDG-base algorithm were proposed to handle theses features [LH96]. They mostly consists of additional edges to reflect more dependences.

Aliasing is intensively used in the object-oriented paradigm, especially to exploit polymorphism. With the points to information described above, we could also have precise

information about the real type of an object at run-time, in order to resolve polymorphic calls. Indeed, let us consider an abstract class `Shape` with an abstract method:

```
protected abstract void move(int dx, int dy);
```

and two classes `Rectangle` and `Circle` that extend the class `Shape`. Let us also imagine that we have the following piece of code:

```
1  Shape c = new Circle();
2  Rectangle r = new Rectangle();
3  c = r;
4  c.move(10,20);
```

and that we want to slice it from the end for the variable `c`. If we just have the information that `c` is of type `Shape`, like it is declared, then we need to be conservative and assume that `c` could be either of class `Rectangle` or of class `Circle`. Hence, we need to slice the two different implementations of the abstract method `move`, and this may result in very inaccurate slices because of cascade analysis. Here, it is obvious that `c` is of class `Rectangle` but `c` could be passed to the method as a parameter of type `Shape`. In such a case, we need to know what was the real type of the argument when the method was called. Thus, resolving aliasing is a complex but necessary to obtain accurate slices.

Other object-oriented concepts are more easily supported, especially within the inference rules approach. For instance, handling scoping can be done by removing from the slicing context variables that can not be accessed anymore when exiting a procedure. This is specially useful for optimization purposes to avoid manipulating large contexts. Besides, fields of an object behave like global variables. Once again, the slicing context handles them straightforwardly. We just need to keep them in the entry context of a method which may have access to them and to keep them as well when we update back the calling context after the call to this method.

In fact, there are many subtleties we could cite but most of them are dependent on the language you want to slice. In this section, we wanted essentially to give a brief overview about the ability to extend our rules system to support more features. Most of theses extensions were indeed useful for the implementation of our algorithm to slice programs written in the *Java* language.

# Chapter 5

# Implementation

We have experimented with our inference rules approach on slicing Java programs within the *Eclipse* project. Slicing Java programs has of course been studied before but the solution was based on system dependence graphs [KMG96]. In this chapter, we first present Eclipse and describe the useful features it provides. Then, we discuss briefly the design decision of our implementation. Last, we mention the related work that was done thanks to this implementation.

## 5.1 The Eclipse project

### 5.1.1 Presentation

Eclipse is an open-source extensible development platform "for anything, and for nothing in particular" [4]. It was initially developed by the OTI and IBM teams responsible for the IBM IDE products. At the end of 2001, IBM donated the base code of the platform to the open source community.

The project runs on a wide range of operating systems and is build to facilitate additional tools integration. The Eclipse architecture is indeed explicitly designed to allow plug-ins to be built using others thanks to published Application Programming Interfaces. Therefore, any tool builder like us can take advantage of the existing work. In fact, one of the key strength of Eclipse is its active plug-in development community.

A plug-in is the smallest unit of Eclipse platform function that can be developed and delivered separately. Except for the platform runtime which manages the plug-ins mechanism, all of the Eclipse platform's functionality is located in plug-ins.

In Eclipse, the *workspace* consists of one or more top-level projects, where each project contains files that are created and manipulated by the user. All files in the workspace are directly accessible by the integrated tools or any additional plug-ins provided by third

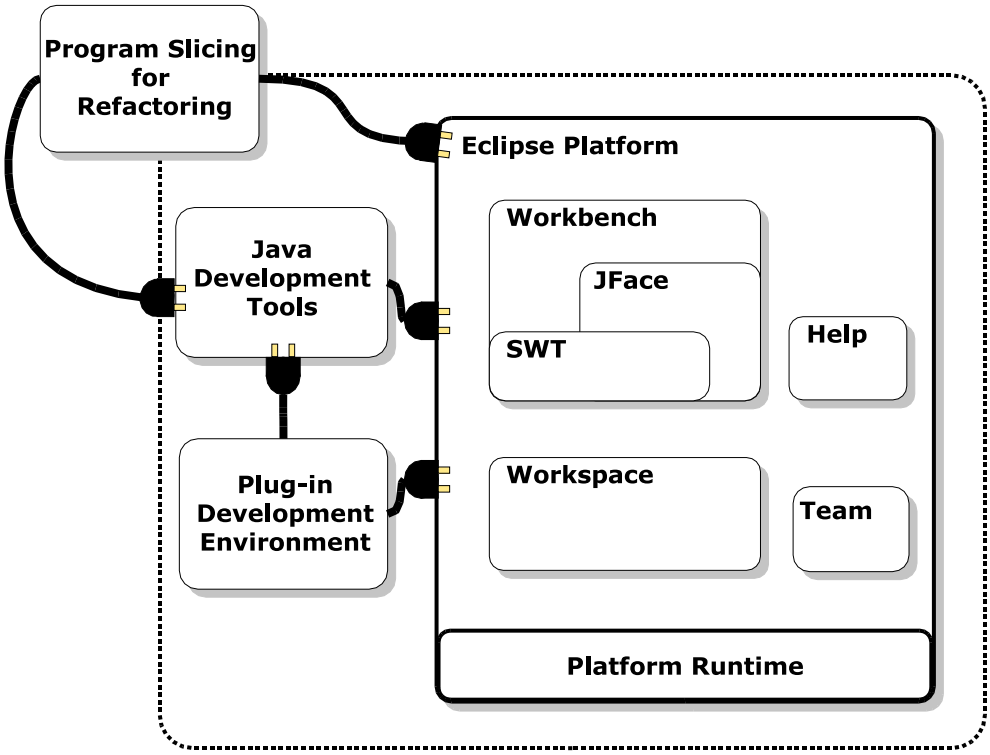



Figure 5.1: Overview of the Eclipse project architecture.

parties.

The *User Interface* is built around an extensible *workbench* which visually consists of views and editors using two toolkits: SWT and JFace. SWT is a widgets set and graphics library integrated with the native window system but with an OS-independent API. JFace is a toolkit implemented on top of SWT that simplifies common UI programming. These two toolkits are not specific to Eclipse and can be used in stand-alone applications instead of AWT and Swing.

Furthermore, a *team* mechanism allows a project in the workspace to be placed under version management with an associated team repository, using *Concurrent Versions System* for instance.

Last, any tool can define its own help files and contribute to the entire documentation of the platform thanks to the provided *help* mechanism.

### 5.1.2 Plug-in development

All plug-ins, including those of the platform, are coded in Java. A plug-in is described in an XML manifest file, which tells the Eclipse runtime what it needs to know to activate the plug-in.

Deploying a plug-in in Eclipse requires copying the resources that constitute the plug-in (the manifest file, jar files and other resources) into an individual folder under the platform's plug-ins directory. The plug-in can then be activated by the Eclipse runtime when it is required to perform some function, i.e. its runtime class is loaded, instantiated and the resulting instance is initialized.

A plug-in may be related to another plug-in by two relationships. It can be *dependent* on another prerequisite one, or it can *extend* the functions of another one, by declaring any number of extension points.

As an example, our slicing plug-in requires four other plug-ins:

```
<requires>
  <import plugin="org.eclipse.core.resources"/>
  <import plugin="org.eclipse.ui"/>
  <import plugin="org.eclipse.jdt.core"/>
  <import plugin="org.eclipse.jdt.ui"/>
</requires>
```

and declares an extension point in a popup menu accessible from the editor of a compilation unit:

```
<extension point="org.eclipse.ui.popupMenus">
  <viewerContribution targetID="#CompilationUnitEditorContext"
                      id="uk.ac.ox.comlab.nate.ui.slicing.popupMenus">
    <action label="Slice..."
            class="uk.ac.ox.comlab.nate.ui.slicing.SlicingActionDelegate"
            menubarPath="org.eclipse.jdt.ui.refactoring.menu/codingGroup"
            id="uk.ac.ox.comlab.nate.ui.slicing.SlicingActionDelegate">
    </action>
  </viewerContribution>
</extension>
```

To help us develop plug-ins while working inside the platform workbench, the Eclipse Platform provides a Plug-in Development Environment, which defines a set of platform extension contributions such as views and editors that collectively ease the process of developing plug-ins. Furthermore, we can launch an other instance of Eclipse, a *run-time workbench*, which has its own workspace, in order to test the plug-in currently being developed.

### 5.1.3 Java development tools

The *Java Development Tools* adds the capabilities of a full-featured Java IDE to the Eclipse platform.

The JDT defines a special Java project nature at the workspace level, and provides a built-in Java compiler that supports incremental building thanks to the use of a files dependence graph. This incremental strategy allows the JDT to run builds very frequently, such as after every source file save operation, even for big projects containing thousands of files.

The *Java model* provides an API for navigating the Java elements tree, which represents a project in terms of Java element types:

- a *package fragment root* represents a project's source folder or a JAR library ;
- a *package fragment* describes a specific package in a package fragment root ;
- a *compilation unit* is an individual Java source (*.java) or a binary class (*.class) file ;
- various *declarations* handle the package, imports, types (classes or interfaces), methods, constructors, fields and initializers declarations of a compilation unit.

Navigating and operating on a Java project via the Java model is more convenient than navigating the underlying resources. Yet, the entire Java elements tree can not be kept in memory since if would require reading and parsing all Java files.

Therefore, the Java elements tree is built only on demand using the Java compiler that parses individual compilation units to extract their declaration structure. In addition to the Java elements tree, it is also possible to ask the compiler for the whole *Abstract Syntax Tree* of a compilation unit whose source is available, and for bindings information, so that we can know, for instance, at any point of program which variable a name refers to and what is the declared type of this variable.

The JDT already plugs into the platform some Java refactorings, including safe renaming for methods, moving a class from one package to another and the Extract Method refactoring we have discussed in Chapter 1. When the developer selects a refactoring operation, an interactive dialog window is prompted to let him select the refactoring's criteria, preview the transformations proposal of the tool and finally confirm the modifications.

Our plug-in for program slicing declares an extension point in the refactoring's menu of the workbench. When the user selects the slicing operation, the plug-in retrieves the current project, the compilation unit which is being edited and the current text selection in the editor. Then, it asks the compiler for the AST of the compilation unit and proceeds to slicing with respect to the criterion defined by the user text selection. Results are displayed in a preview dialog provided within Eclipse to show deltas between two pieces of code.

## 5.2 Design

Let us now describe very briefly the main data structures and operations of the algorithm and their implementation.

### 5.2.1 The slicing context

Variables occurring in a program are of different kinds. We shall distinguish local variables, parameters, types, fields and even the Java key word **this** for convenience. In order to avoid conflicts of variables names in the slicing context, each variable has a unique identifier.

In summary, considering the set $Id$ of all possible identifiers, a variable is defined, in $Z$, as :

$$\textbf{Variable} ::= \mathit{local}\langle\!\langle Id \rangle\!\rangle \mid \mathit{parameter}\langle\!\langle Id \rangle\!\rangle \mid \mathit{type}\langle\!\langle Id \rangle\!\rangle \mid \mathit{field}\langle\!\langle Variable \times Id \rangle\!\rangle \mid \mathit{this}$$

The identifier of a type consists of its full-qualified name (including the package name). The identifier of a field is made of the identifier of the underlying variable and the name of the field. Last, local variables and parameters identifiers are built using the full-qualified name of the declaring method (i.e. the full-qualified name of the declaring class plus the signature of the method) and an identifier provided by the compiler symbols table.

For the sequel, we shall recall the $Z$ definition of **State** and **Context**:

$$\textbf{State} ::= \mathit{ref} \mid \mathit{mod} \mid \mathit{dec}$$

$$\textbf{Context} ::= \textbf{Variable} \pfun \textbf{State}$$

Practically in the implementation, we define an abstract class `Variable` and implement it distinctly for the different kinds of variables. A state is modelled through a class supporting only three instances for *ref*, *mod* and *dec*. Last, the slicing context consists of a hash table mapping variables to a particular state.

### 5.2.2 Inference rules

Basic rules to handle the control flow traversal are implemented directly and do not require any specific data structures. Other rules consist of a condition that states when the rule is applicable and a local context for updating the current slicing context:

**Condition**

$variable$ : **Variable**
$state$ : **State**

---

$state \in \{mod, ref\}$

**Rule**

*condition* : **Condition**
*localContext* : **Context**

---

$condition.variable \in \text{dom}\, localContext$
$condition.state = ref \Leftrightarrow localContext\ condition.variable = mod$
$condition.state = mod \Leftrightarrow localContext\ condition.state = dec$

An interprocedural rule has a slightly different definition since its local context is initially empty and initialized on demand. A complex expression requires a complex rule where several conditions and local contexts are involved but within a particular order. We use a chain of rules to represent such an expression, that may even include a call to a method for instance. A call site is represented with a delegate rule holding the arguments and qualifiers of the call and pointing to a set of interprocedural rules relevant for the call. Therefore, if a call occurs inside a complex expression, the delegate rule of the call will be included in the chain of rules of the complex expression.

Then, we define some *statements rules*. A statement rule associates the AST node of a statement to the inference rule related to this statement and its expressions. Each statement rule includes a mechanism to mark the statement it represents as relevant or not. Statements rules are in fact designed to be used like AST nodes. We aim indeed at representing the program to be sliced as a tree of rules that can be visited like an AST in order to compute the slice.

### 5.2.3 Intensive use of the Visitor pattern

*Design patterns* are elements of reusable object-oriented software. Each design pattern proposes an object-oriented design solution to a particular problem. The *Visitor* design pattern provides the programmer means to define new operations on an object without modifying its class. It is decomposed into two set of classes: the elements and the visitors. The elements represent the structure on which the operations have to be performed. The visitors represent the operations. Each visitor is intended to achieve a specific operation.

The visitors have to implement a Visitor interface which contains a visiting method adapted to each type of element that can be encountered. In the other hand, the elements have to implement an Element interface which contains an accept method.

Eclipse already provides a Visitor mechanism to visit the AST of a compilation unit. We have implemented a similar mechanism to visit the statements rules.

Let us describe the usual slicing process. Once we have obtained the compilation unit being edited and the user text selection, we ask the compiler for the AST of the compilation unit and check using the AST that the text selection corresponds to a variable. Then, still

in the AST, we look for the wrapping method of this variable, so that we know at the end of which procedure we need to start.

A slicing dispatcher is then used to retrieve the summary information for each procedure that may be sliced as we have explained in Subsection 4.4.4. The AST of each method is visited and its resulting summary information is stored in an instance of a special class `MethodSummary`. The slicing dispatcher stores the summary information of all the methods that have been analysed in a hash table. Sometimes, the method to be analysed does not belong to the same compilation unit. Hence, we may need to ask the compiler for the AST of the new compilation unit. The method can even be in libraries for which we do not have any source code. In such a case, the summary information is reduced to the parameters and some scoping information.

Next, still using an AST visitor, we build the statements rules of the method at the end of which we are about to slice, we initialize the slicing context with respect to the variable selected by the user, and start visiting backwards the statements rules. If a statement rule is used to update the context, we mark it as relevant.

If we need to retrieve the lazy local context of an interprocedural rule, we ask the slicing dispatcher for the statements rules of the related method. The slicing dispatcher computes them if they have not been computed yet and stores them for potential reuse. Then, we visit these statements rules for the entry context deduced from the calling context in order to compute the final local context of the rule we needed. Once it is done, we cache the result in the interprocedural rule and continue visiting backwards the initial method. For the moment, we stop the analysis when we reach the beginning of the initial method (i.e. we do not slice up).

At last, we visit the statements rules to delete the nodes of the rules that were not marked as relevant and we display the slice in the preview dialog provided within Eclipse.

## 5.3 Experimentation

### 5.3.1 Implemented features

The implementation is still experimental and we support only a little subset of the Java language. The supported control statements are **if** and **while**. Interprocedural slicing works properly in case of calls to methods for which the source code is available. We try to handle conservatively library calls but the results may be incorrect since we do not account for the variables that could be modified due to their presence in the scope of the called method. Recursive calls are partly supported. There is still a problem in the case of recursive structures such as trees or linked lists. Indeed, the identifiers of the variables are not cycle free. For instance, in the case of a linked list, if a method performs an operation on the value of a cell and calls itself on the next cell, computing the identifier of a variable

holding the value of a cell may never stop ( **this**.next.next.next...). Moreover, we do not perform any aliasing analysis and do not handle aliases at all.

Nevertheless, enough features are supported to allow the development of an experimental refactoring tool and some samples of slices computed by our plug-in are given in appendix B.

### 5.3.2 A novel refactoring tool

Ran Ettinger, who works on the global project we have mentioned in the introduction, has used the program slicing plug-in to build a tool for a refactoring that has never been supported by any tool before: The *Disentangle* refactoring. We describe in the sequel what it achieves and how it works.

Sometimes, the code you want to extract to another method by using the Extract Method refactoring is tangled into a complex method. Selecting consecutive statements to be extracted is not only inappropriate (since it may include some irrelevant computations) but it can also be impossible when several primitive type variables are modified and need to be returned from the extracted method (see Section 1.1). Instead of selecting a whole block, the developer may select the variable he is interested in. The disentangling tool first computes the relevant slice for this variable, then checks refactoring preconditions to know if the slice can be extracted. If the preconditions are not met, the developer is informed and the refactoring is rejected. Otherwise, the tool performs the transformation by extracting the slice to a new method, replacing it with a method call, and for each extracted statement, determines whether to delete it from the source method or leave it there as it is still relevant for the remaining statements. Determining which statements need to be deleted is done by slicing a second time with multiple criteria corresponding to the variables in the statements that were not in the original slice.

We show a sample application of this tool in appendix B. This refactoring is the first step towards refactoring into aspects. Indeed, the extracted code could be defined as an aspect instead of a new method.

# Chapter 6

# Conclusion

## 6.1 Summary

The aim of this thesis was to explore program slicing techniques and implement one of them in order to use program slicing in new refactoring tools.

We have described the different concepts of program slices that have been proposed so far and have characterized the kind of slicing we were interested in for refactoring. Then, we have presented the two most common slicing algorithms, respectively based on data flow equations and on dependence graphs, and have mentioned the calling context problem to which we need to be careful in order to compute accurate slices. As we have explained, we thought these two approaches did not fit refactoring's constraints.

Hence, we have proposed a novel variant of the existing approaches which is more appropriate for refactoring. Our approach is based on inference rules computed on demand and cached for possible reuse. Its strength is that it is easily extensible to support modern languages features such as complex expressions, variables declarations, structured jumps, aliasing and object-oriented concepts.

The core of the algorithm and some of the above features were implemented as an experimental plug-in for the Eclipse project. The implementation was successful enough to allow the development of a new experimental refactoring tool that disentangles and extracts, from a complex method, the piece of code relevant to a particular computation. Slicing-based refactoring seems to be promising.

## 6.2 Future work

Nevertheless, much work needs still to be done to support all the features of the Java language and to improve our approach.

First, performing a full-featured Java program slicing requires handling aliasing, polymorphism (as we have explained in Section 4.5) but also reflection, exceptions and threads, which appears to be very complex. Of course, a solution could consist in rejecting refactoring when the program to be sliced involves threads for instance. Yet, this would considerably restrict the range of possible applications of slicing-based refactoring tools.

An optional but appreciable feature would be to slice libraries as well, in order to compute more accurate slices when they involve library calls. This would require analysing Java bytecode. *Soot*, a Java optimization framework that provides intermediate representations for analyzing and transforming Java bytecode, may be helpful in this task [5]. It is currently being plugged into Eclipse.

Concerning the algorithm, slicing using inference rules is just another approach, and can certainly be expressed using the framework for demand-driven interprocedural data flow analysis, proposed by Duesterwald *et al.* in [DGS97]. Moreover, we shall state the complexity of our approach and compare it to the existing slicing approaches both in a theoretical way and practically by slicing sample programs.

Last, computations of the rules could be incremental by listening to the changes in the source code and recomputing only the rules that may have changed directly or indirectly because of dependences. The approach should also be made robust with incomplete information. Harmonia, an open extensible framework for constructing interactive programming tools, deals with these issues [6]. It is also being ported to Eclipse and may be useful to achieve future work.

# Bibliography


[1] *Martin Fowler's refactoring page*. http://www.refactoring.com.

[2] *Aspect-Oriented Software Development*. http://www.aosd.net.

[3] *AspectJ*. http://eclipse.org/aspectj.

[4] *Eclipse*. http://www.eclipse.org.

[5] *The Soot Framework*. http://www.sable.mcgill.ca/soot.

[6] *The Harmonia Research Project*. http://harmonia.cs.berkeley.edu.

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, 1994.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[HD03] Mark Harman and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[KMG96] Gyula Kovcs, Ferenc Magyar, and Tibor Gyimthy. Static slicing of Java programs. Unpublished, 1996.

[LDK99] Arun Lakhotia, Jean-Christophe Deprez, and Shreyash S. Kame. Flow analysis models for interprocedural program slicing algorithms. Unpublished, 1999.

[LH96] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, 1996.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *Software Development Environments (SDE)*, pages 177–184, 1984.

[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.

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

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

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

# Appendix A

# The calling context problem

We have mentioned the calling context problem in Section 3.3. Figure A.1 reveals the problem when we slice the multiprocedural program of Figure 3.4 with respect to node `print(product);` using the SDG initially introduced. Indeed, it shows a directed path from `sum := 0;` to `print(product);`, although node `sum := 0;` should not be in the slice.

To overcome this problem, Horwitz *et al.* introduced some summary edges at call sites and proposed to compute slices in two phases. In the first one, we do not descend into called procedures. Instead we use summary edges. In the second one, we do not ascend into calling procedures, so that no infeasible path can be followed. Figure A.2 shows the new SDG with summary edges represented by dotted arrows.

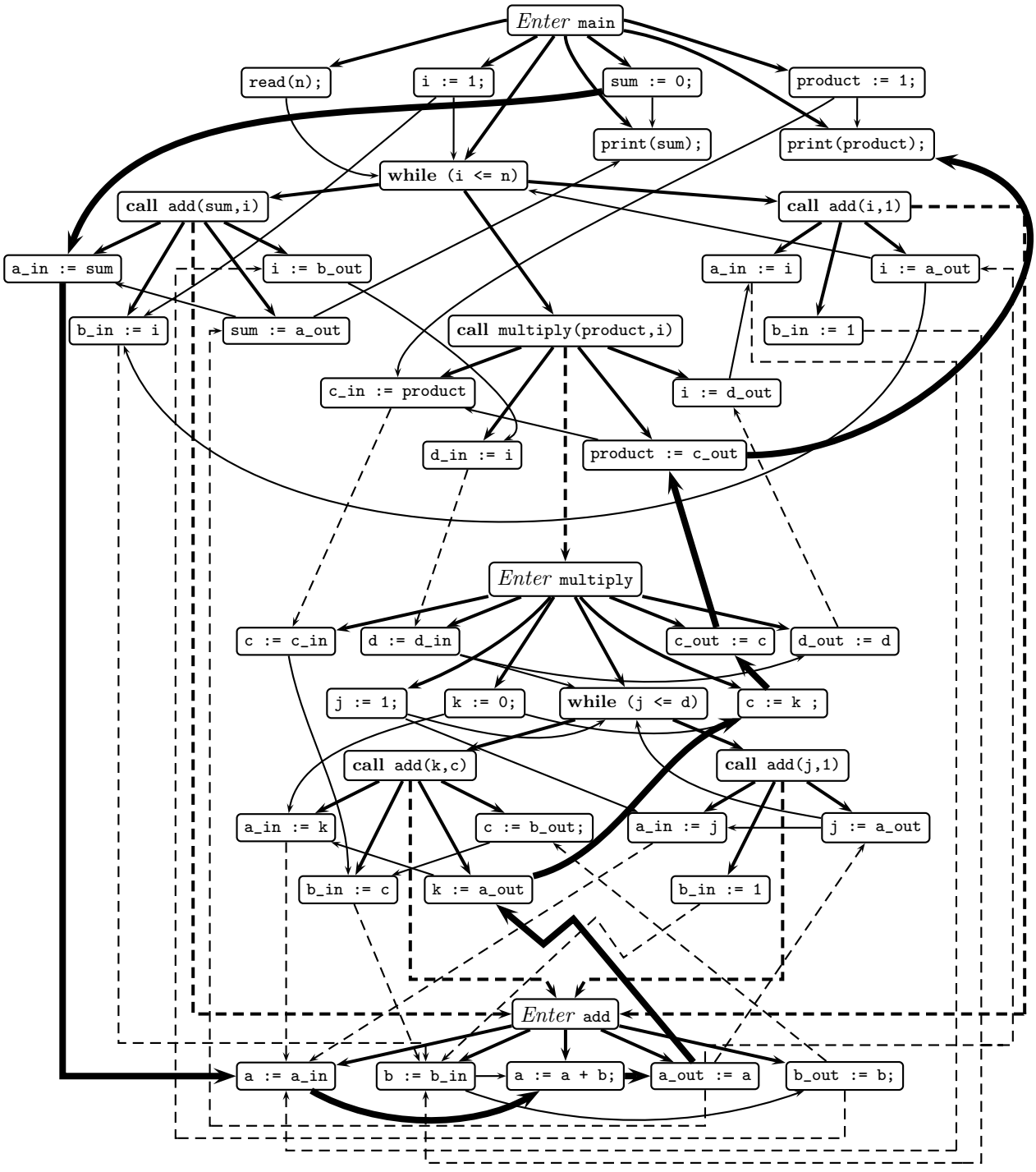



Figure A.1: An infeasible execution path from `sum := 0;` to `print(product);`.

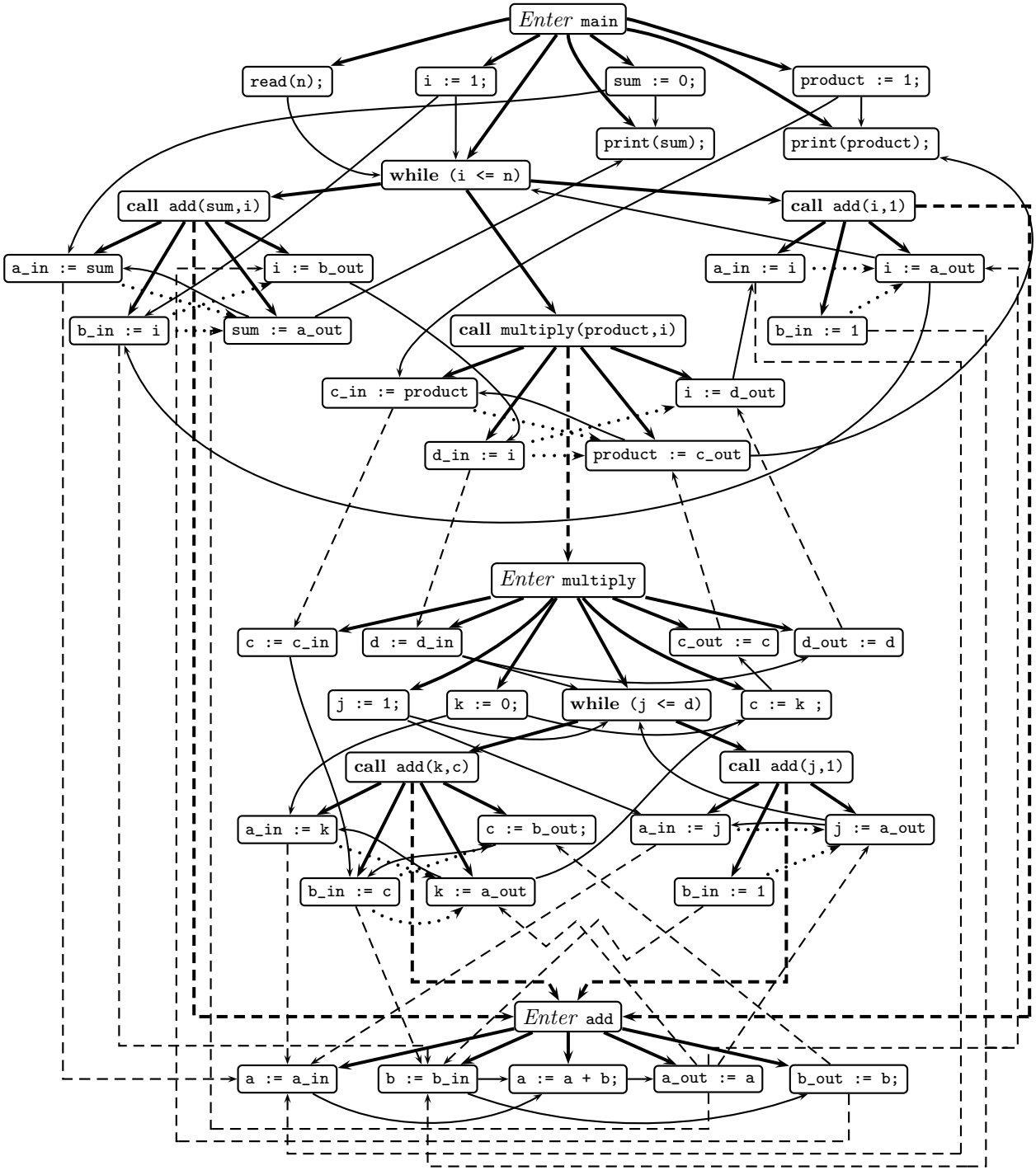



Figure A.2: SDG of the multiprocedural program of Figure 3.4 with summary edges.

# Appendix B

# Screenshots

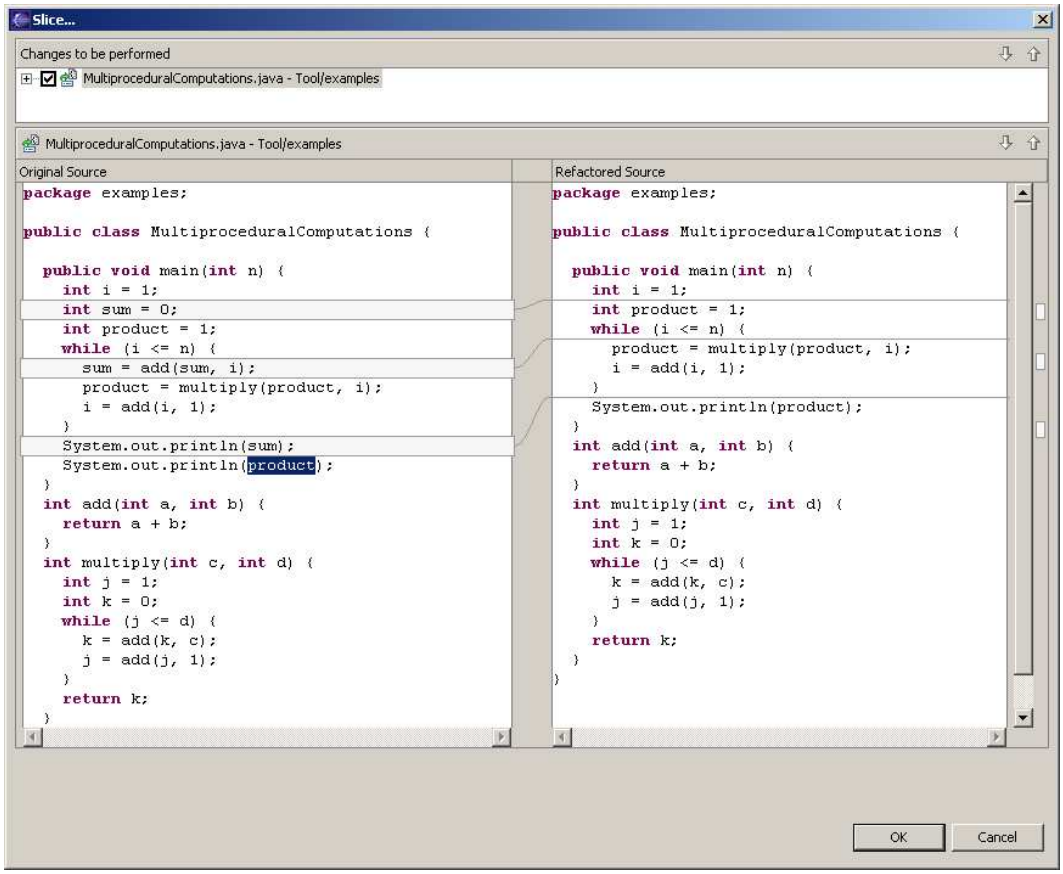



Figure B.1: The slice of our sample multiprocedural program computed from the last statement for the variable *product* (highlighted text). The irrelevant statements are removed.

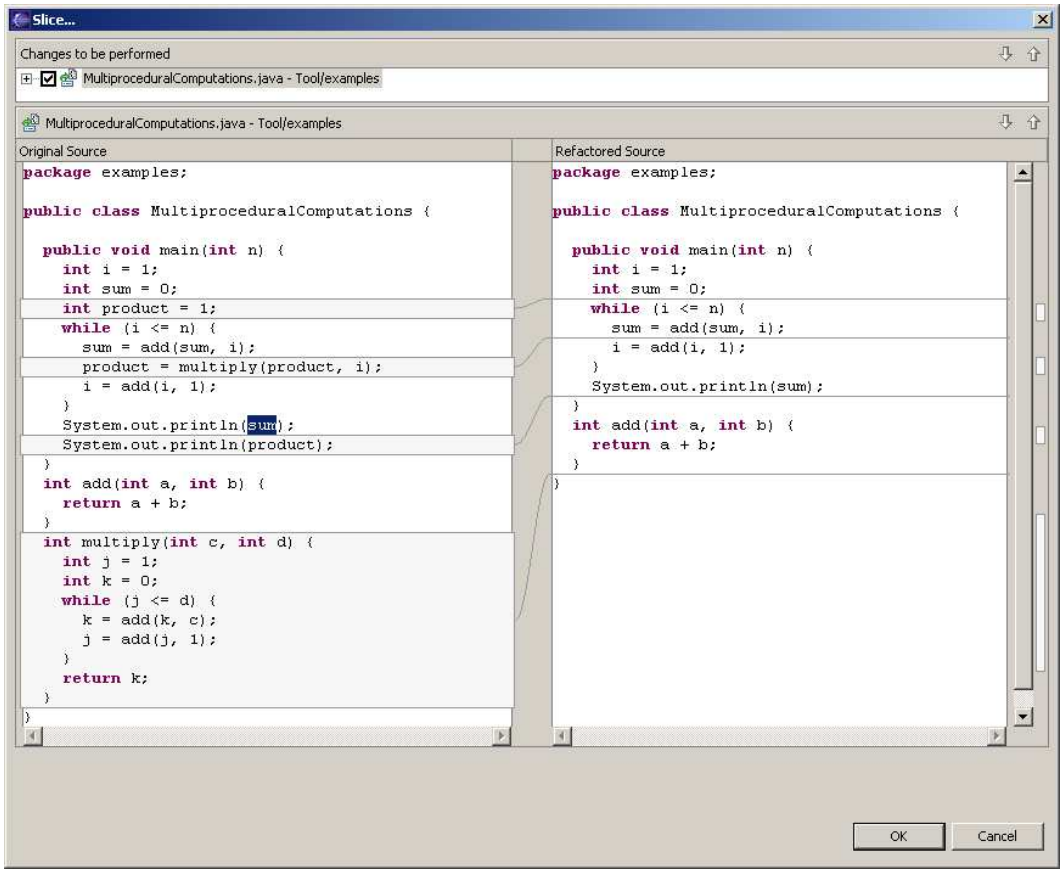



Figure B.2: The slice of our sample multiprocedural program computed for the highlighted variable *sum*.

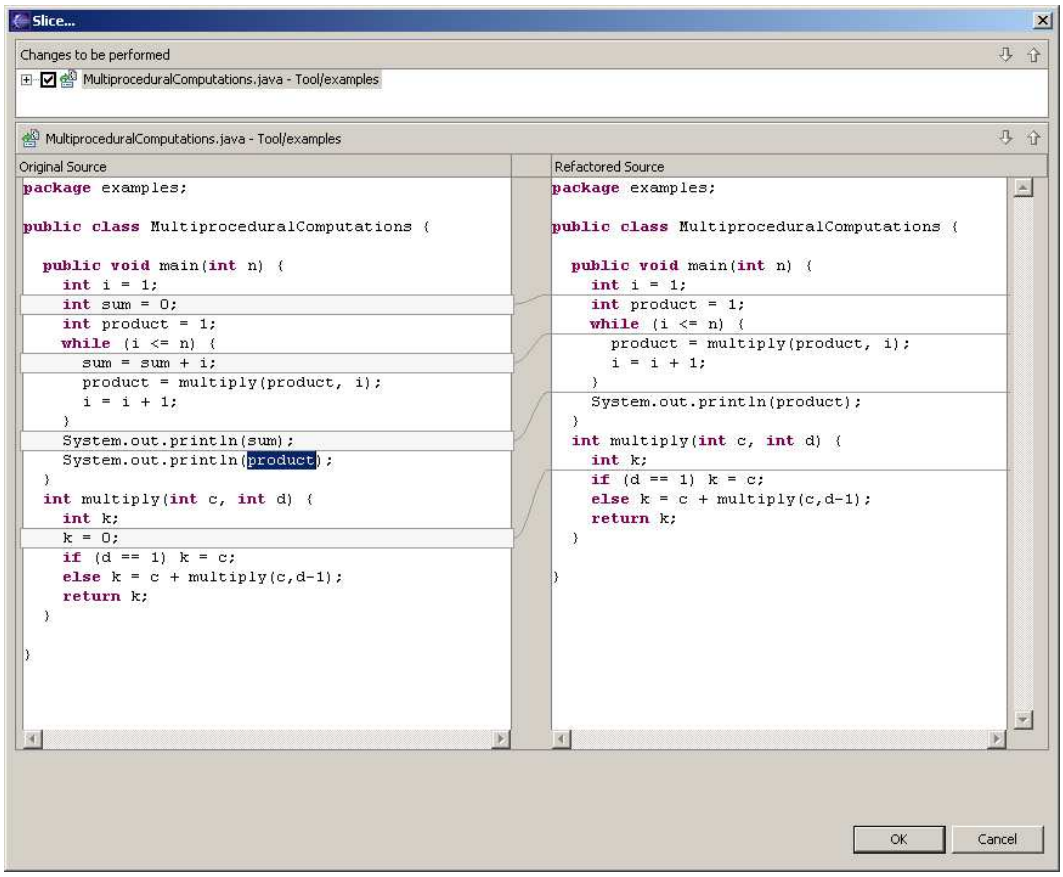



Figure B.3: Slicing of a variant multiprocedural program involving a recursive call. The initialization of $k$ to zero is irrelevant because $k$ is assigned in the two branches of the **if** statement.

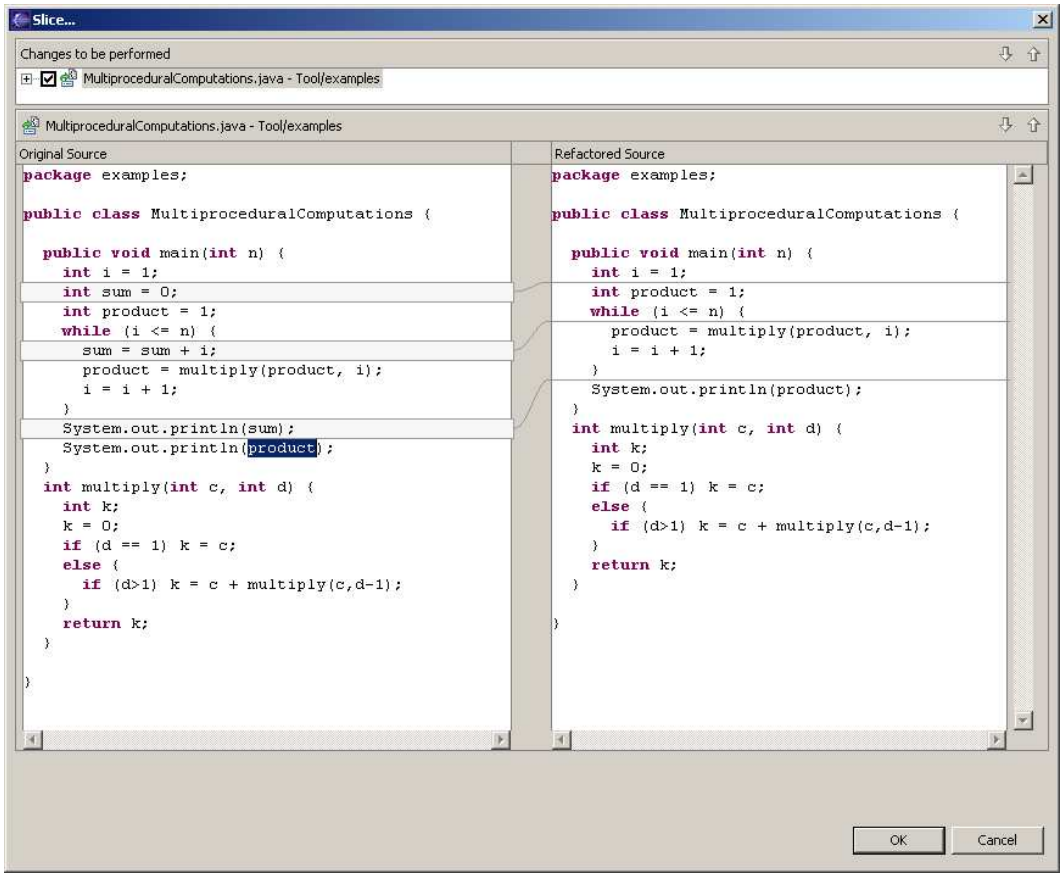



Figure B.4: Slicing of almost the same variant in order to show the account for control flow. The initialization of $k$ to zero is here relevant because assignments of $k$ are only potential in the **if** statements.

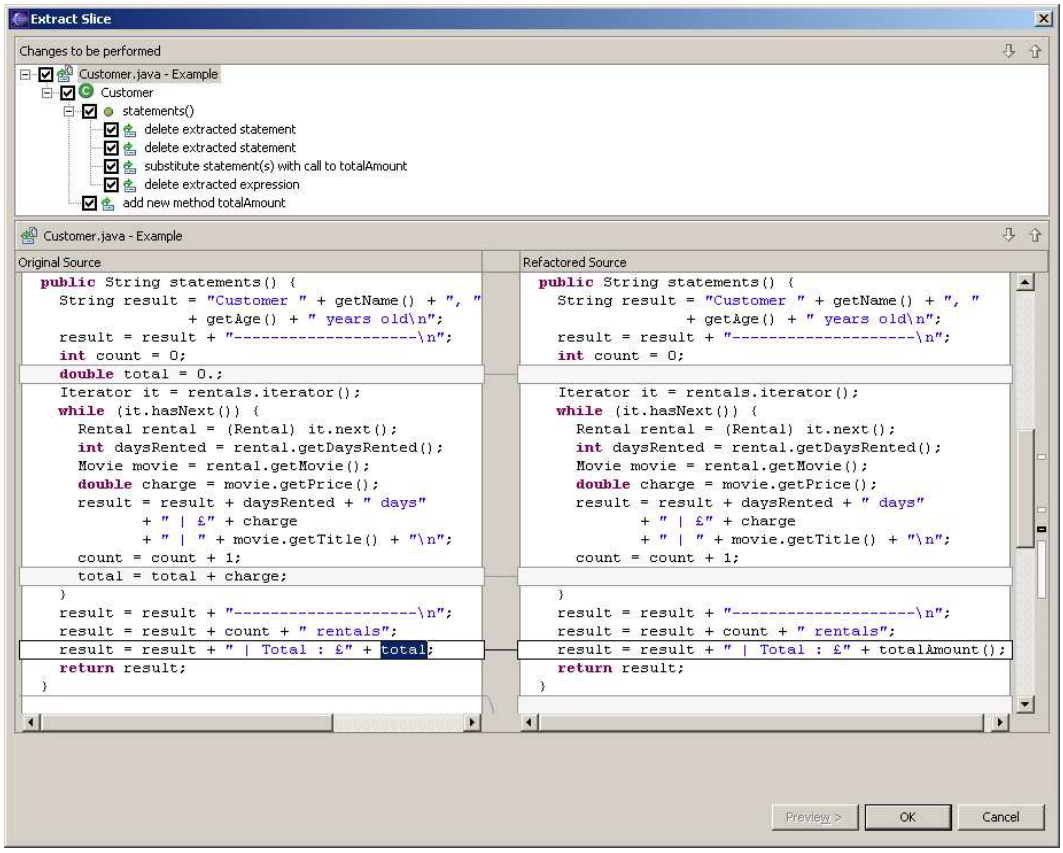



Figure B.5: The disentangling tool extracts the slice relevant to *total* from the *statements* method into...

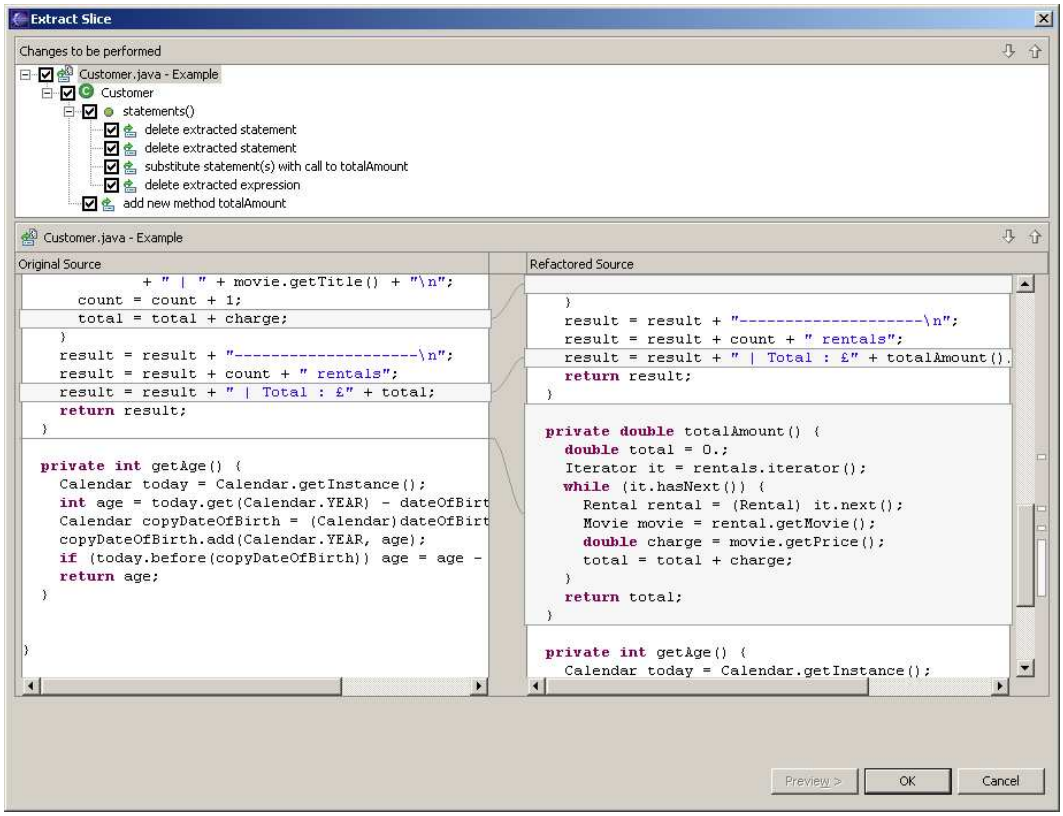



Figure B.6: ... a new method called *totalAmount*.