# Fragmentación de programas con excepciones

Trabajo Fin de Máster

**Máster Universitario en Ingeniería y Tecnología de Sistemas Software**

Departamento de Sistemas Informáticos y Computación

**Autor**: Carlos S. Galindo Jiménez
**Tutor**: Josep Francesc Silva Galiana
Valencia, diciembre de 2019
Curso 2019-2020

**Abstract**

#CCC: por completar

**Resumen**

#CCC: por completar

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

#CCC: Presentar más que definir program slicing.

Program slicing [17] is a debugging technique that, given a line of code and a set of variables of a program, simplifies such program so that the only parts left of it are those that affect or are affected by the values of the selected variables. #JJJ: aqui, antes del ejemplo, habria que decir de manera informal que es un slice y que es un SC

#SSS: Se me hace corta esta definicion y me faltan algunas utilidades del program slicing, por que se usa? Realmente no se usa solo en depuracion. Tiene mas usos, esto ademas da referencias a poner si queremos.

#SSS: Carpeta SAC 2017 (paper-poster 3 paginas): "Program slicing is a technique for program analysis and transformation whose main objective is to extract from a program those statements (the slice) that influence or are influenced by the values of one or more variables at some point of interest, often called slicing criterion [13, 12, 1, 9]. This technique has been adapted to practically all programming languages, and it has many applications such as debugging [3], program specialization [8], software maintenance [5], code obfuscation [7], etc.".

#SSS: Cogeria algo de aqui para hacer una definicion mas completa, ademas ya usamos terminologia de slicing como *slice* y *slicing criterion.* #JJJ: De acuerdo con Sergio. Un par de cosas más: Entra muy a saco la introducción con una definición. :-) Por otra parte, tal y como está definido (para el lector profano), parece que un slice es todo lo que afecta O es afectado por el slicing criterion. Es decir, como si el "O" formara parte de la definición. Yo hablaría aquí solo de backward slicing, y dejaría forward para luego (igual que has dejado dynamic para luego).

**Example 1** (Program slicing #JJJ: #Deleted: in#Added: applied to a simple method)**.** #SSS: Consider the code shown below / in Figure XX, containing a simple method written in Java. If the left program is sliced on (line 5, variable `x`),#SSS: Si hemos usado ya slice y slicing criterion aqui podemos decir que el slicing criterion es tal y el slice es cual y empezar a usar la terminologia correcta de manera mas natural. the result would be the program on the right, with the `if` block removed, as it does not affect the value of `x`.

```
1 void f(int x) {
2   if (x < 0)
3     System.err.println(x);
4   x++;
5   System.out.println(x);
6 }
```

```
1 void f(int x) {
2
3
4   x++;
5   System.out.println(x);
6 }
```

#CCC: Detallar los distintos usos y evitar relacionar debugging con ejecutable.

Slices are executable programs whose execution produces the same values #SSS: OJO!, cuidao con ese jardin que luego esta el weak slice.#JJJ: puedes evitar el jardín empezando la frase así: “In its more general form, slices are...” #CCC: Alternativa: programa que se comporta igual (luego se define mismos valores o lista prefija.) for the specified line and variable as the original program, and they are used to facilitate debugging of large and complex programs, where the control and data flow may not be easily understandable. #JJJ: en realidad los executable slices no suelen usarse en debugging. Más bien en Program specialization...

Though it may seem a really powerful technique, #JJJ: many languages lack of a mature program slicer or one that covers the whole language. For instance, the whole Java #SSS: Primera aparicion de Java, mencionar que el ejemplo es Java porque sino parece que te aparece Java out of the blue. language is not completely covered by it, and that makes it difficult to apply in practical settings.

#SSS: Propongo algo asi para conectar program slicing y las exceptions:

#SSS: Though it may seem a really powerful technique, the amount of analysis that need to be done to properly obtain a correct slice is very considerable. Many situations of the Java language lead to several scenarios (podriamos poner algun ejemplo de cosas chungas, rollo recursividad, arrays, objetos... para que se vea que no todo tiene una solucion unica ni perfecta, sino que muchas propuestas son mejorables.) that are quite difficult to analyse, which is the reason because there does not exist a universal solution for all the existent problems in the field of program slicing. Conversely, many different approaches are usually proposed to solution the same slicing problem.

#SSS: Se que hay mucha verborrea, pero es para hacer la lectura menos agresiva xD.#JJJ: Carlos va directo al grano, no se anda con rodeos. :-). Pero, efectivamente es necesario (luego no, pero aquí en a introducción sí) darle un poco de cremita al lector. Esto es la motivación y de momento no ha habido motivación. Echo en falta decirle que la técnica ha sido aplicada y estudiada en prácticamente todos los lenguajes de programación y que es una técnica de optimización que usan los compiladores y muchas técnicas de análisis estático. Que se aplica en debugging, program compehension, paralelización, eliminación de código muerto, etc. falta motivar que este area es importante, no es solo una paja mental y una vuelta de tuerca mas... todo esto para los profanos (e.g., el tribunal ;-))

#SSS: Inside all this slicing problems, there is An area that has been investigated, #SSS: but? (por evitar el yet ... yet)yet does not have a definitive solution yet#SSS: , #Deleted: is#JJJ: el is no hay que borrarlo exception handling. Example 2 demonstrates#JJJ: shows how, even using the latest developments #JJJ: to handle exceptions in in #SSS: exception handling slicing program slicing [2], the sliced version does not include the catch block #SSS: this approach is not able to include the catch block in the obtained slice, and therefore does not produce a correct slice.

**Example 2** (Program slicing with exceptions)**.** #Added: Consider#Deleted: If the following program #JJJ: on the left that has been sliced (on the right) using#Deleted: is sliced using Allen and Horwitz’s proposal [2] with respect to (line 17, variable `a`)#Added: . As #JJJ: it can be appreciated, t#Deleted: , the slice is incomplete, #JJJ: becauseas it lacks the `catch` block from lines 4-6.

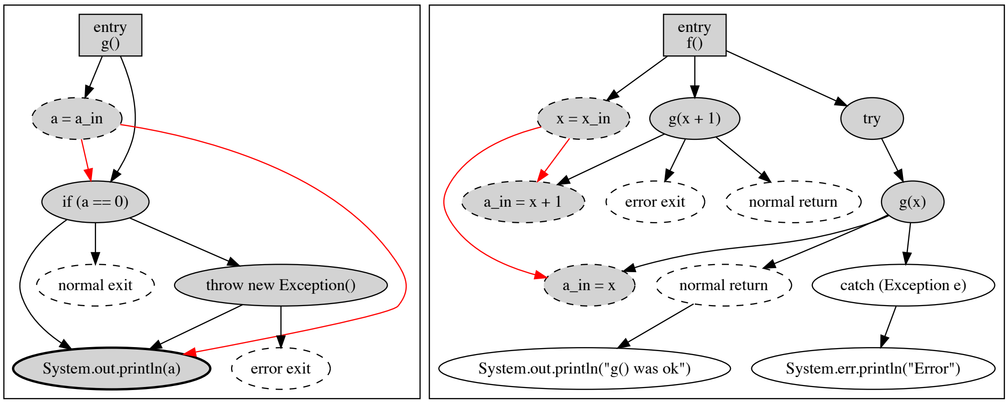
```java
1  void f(int x) throws Exception {
2    try {
3      g(x);
4    } catch (Exception e) {
5      System.err.println("Error");
6    }
7
8    System.out.println("g()␣was␣ok");
9
10   g(x + 1);
11 }
12
13 void g(int a) throws Exception {
14   if (a == 0) {
15     throw new Exception();
16   }
17   System.out.println(a);
18 }
```

```java
1  void f(int x) throws Exception {
2    try {
3      g(x);
4    }
5
6
7
8
9
10   g(x + 1);
11 }
12
13 void g(int a) throws Exception {
14   if (a == 0) {
15     throw new Exception();
16   }
17   System.out.println(a);
18 }
```

#SSS: Captions? para referirnos a ellas por separado como programa original (izquierda) y slice (derecha)?. Indicar en ella el SC en negrita en el codigo o de otro color o algo para destacarlo.

When the program is executed #JJJ: from the callas `f(0)`, the execution log #SSS: hay que decir que esto es la lista de instrucciones que se ejecutan y en el orden en el que lo hacen.#JJJ: en program slicing se llama execution history, y puedes poner una cita a Corel y Lasky would be: `1, 2, 3, 13, 14, 15, 4, 5, 8, 10, 13, 14, 17`. In the only execution of line `17`, variable `a` has value 1 in that line. However, in the slice produced, the execution log is `1, 2, 3, 13, 14, 15`. The exception thrown in `g()` is not caught in `f()`, so it returns with an exception and line `17` never executes.

The problem in this example is that the `catch` block in line `4` is not included, because — according to the dependency graph #JJJ: computed by [] and shown #JJJ: in Figure ??below— it does not influence the execution of line `17`. Two kinds of dependencies among statements are considered: data dependence (a variable is read that may have gotten its value from a given statement) and control dependence (#JJJ: an the instruction controls whether another #JJJ: instruction executes). In the graph, the #JJJ: node associated with the slicing criterion is marked in bold, the nodes that represent the slice are filled in grey#JJJ: demasiado clarito. En mi ordenador no se ve, and dependencies are displayed as edges, with control dependencies in black and data dependencies in red. Nodes with a dashed outline represent elements that are not statements of the program.




#JJJ: transforma todas las figuras en figuras reales (referenciables) y con caption #JJJ: Yo vería más claro el grafo conectando llamada y llamado

#CCC: mover todas las imágenes y segmentos de código a figuras separadas
#CCC: indicar la conexión entre grafos
#CCC: mover el grafo y la explicación a después del background; el porqué y la solución se presenta en sección X

Example 2 #JJJ: is a contribution of this work because it showcases an important error in the current slicing procedure for programs that handle errors with exceptions#JJJ: #Deleted: ; because#Added: where the `catch` block is disregarded. The only way a `catch` block can be included in the slice is if a statement inside it is needed for another reason. However, Allen and Horwitz [2] did not encounter#JJJ: tackle? account for? this problem in their paper, as the values outputted by method calls are extracted after the `normal return` and each `catch`, and in a typical method call with output, the `catch` is included by default when the outputted value is used. This detail makes the error much smaller, as most `try-catch` structures are run to obtain a value. #SSS: Anyadir el nodo *out* para que lo que has explicado aqui quede mas comprensible. Viendo que existe el nodo *out*, pero que nadie el SC no lo necesita.

#Added: There is also another #Deleted: A notable case where a method that may throw an exception is run and no value is recovered (at least from the point of view of program slicing)#Added: . It occurs#Deleted: is when writing to the filesystem or making connections to servers, such as a database or a webservice to store information. In this case, if no confirmation is outputted signaling whether the storage of information was correct, the `catch` block #Deleted: will be#Added: is omitted, and the #JJJ: program slicer #JJJ: #Deleted: software #Deleted: will produce#Added: s an incorrect result.

## 1.2 Contributions

The main contribution of this paper#CCC: thesis#SSS: paper?research?work?#JJJ: work o research is a #Added: new approach for program slicing with exception handling for Java programs. #Deleted: complete technique for program slicing programs in the presence of exception handling constructs for Java. #Added: Our approach#Deleted: This technique extends the previous technique #Added: proposed by Allen et al. [2]. It #Added: is able to properly slice#Deleted: considers all cases considered in #Deleted: that#Added: their work, but it also provides a solution to #SSS: some other cases not #Deleted: considered#Added: contemplated#JJJ: considered by them.

For the sake of completeness and in order to understand the process that leaded us to this solution, we #JJJ: #Deleted: will present#JJJ: first summarize the fundamentals o background a brief history#SSS: background? of program slicing #Added: terminology, specifically those changes that have affected exception handling.#SSS: delving deeper in the progress of program slicing techniques related to exception handling.? Furthermore, we provide a summary of the different contributions each author has made to the field.

The rest of the paper is structured as follows: chapter 2 summarizes the theoretical background required in program slicing and exception handling, chapter 3 #JJJ: analyzeswill analyze each structure used in exception handling, explore#JJJ: s the already available solution and propose#JJJ: s a new technique that subsumes all of the existing solutions and provides correct slices for each case.#JJJ: frase demasiado larga Chapter 5 provides a bird's eye view of the current state of the art, chapter 4 provides a summarized description of the new algorithm with all the changes proposed in chapter 3, and finally, chapter **??** #JJJ: concludes?summarizes the paper#SSS: work? and explores future avenues of work#SSS: possible improvements?.

# Chapter 2

# Background

## 2.1 Program slicing

#CCC: citar a Weiser solo hablando del inicio del campo
#CCC: el resto, utilizar surveys (Tip95, Sil12)
#CCC: mover párrafo a la intro, aquí poner definiciones formales de program slicing, citar a [1]

*Program slicing* [17, 14]#SSS: hay alguna razon para que [14] no este en la intro?, la unica cita alli es[17]. Propongo eliminar [14] por homogeneidad#JJJ: mas bien, tendria que estar 13 también en la intro is a debugging technique that answers the question: "which parts of a program #JJJ: do? affect a given statement and set of variables?" The statement and the variables are the basic input to create a slice and are called the *slicing criterion*. The criterion can be more complex, as different slicing techniques may require additional pieces of input. The *slice* of a program is the list of statements from the original program —which constitutes a valid program— whose execution will result in the same values for the variables #JJJ: frase enrrevesada. yo la. cambiaria. De todas formas, para que sea correcta le sobran los parentesis (selected in the slicing criterion). There exist two fundamental dimensions along which the problem of slicing can be proposed [14]:

#SSS: Mi propuesta es mover el concepto naive de aqui a la intro para que entiendan algo del ejemplo y aqui hacer referencia a la definicion anterior o introducir las dimensiones de slicing directamente con un pequenyo preambulo. Una fuerte razon para definirlo alli es que usamos todo el rato la palabra slice y de repente, despues de usarla un rato, la definimos.

- *Static* or *dynamic*: slicing can be performed statically or dynamically. *Static slicing* [17] produces slices which#JJJ: that consider all possible executions of the program: the slice will be correct regardless of the input supplied. In contrast, *dynamic slicing* [11, 1] considers a single execution of the program, thus, limiting the slice to the statements present in an execution log. The slicing criterion is expanded to include a position in the log#JJJ: execution history that corresponds to one instance of the selected statement, making it much more specific. It may help #JJJ: tofind a bug related to indeterministic behavior (such as a random or pseudo-random number generator), but #SSS: , despite selecting the same slicing criterion, the slice must be recomputed for each case#SSS: different input value/execution considered? being analyzed.

- *Backward* or *forward*: *backward slicing* [17] is generally more used #SSS: habra que decir lo que es antes de decir que se usa mas no? Cambiar el orden y reescribir esta frase. Decimos que es y luego que es el que generalmente se estudia o algo de eso, because it looks at the statements that affect the slicing criterion. In contrast, *forward slicing* [4] computes the statements that are affected by the slicing criterion. There also exists a mixed approach called *chopping* [7], which is used to find all statements that affect some variables in the slicing criterion and at the same time they are affected by some other variables in the slicing criterion.

Since the definition of program slicing#SSS: Since Weiser defined program slicing in 1981, the most #Deleted: extended form#Added: studied configuration? of slicing has been *static backward slicing*, which obtains the list of statements that affect the value of a variable in a given statement, in all possible executions of the program (i.e., for any input data).

**Definition 1** (Strong static backward slice [17]). Given a program $P$ and a slicing criterion $C = \langle s, v \rangle$, where $s$ is a statement and $v$ is a set#SSS: los set no se representan con letras mayusculas? #CCC: no of variables in $P$ (the variables may or may not be used in $s$), $S$ is the *strong slice* of $P$ with respect to $C$ if $S$ has#SSS: fulfils? the following properties:

1. $S$ is an executable program.
2. $S \subseteq P$, or $S$ is the result of removing code#SSS: code o 0 or more statements? from $P$.
3. For any input $I$, the values produced on each execution of $s$ for each of the variables in $v$ is the same when executing $S$ as when executing $P$.

#SSS: Esta definicion no obligaba tambien a acabar con el mismo error en caso de que la ejecucion no termine? Si es asi, plantearse poner algo al respecto. #JJJ: hay que revisar la definición de (1) Weiser, (2) Binkley y Gallagher y (3) Frank Tip. Mi opinion es que NO: Creo que no es necesario que el error se repita. Lo que dice es que el valor de las variables del SC debe ser el mismo, pero no dice nada del error.

**Definition 2** (Weak static backward slice [13]). #CCC: Check citation and improve "formalization"? #JJJ: Si esa cita no es, entonces puedes usar la de Binkley: `https://cgi.csc.liv.ac.uk/~coopes/comp319/2016/papers/ProgramSlicing-Binkley+Gallagher.pdf` Given a program $P$ and a slicing criterion $C = \langle s, v \rangle$, where $s$ is a statement and $v$ is a set of variables in $P$ (the variables may or may not be used in $s$), $S$ is the *weak slice* of $P$ with respect to $C$ if $S$ has#SSS: fulfils? the following properties:

1. $S$ is an executable program.
2. $S \subseteq P$, or $S$ is the result of removing code from $P$. #SSS: idem
3. For any input $I$, the values produced on each execution of $s$ for each of the variables in $v$ when executing $P$ is a prefix of those produced while executing $S$ —which means that the slice may continue producing values, but the first values produced always match up with all those produced by the original program.

#SSS: $\forall\ i\ \in\ I, v \in\ V\ \rightarrow\ seq(i, v, P)\ Pref\ seq(i, v, S)$ where $seq(i, a, A)$ representa la secuencia de valores obtenidos para $a$ al ejecutar el input $i$ en el programa $A$. $I$ es el conjunto de todos los inputs posibles para $P$. Por ahi irian los tiros creo yo. #SSS: Formalizacion existente en el repo: Program Slicing → Trabajos → Erlang Benchmarks → Papers → ICSM 2018 → Submitted (Section III - A) #JJJ: Si se formaliza con el uso de seq, entonces puedes mirar la definicion del paper de POI testing (Sergio sabe cual es).

Both definitions (1 and 2) are used throughout the literature (see, e.g., [**?**]#CCC: Which citation? Most papers on exception slicing do not indicate or hint whether they use strong or

| #Deleted: Iteration#Added: Evaluation Number | **1** | **2** | **3** | **4** | **5** |
|---:|---|---|---|---|---|
| Original | 1 | 2 | 6 | - | - |
| Slice A | 1 | 2 | 6 | - | - |
| Slice B | 1 | 2 | 6 | 24 | 120 |
| Slice C | 1 | 1 | 3 | 5 | 8 |

Table 2.1: #Deleted: Execution logs of different slices and their original program.#Added: Sequence of values obtained for a certain variable of the original program and three different slices A, B and C for a particular input.

weak.#SSS: Josep?#JJJ: para Strong se puede poner a Weiser. Para Weak se puede poner a Binkley `https://cgi.csc.liv.ac.uk/~coopes/comp319/2016/papers/ProgramSlicing-Binkley+Gallagher.pdf`), #JJJ: este final de frase lo quitaria:with some cases #Deleted: favoring#Added: favouring the first and some the second. Though the definitions come from the corresponding citations, the naming was first used in a control dependency analysis by Danicic [5], where slices that produce the same output as the original are named *strong*, and those where the original is a prefix of the slice, *weak*. Weak slicing tends to be preferred —specially for debugging— for two reasons: the algorithm can be simpler and avoid dealing with termination #JJJ: termination no esta contemplada ni en weak ni en strong. Mas bien di que en debugging lo que importa es que el error se produzca. En general da igual cuantas veces se produzca o que se siga produciendo despues., and the slices can be smaller, narrowing the focus of the debugger. For some applications, #Deleted: strong slices are preferred, such as extracting a #JJJ: component or a specialized programfeature from a program, where there is a requirement that the resulting slice behave#JJJ: s exactly like#JJJ: as the original#Added: , strong slices are preferred#JJJ: esto queda muy lejos ya. Yo partiria la frase en dos. In this paper we will #JJJ: Along the thesis, we indicate indicate which kind of slice is produced with each new technique proposed. #SSS: Generamos alguna vez strong? Joder que cracks somos xD

**Example 3** (Strong, weak and incorrect slices)**.** #CCC: The table is labeled execution logs of... but the execution log is a different thing. In table 2.1 we can observe examples for the various definitions. Each row shows the values #SSS: for a specific variable $v$ in the slicing criterion, produced by #Deleted: the#Added: a particular execution of #Deleted: a#SSS: the original program or one of its slices. The first #Added: row stands for#Deleted: is the original #Added: program, which computes 3!. Slice A's #Deleted: execution log#Added: generated sequence of values is identical to the original and therefore it is a strong slice. Slice B is a weak slice: its execution correctly produces the same #Added: sequence of values as the original program, but it continues producing values after the original stops. Slice C is incorrect, as the #Added: generated sequence of values differ#Added: s from the #Added: sequence generated by the original #Added: program. #SSS: Taking a closer look, one could think that Some data or control dependency has not been included in the slice #JJJ: lo que sigue quitarlo. Lia...and the program produce#JJJ: s different results, in this case the slice computes Fibonacci numbers instead of factorials.#SSS: Esto no parece muy relevante, plantearse quitarlo para no liar con Fibonacci.

#JJJ: Even though the original proposal by Weiser [17] focussed on an imperative language, program slicing is a language–agnostic technique. Program slicing is a language–agnostic tool#SSS: program slicing es tool o technique?, but the original proposal by Weiser [17] covered a simple imperative programming language. Since then, the literature has been expanded by dozens of authors, that have described and implemented slicing for more complex structures, such as uncontrolled control flow [6], global variables [**?**], exception handling [2]; and for other programming paradigms, such as object–oriented languages [?] or functional languages [?]. #CCC: Se pueden poner más, faltan las citas correspondientes.#SSS: Guay, hay que buscarlas y ponerlas, la biblio la veo corta para todos los papers que hay, yo creo que cuando este todo deberia haber sobre 30 casi, si no mas. #JJJ: Si. Muchas de esas referencias puedes sacarlas de los ultimos surveys de slicing.

### 2.1.1 The System Dependence Graph (SDG)

There exist multiple approaches to compute a slice#SSS: esto me suena raro, yo diria program representations o data structures that allow the use of program slicing techniques o algo asi, debatirlo#CCC: DENIED from a given program and slicing criterion, but the most efficient and broadly used #JJJ: technique is based on a data structure called data structure is the System Dependence Graph (SDG), first introduced by Horwitz, Reps#JJJ: , and Blinkey #SSS: in 1988#SSS: Todos los autores o los citamos con et al.? lo digo por seguir la misma regla durante todo el document [6]. It is computed from the program's statements#SSS: source code, and once built, a slicing criterion is chosen #JJJ: and mapped on the graph, then , the graph #Added: is traversed using a specific algorithm, and the slice #Added: is obtained. Its efficiency resides#JJJ: relies? on in the fact that#Added: , for multiple slices #Deleted: that share#Added: calculated for the same program, the graph #Deleted: must only be built#Added: generation process is only performed once. On top of that, building the graph has a complexity of $\mathcal{O}(n^2)$ #CCC: uso $\mathcal{O}$ o $O$?#SSS: Josep?#JJJ: $\mathcal{O}$ with respect to the number of statements in #Deleted: a#Added: the program, but the traversal is linear with respect to the number of nodes in the graph (each corresponding to a statement) #SSS: footnote?.

The SDG is a directed graph, and as such it has vertices or nodes, each representing an #Deleted: instruction#Added: statement in the program —barring some auxiliary nodes introduced by some approaches— and directed edges, which represent the dependencies among nodes. Those edges represent various#SSS: several kinds of dependencies —control, data, calls, parameter passing, summary— which#JJJ: that are defined will be defined#SSS: further explained? in section 3.1. #CCC: add how a graph is sliced.

To create the SDG, first #JJJ: yo dejaria el a (como estaba)#Deleted: a#Added: the corresponding *control flow graph* (CFG) is built for each method in the program, then#Added: , its #Added: associated control and data dependencies are computed, resulting in #Added: a new graph representation known as the *program dependence graph* (PDG)#SSS: cita??#JJJ: si, a Ottenstein and Ottenstein#CCC: TENSTEIN, K. J., AND O'ITENSTEIN, L. M. The program dependence graph in a software development environment. Finally, all the graphs from every method are joined #CCC: NO by the appearance of a new kind of inter-procedural arcs, the argument-in argument-out arcs that link function definitions with function calls, obtaining#Deleted: into the #Added: final SDG. This process will be explained at greater lengths in section 3.1.

#CCC: falta mencionar el recorrido del grafo. An example #Added: of how an initial CFG is augmented and enhanced with all mentioned dependencies obtaining the corresponding PDG and the final SDG is provided in figure 2.1, where a #Added: the process is illustrated for a simple multiplication program#JJJ: pon el codigo del programa. asi pueden entender de que va esto los que no sepan de slicing. Sin el programa lo tienen mas complicado... Acabo de ver que ya esta el codigo. Entonces referencialo, presentalo: Consider the multiplication program in Figure X. The standard CFG and PDG generated for this code are... bla bla bla #Deleted: is converted to CFG, then PDG and finally SDG. For simplicity, #JJJ: quita el onlyonly the CFG and PDG of `main` are omitted#SSS: no entiendo esto de main. Donde esta main?. Control dependencies are #Added: represented with black #Added: arcs, data dependencies #Added:

with red #Added: arcs, and summary edges #Added: are depicted with blue #Added: arcs.

#SSS: nose si vale la pena poner la Figure 2.1 aqui, no hemos contado aun como se genera, sino que se genera y se supone que se cuenta mas adelante, tal vez sea mas util hacer referencia forward solo y no poner esta figura aqui, sino mas adelante. Plantearselo

### 2.1.2 Metrics

#SSS: Metrics o slicing indicators/features?

#JJJ: The main four metrics used to assess a program slicing algorithm are:There are four relevant metrics considered when evaluating a slicing algorithm: #SSS: Se me hace muy escueto esto, yo meteria algo de bullshit como dice Tama.

#SSS: PROPOSAL:

#SSS: In the area of program slicing, there are many different slicing techniques and tools implementing them. This fact has created the necessity to classify them by defining a set of different metrics. These metrics are commonly associated to some features of the generated slices. In the following, we list the most relevant metrics considered when evaluating a program slice:

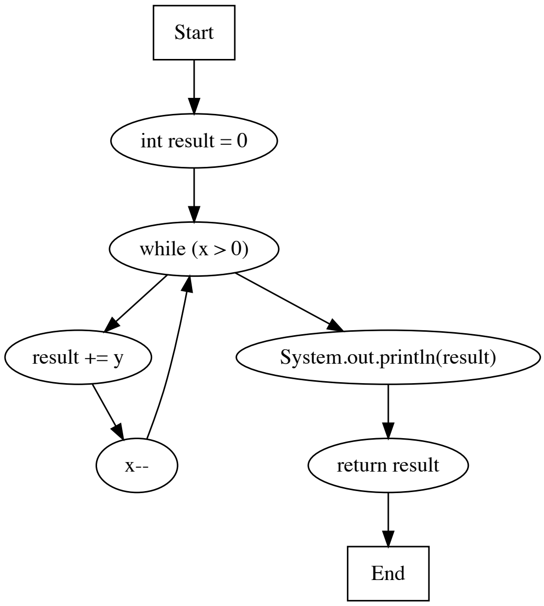
**Completeness.** The solution includes all the statements that affect the slicing criterion. This is the most important feature, and almost all publications#JJJ: techniques and implemented tools achieve at least completeness. Trivial completeness is easily achievable, as simple as including the whole program in the slice.

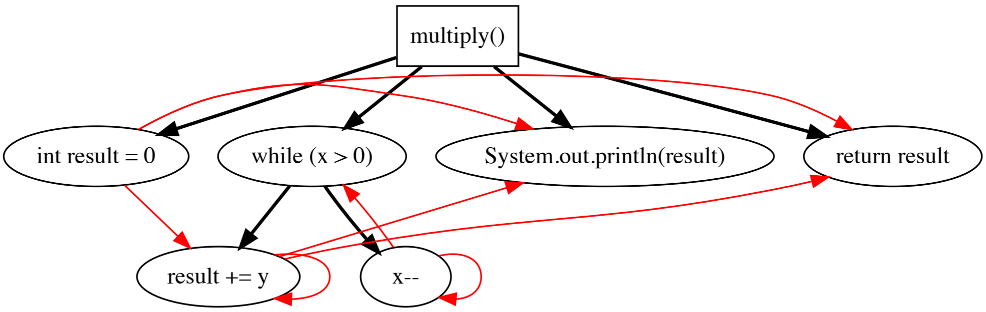
**Correctness.** The solution excludes all statements that do not affect the slicing criterion. Most solutions are complete, but the degree of correctness is what sets them apart, as solutions that are more correct will produce smaller slices, which will execute fewer instructions to compute the same values, decreasing the executing time and complexity.

**Features covered.** Which features #JJJ: (polymorphism, global variables, arrays, etc.) or language#JJJ: s/paradigms a slicing algorithm covers. Different approaches to slicing cover different programming languages and even paradigms. There are slicing techniques (published or commercially available) for most popular programming languages, from C++ to Erlang. Some slicing techniques only cover a subset of the targeted language, and as such are less useful for commercial applications, but can be a stepping stone in the betterment of the field.#SSS: Tambien estan las valen para todos los lenguajes, ORBS entraria en ese caso no Josep?#JJJ: si, hay algunas tecnicas que son independiente del paradigma, entre ellas ORBS. A cambio pagan un precio que suele ser una perdida de precision. Yo no me extenderia en ese tema, pero si estaria bien meter una cita a ORBS y sus semejantes al decir lo de even paradigms

**Speed.** Speed of graph generation and slice creation. As previously stated, slicing is a two-step process: building a graph and traversing it #SSS: esta frase hace parece que hacer slicing es dibujo libre... darle algo de importancia hablando de traducir el codigo a una representacion en forma de grafo con un estructura de datos compleja bla bla bla.... The traversal is a linear two–pass analysis of a graph in most proposals, with small variations. Graph generation tends to be a longer process, but it is not as relevant, because it is only done once (per program being analyzed), making this the least important metric. #SSS: Puedes anyadir que aunque la metrica del proceso de generacion no se suele tener muy en cuenta, esta existe porque es donde hay que hacer el analisis mas costoso sobre el programa y tal... relleno a saco! Que parece que no tiene ni merito generar el grafo :( Only proposals

```
int multiply(int x, int y) {
  int result = 0;
  while (x > 0) {
    result += y;
    x--;
  }
  System.out.println(result);
  return result;
}
```



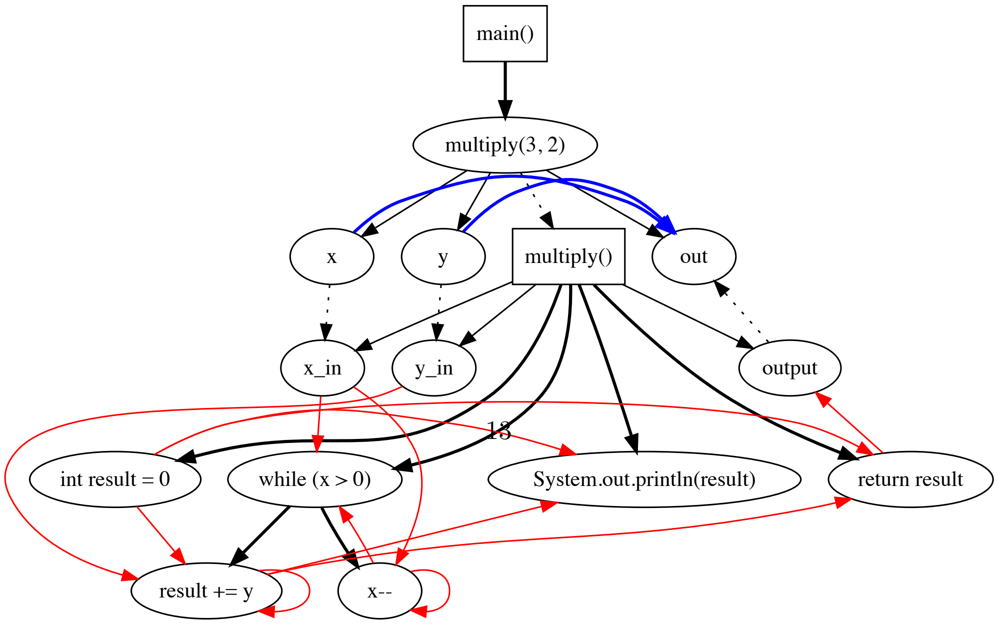






Figure 2.1: A simple multiplication program, its CFG, PDG and SDG

that deviate from the aforementioned schema of building a graph and traversing it show a wider variation in speed.

### 2.1.3 Program slicing as a debugging technique

#SSS: Soy pesado pero esto se me vuelve a hacer muy corto :/. Retoco esto un poco

#Added: As stated before, there are many uses for program slicing: program specialization, software maintenance, code obfuscation... but there is no doubt that p#Deleted: Program slicing is first and foremost a debugging technique#Added: . #Deleted: , having e#Added: Each#Deleted: variation#Added: configuration of different dimensions serves a different purpose:

**Backward static.** Used to obtain the lines that affect a statement, normally used on a line which outputs an incorrect value, to narrow#SSS: track? down the source of the bug.

**Forward static.** Used to obtain the lines affected by a statement, used to identify dead code, to check the effects a line has on the rest of the program.#JJJ: la principal aplicacion de forward slicing es software maintenance: Predecir a que partes del programa va a afectar un cambio. #CCC: https://ieeexplore.ieee.org/document/83912

**Chopping static.** Obtains both the statements affected by and the statements that affect the selected statement.

**Dynamic.** Can be combined with any of the previous variations, and limits the slice to an execution log#JJJ: history, only including statements that have run in a specific execution. The slice produced is much smaller and useful.

**Quasi–static.** #Added: In this slicing method s#Deleted: Some input values are given, and some are left unspecified: the result is a slice between the small dynamic slice and the general but bigger static slice. It can be specially useful when debugging a set of function calls which have a specific static input for some parameters, and variable input for others.

**Simultaneous.** Similar to dynamic slicing, but considers multiple executions instead of only one. Similarly to quasy–static slicing, it can offer a slightly bigger slice while keeping the scope focused on the source of the bug.

#CCC: añadir más quizá??? #SSS: a mi me parecen suficientes, puedes decir una frasecita de 2 o 3 lineas diciendo que hay mas y algun uso de alguno de los otros que queden asi a lo general, pero yo los veo suficientes. #JJJ: suficientes. Añade un párrafo diciendo que existen otras dimensiones que dan lugar a otras tecnicas y que en [16] se puede encontrar una análisis de las diferentes dimensiones que pueden usarse para clasificar tecnicas de slicing

## 2.2 Exception handling in Java

Exception handling is common in most modern programming languages. #Added: Exception handling generally consists in a set of statements that modify the normal execution flow noticing the existence of an abnormal program behaviour (controlled or not), and can be handled manually by the programmer or automatically by the system, depending on the programming language. In our work we focus on the Java programming language, so in the following, we describe the elements that Java uses to represent and handle exceptions: #Deleted: In Java, it consists of the following elements:

**Throwable.** An interface that encompasses all the exceptions or errors that may be thrown. Its child classes are `Error` for internal errors in the Java Virtual Machine and `Exception` for normal errors. Exceptions can be classified as *unchecked* (those that extend `RuntimeException`#SSS: se sale esto de la linea por el texttt or `Error`) and *checked* (all others, may inherit from `Throwable`, but typically they do so from `Exception`). The first kind may be thrown anywhere without warning, whereas the second, if thrown, must be either caught in the same method or declared in the method header.

**throws.** A statement that activates an exception, altering the normal control-flow of the method. If the statement is inside a *try* block with a *catch* clause for its type or any supertype, the control flow will continue in the first statement of such clause. Otherwise, the method is exited and the check performed again, until either the exception is caught or the last method in the stack (*main*) is popped, and the execution of the program ends abruptly.

**try.** This statement is followed by a block of statements and by one or more *catch* clauses. All exceptions thrown in the statements contained or any methods called will be processed by the list of catches. Optionally, after the *catch* clauses a *finally* block may appear.

**catch.** Contains two elements: a variable declaration (the type must be an exception #SSS: exception o exception type?) and a block of statements to be executed when an exception of the corresponding type (or a subtype) is thrown. *catch* clauses are processed sequentially, and if any matches the type of the thrown exception, its block is executed, and the rest are ignored. Variable declarations may be of multiple types `(T1|T2 exc)`, when two unrelated types of exception must be caught and the same code executed for both. When there is an inheritance relationship, the parent suffices.[1]

**finally.** Contains a block of statements that will always be executed if the *try* is entered. It is used to tidy up, for example closing I/O streams. The *finally* can be reached in two ways: with an exception pending (thrown in *try* and not captured by any *catch* or thrown inside a *catch*) or without it (when the *try* or *catch* block end successfully). After the last instruction of the block is executed, if there is an exception pending, control will be passed to the corresponding *catch* or the program will end. Otherwise, the execution continues in the next statement after the *try-catch-finally* block.

#SSS: Me han molao las explicaciones, se entiende muy bien como funciona Java, parece que sea hasta facil de usar :D

### 2.2.1 Exception handling in other programming languages

In almost all programming languages, errors can appear (either through the developer, the user or the system's fault), and must be dealt with. Most of the popular object–oriented programs feature some kind of error system, normally very similar to Java's exceptions. In this section, we will perform a small survey of the error-handling techniques used on the most popular programming languages. The language list has been extracted from a survey performed by the programming Q&A website Stack Overflow[2]. The survey contains a question about the technologies used by professional developers in their work, and from that list we have extracted those languages with more than 5% usage in the industry. Table 2.2 shows the list and its source. Except Bash, Assembly, VBA, C and G,#SSS: Bash y companyia no tienen mecanismo de exception handling?

[1]Introduced in Java 7, see `https://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html` for more details.

[2]`https://stackoverflow.com`

| Language | % usage |
|---|---|
| JavaScript | 69.7 |
| HTML/CSS | 63.1 |
| SQL | 56.5 |
| Python | 39.4 |
| Java | 39.2 |
| Bash/Shell/PowerShell | 37.9 |
| C# | 31.9 |
| PHP | 25.8 |
| TypeScript | 23.5 |
| C++ | 20.4 |

| Language | % usage |
|---|---|
| C | 17.3 |
| Ruby | 8.9 |
| Go | 8.8 |
| Swift | 6.8 |
| Kotlin | 6.6 |
| R | 5.6 |
| VBA | 5.5 |
| Objective-C | 5.2 |
| Assembly | 5.0 |

Table 2.2: The most commonly used programming languages by professional developers[3]

o no se parece al de Java? No queda claro en esta frase the rest of the languages shown feature an exception system similar to the one appearing in Java.

The exception systems that are similar to Java are mostly all the same, featuring a `throw` statement (`raise` in Python), try-catching structure and most include a finally block that may be appended to try blocks. The difference resides in the value passed by the exception, which in languages that feature inheritance it is a class descending from a generic error or exception, and in languages without it#SSS: este “it” se refiere a inheritance? pon algun objeto y elimina algun it porque hay muchos y me lian xD, it is an arbitrary value (e.g. JavaScript, TypeScript). In object–oriented programming, the filtering is performed by comparing if the exception is a subtype of the exception being caught (Java, C++, C#, PowerShell[4], etc.); and in languages with arbitrary exception values, a boolean condition is specified, and the first catch block that fulfills its condition is activated, in following#SSS: in following o following? a pattern similar to that of `switch` statements (e.g. JavaScript). In both cases there exists a way to indicate that all exceptions should be caught, regardless of type and content.

On the other hand, in #Deleted: the other languages #SSS: “the other languages” es muy vago#Added: those languages that do not offer explicit exception handling mechanisms, #Deleted: there exist a variety of systems that emulate or replace exception handling:#Added: this feature is covered by a variety of systems that emulate or replace their behaviour:

**Bash.** The popular Bourne Again SHell features no exception system, apart from the user's ability to parse the return code from the last statement executed. Traps can also be used to capture erroneous states and tidy up all files and environment variables before exiting the program. Traps allow the programmer to react to a user or system–sent signal, or an exit run from within the Bash environment. When a trap is activated, its code run, and the signal does not proceed and stop the program. This does not replace a fully featured exception system, but `bash` programs tend to be short, with programmers preferring the efficiency of C or the commodities of other high–level languages when the task requires it.

**VBA.** Visual Basic for Applications is a scripting programming language based on Visual Basic that is integrated into Microsoft Office to automate small tasks, such as generating documents from templates, making advanced computations that are impossible or slower with spreadsheet functions, etc. The only error–correcting system it has is the directive

[3]Data from `https://insights.stackoverflow.com/survey/2019/#technology-_-programming-scripting-and-markup-languages`

[4]Only since version 2.0, released with Windows 7.

On Error $x$, where $x$ can be 0 —lets the error crash the program—, Next —continues the execution as if nothing had happened— or a label in the program —the execution jumps to the label in case of error. The directive can be set and reset multiple times, therefore creating artificial try-catch blocks, but there is no possibility of attaching a value to the error, lowering its usefulness.

**C.** In C, errors can also be controlled via return values, but some instructions featured in it can be used to create a simple exception system. setjmp and longjmp are two instructions which set up and perform inter–function jumps. The first makes a snapshot of the call stack in a buffer, and the second returns to the position where the buffer was safe, destroying the current state of the stack and replacing it with the snapshot. Then, the execution continues from the evaluation of setjmp, which returns the second argument passed to longjmp.

**Example 4** (User-built exception system in C)**.**

```
1 int main() {
2   if (!setjmp(ref)) {
3     res = safe_sqrt(x, ref);
4   } else {
5     // Handle error
6     printf /* ... */
7   }
8 }
```

```
1 double safe_sqrt(double x, int ref) {
2   if (x < 0)
3     longjmp(ref, 1);
4   return /* ... */;
5 }
```

In the main function, line 2 will be executed twice: first when it is normally reached —returning 0 and continuing in line 3— and the second when line 3 in safe_sqrt is run, returning the second argument of longjmp, and therefore entering the else block in the main method.

**Go.** The programming language Go is the odd one out in this section, being a modern programming language without exceptions, though it is an intentional design decision made by its authors[5]. The argument made was that exception handling systems introduce abnormal control–flow and complicate code analysis and clean code generation, as it is not clear the paths that the code may follow. Instead, Go allows functions to return multiple values, with the second value typically associated to an error type. The error is checked before the value, and acted upon. Additionally, Go also features a simple panic system, with the functions panic —throws an exception with a value associated—, defer —runs after the function has ended or when a panic has been activated— and recover —stops the panic state and retrieves its value. The defer statement doubles as catch and finally, and multiple instances can be accumulated. When appropriate, they will run in LIFO#Deleted: order (Last In–First Out) #Added: order.

**Assembly.** Assembly is a representation of machine code, and each computer architecture has its own instruction set, which makes an analysis impossible. In general, though, no unified exception handling is provided. #CCC: complete with more info on kinds of error handling at the processor level or is this out of scope???#SSS: Si metes una explicacion asi breve que se entienda bien, si va a ser muy tecnico yo pararia aqui. Diria que las excepciones se manejan a nivel de procesador o lo que sea asi por encima y matizao

[5] For more details on Go's design choices, see https://golang.org/doc/faq#exceptions. #CCC: Possible transformation to citation???#SSS: No creo que nos vaya a hacer falta. Con el state of the art y la intro tendremos bastantes.#JJJ: mantenlo como footnote

# Chapter 3

# Main explanation?

#CCC: Review if we want to call nodes “Enter” and “Exit” or “Start” and “End” (I’d prefer the first one). #SSS: Enter o Entry? #JJJ: No es una decision nuestra, coge la misma palabra que Orwitz en el paper del SDG

## 3.1 First definition of the SDG

The system dependence graph (SDG) is #Deleted: a method#Added: the main data structure for program representation used in the#Deleted: for program slicing#Added: area. It#Deleted: that was first proposed by Horwitz, Reps and Blinkey [6]#Added: and, since then, many approaches have based their models on it. It builds upon the existing control flow graph (CFG), defining dependencies between vertices of the CFG, and building a program dependence graph (PDG), which represents them.#SSS: Volvemos a poner las siglas y su significado?CFG?PDG? ya se han puesto antes The #Deleted: system dependence graph (SDG#Deleted: ) is then built from the assembly of the different PDGs (each representing a method of the program), linking each method call to its corresponding definition. Because each graph is built from the previous one, new constructs can be added with to the CFG, without the need to alter the algorithm that converts #Added: each CFG to PDG and then to #Added: the final SDG. The only modification possible is the redefinition of a#Added: n already defined dependency or the addition of new kinds of dependence.

The language covered by the initial proposal #Deleted: was#Added: is#SSS: todo en presente o todo en pasado a simple one, featuring procedures with modifiable parameters and basic instructions, including calls to procedures, variable assignments, arithmetic and logic operators and conditional instructions (branches and loops)#Deleted: :#Added: , i.e.,#SSS: no se si i.e., queda bien aqui :/ the basic features of an imperative programming language. The #Deleted: control flow graph was#Added: CFGs are as simple as the programs themselves, with each graph representing one procedure. The instructions of the program are represented as vertices of the graph and are split into two categories: statements, which have no effect on the control flow (#Added: e.g., assignments, procedure calls) and predicates, whose execution may lead to one of multiple —though traditionally two— #Added: different paths (#Added: e.g., conditional instructions). #Deleted: S#Added: While statements are connected sequentially to the next instruction#Deleted: . P#Added: , on the contrary, predicates have two outgoing edges, each#Added: of them connected to the first statement that should be executed#Deleted: , according to the result of evaluating the conditional expression in the guard of the predicate.

**Definition 3** (Control Flow Graph #CCC: add original citation)**.** A *control flow graph* $G$ of a program#SSS: program o method? $P$ is a directed graph, represented as a tuple $\langle N, E \rangle$, where $N$ is a set of nodes #JJJ: such that for each statement $s$ in $P$there is a node in $N$ labeled with $S$ and there are two special nodes..., composed of a method's#SSS: method o program? statements plus two special nodes, "Start" and "End"; and $E$ is a set of edges of the form $e = (n_1, n_2) \,|n_1, n_2 \in N$. #JJJ: Esto es una definicion. No pueden haber opinion ni contenido vago. O defines que Start y End son nodos o no lo defines. Pero no diugas lo que han hecho otros en una definicion. Lo que sigue yo lo quitaría Most algorithms#Added: , in order to generate the SDG#Added: , mandate the "Start" node to be the only source and #Added: the "End" #Added: node to be the only sink in the graph. #CCC: Is it necessary to define source and sink in the context of a graph?#JJJ: quitalo.

#JJJ: desde aquiEdges are created according to the possible execution paths that exist; each statement is connected to any statement that may immediately follow it. Formally, #JJJ: hasta aqui sacalo fuera de la definicion, para explicarla., Pero no tiene sentido que digas algo informal en una definicion y dentro incluso de la definicion digas formally, Debe ser TODO formally por definicion (valga la redundancia)an edge $e = (n_1, n_2)$ exists if and only if there exists an execution of the program where $n_2$ is executed immediately after $n_1$. #JJJ: de nuevo, no puedes decir in general. O defines que si se evaluan o que no, pero no digas lo que se suele hacer. Aqui estas definiendoIn general, expressions are not evaluated#Added: when generating the CFG; so a#Deleted: n `if`#Added: conditional instruction #Added: will have#Deleted: has two outgoing edges #Added: regardless the condition value being #Deleted: even if the condition is always true or false, e.g.#Added: , `1 == 0`.

To build the PDG and then the SDG, there are two dependencies based directly on the CFG's structure: data and control dependence. #SSS: But first, we need to define the concept of postdominance in a graph necessary in the definition of control dependency:#SSS: no me convence mucho pero plantearse si poner algo aqui o dejarlo como esta.

**Definition 4** (Postdominance #CCC: add original citation?)**.** #JJJ: Let $C = (N, E)$ be a CFG. Vertex $b$#$JJJ :\in N$ *postdominates* vertex $a$#$JJJ :\in N$ if and only if $b$ is on every path from $a$ to the "End" vertex.

**Definition 5** (Control dependency#SSS: dependency o dependence? #CCC: add original citation)**.** #JJJ: Let $C = (N, E)$ be a CFG. Vertex $b$#$JJJ :\in N$ is *control dependent* on vertex $a$#$JJJ :\in N$ ($a \rightarrow^{ctrl} b$) if and only if $b$ postdominates one but not all of $a$'s successors. #JJJ: Lo que sigue es en realidad es un lema. No hace falta ponerlo como lema, pero sí sacarlo a después de la definicion.It follows that a vertex with only one successor cannot be the source of control dependence.

**Definition 6** (Data dependency#SSS: dependency o dependence? #CCC: add original citation)**.** #JJJ: Let $C = (N, E)$ be a CFG. Vertex $b$#$JJJ :\in N$ is *data dependent* on vertex $a$#$JJJ :\in N$ ($a \rightarrow^{data} b$) if and only if $a$ may define a variable $x$, $b$ may use $x$ and there exists a #CCC: could it be "an"?? $x$-definition free path from $a$ to $b$.

Data dependency was originally defined as flow dependency, and split into loop and non–loop related dependencies#JJJ: creo que es loop-carried. Me parece que esta en el paper de Frank Tip, but that distinction is no longer useful to compute program slices #SSS: Quien dijo que ya no es util? Vale la pena citarlo?. #JJJ: Si que es useful en program slicing, pero no en debugging. It should be noted that variable definitions and uses can be computed for each statement independently, analysing the procedures called by it if necessary. The variables used and defined by a procedure call are those used and defined by its body.

With the data and control dependencies, the PDG may be built by replacing the edges from the CFG by data and control dependence edges. The first tends to be represented as a thin solid line, and the latter as a thick solid line. In the examples, #Added: data and control dependencies are represented by thin solid red and black lines respectively#Deleted: data dependencies will be thin solid red lines.

**Definition 7** (Program dependence graph)**.** #JJJ: Given a program $P$, The *program dependence graph* (PDG) #JJJ: associated with $P$ is a directed graph (and originally a tree#SSS: ???#JJJ: sobran las aclaraciones historicas en una definicion) represented by #JJJ: a triple $\langle N, E_c, E_d \rangle$ where $N$ is... three elements: a set of nodes $N$, a set of control edges $E_c$ and a set of data edges $E_d$. #SSS: $PDG = \langle N, E_c, E_d \rangle$

Method $M$, CFG $C = \langle N, E \rangle$, the PDG is $P = \langle N', E_c, E_d \rangle$, where

1. $N' = N \backslash \{End\}$
2. $(a, b) \in E_c \iff a, b \in N' \wedge a \rightarrow^{ctrl} b \wedge \nexists c \in N' \ . \ a \rightarrow^{ctrl} c \wedge c \rightarrow^{ctrl} b$
3. $(a, b) \in E_d \iff a, b \in N' \wedge a \rightarrow^{data} b$

The set of nodes corresponds to the set of nodes of the CFG#JJJ: que CFG? no se puede dar por hecho que existe un CFG en una definicion, excluding the "End" node.

Both sets of edges are built as follows#JJJ: :. There is a control edge between two nodes $n_1$ and $n_2$ if and only if $n_1 \rightarrow^{ctrl} n_2$#SSS: acordarse de lo de evitar la generacion de arcos para prevenir la transitividad. Decidir si definimos Control arc como ua definicion aparte., and a data edge between $n_1$ and $n_2$ if and only if $n_1 \rightarrow^{data} n_2$. Additionally, if a node $n$ does not have any incoming control edges, it has a "default" control edge $e = (\text{Start}, n)$; so that "Start" is the only source node of the graph.

Note: #JJJ: dentro de una definicion no pueden haber notas. Esto va fuerathe most common graphical representation is a tree–like structure based on the control edges, and nodes sorted left to right according to their position on the original program. Data edges do not affect the structure, so that the graph is easily readable.

#SSS: creo que en la definicion de CFG y PDG tiene que quedar mas claro que hay varios por programa (uno por funcion), para que esta ultima frase cobre mas sentido.

Finally, the SDG is built from the combination of all the PDGs that compose the program.

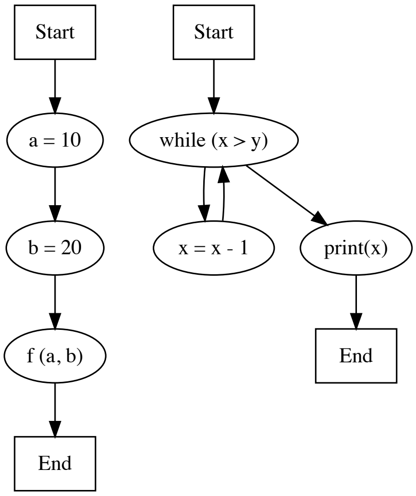
**Definition 8** (System dependence graph)**.** Given a program $P$ composed of a set of $n$ methods $M = \{m_0 ... m_n\}$ and their associated PDGs (each method $m_i$ has a PDG $G^i_{PDG} = \langle N^i, E^i_c, E^i_d \rangle$), the *system dependence graph* (SDG) of $P$ is a graph $G = \langle N', E'_c, E'_d, E_{fc}, E_s \rangle$ where $N = \bigcup_{i=0}^{n} N^i$, , , , and .

#JJJ: Arreglar esta definicion como la del PDG. Ahora mismo es totalmente informal. Deberia definirse encima del PDG. Es decir, una SDG es la conexion adecuada de varios PDGs, uno por método. Y solo definir lo nuevo: call arcs, parameter-in arcs, parameter-out arcs y summary arcs. The *system dependence graph* (SDG) is a directed graph that represents the control and data dependencies of a whole program. It has three kinds of edges: control, data and function call. The graph is built combining multiple PDGs, with the "Start" nodes labeled after the function they begin. There exists one function call edge between each node containing one or more calls and each of the "Start" node#JJJ: s of the method called. In a programming language where the function call is ambiguous (e.g. with pointers or polymorphism), there exists one edge leading to every possible function called.#SSS: Esta definicion ha quedado muy informal no? Donde han quedado los $E_c$, $E_d$, $E_{fc}$, Nodes del PDG...?

**Example 5** (Creation of a SDG from a simple program)**.** Given the program shown below (left), the control flow graphs for both methods are shown on the right:

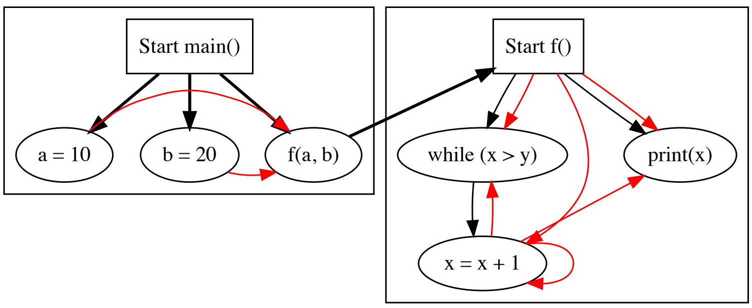
```
proc main() {
  a = 10;
  b = 20;
  f(a, b);
}

proc f(x, y) {
  while (x > y) {
    x = x - 1;
  }
  print(x);
}
```




#SSS: Centrar la figura, sobra mucho espacio a la derecha

Then, control and data dependencies are computed, arranging the nodes in the #JJJ: corresponding PDG#JJJ: s (see the two PDGs inside the two squares below)#SSS: FigureRef missing. Finally, the two graphs are connected with summary edges#SSS: with que? esto no se sabe aun ni lo que es ni para que sirve. En todo caso function call edges, y si ese es el negro que va de f(a,b) a Start f() para diferenciarlo deberia ser de otro color to create the SDG:




**Function calls and data dependencies**

#CCC: Vocabulary: when is appropriate the use of method, function and procedure????#SSS: buena pregunta, yo creo que es jerarquico, method incluye function y procedure y los dos ultimos son disjuntos entre si no? #JJJ: No. metodo implica orientacion a objetos. si estas hablando de un lenguaje en particular (p.e., Java), entonces debes usar el vocabulario de ese lenguaje (p.e., method). Si hablas en general y quieres usar una palabra que subsuma a todos, yo he visto dos maneras de hacerlo: (1) usar routine (aunque podrias usar otra palabra, por ejemplo metodo) la primera vez y ponerle una footnote diciendo que en el resto del articulo usamos routine para referirnos a metodo/funcion/procedimiento/predicado. (2) Usar metodo/funcion/procedimiento/predicado así, separado por barras. En esta tesina parece mas apropiado hablar de metodo, y la primera vez poner una footnote que diga que hablaremos de métodos, pero todos los desarrollos son igualmente aplicables a funciones y procedimientos.

In the original definition of the SDG, there was special handling of data dependencies when calling functions, as it was considered that parameters were passed by value, and global variables did not exist. #CCC: Name and cite paper that introduced it solves this issue by splitting function calls and function #Added: definitions into multiple nodes. This proposal solved #JJJ: the problemeverything#SSS: lo resuelve todo? related to parameter passing: by value, by reference, complex variables such as structs or objects and return values.

To such end, the following modifications are made to the different graphs:

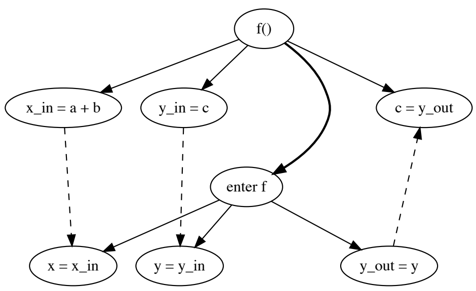
**CFG.** In each CFG, global variables read or modified and parameters are added to the label of the "Start" node in assignments of the form $par = par_{in}$ for each parameter and $x = x_{in}$ for global variables. Similarly, global variables and parameters modified are added to the label of the "End" node as #Added: assignments of the form $x_{out} = x$. #Added: From now on, we will refer to the described assignments as input and output information respectively. #SSS: {The parameters are only passed back if the value set by the called method can be read by the callee#SSS: } no entiendo a que se refiere esta frase. Finally, in method calls the same values must be packed and unpacked: each statement containing a function called is relabeled to contain #Added: its related input (of the form $par_{in} = \exp$ for parameters or $x_{in} = x$ for global variables) and output (always of the form $x = x_{out}$) #Added: information. #SSS: no hay parameter_out? asumo entonces que no hay paso por valor?

**PDG.** Each node #Added: augmented with input or output information#Deleted: modified in the CFG is #Added: now split into multiple nodes: the original #Deleted: label#Added: node #Added: (Start, End or function call) is the main node and each assignment #Added: contained in the input and output information is represented as a new node, which is control–dependent on the main one. Visually, #Added: new nodes coming from the input information#Deleted: input is #Added: are placed on the left and #Added: the ones coming from the output information#Deleted: output on the right; with parameters sorted accordingly.

**SDG.** Three kinds of edges are introduced: parameter input (param–in), parameter output (param–out) and summary edges. Parameter input edges are placed between each method call's input node and the corresponding method definition input node. Parameter output edges are placed between each method definition's output node and the corresponding method call output node. Summary edges are placed between the input and output nodes of a method call, according to the dependencies inside the method definition: if there is a path from an input node to an output node, that shows a dependence and a summary method is placed in all method calls between those two nodes.#SSS: Tengo la sensacion de que la explicacion de que es un summary llega algo tarde y tal vez deberia estar en alguna definicion previa. Que opine Josep que piensa#JJJ: Efectivamente. Llega tarde. No pueden definirse estas dependencias despues de definir el SDG, porque entonces lo que has definido en la definicion formal no es un SDG (solo una parte de el) y cuando hables de SDG a partir de ahora todo estara incompleto. Las definiciones son sagradas, así que hay dos soluciones: (1) explicar estos tres arcos antes de la definicion de SDG para poder definirlos formalmente en la definicion de SDG, o (2) retrasar la definiucion formal de SDG hasta aqui (para poder incluirlos). O cualquier otra cosa que haga que el SDG esté bien definido

Note: #Deleted: parameter input and output#Added: param-in and param-out edges are separated because the traversal algorithm traverses them only sometimes (the output edges are excluded in the first pass and the input edges in the second).#SSS: delicado mencionar lo de las pasadas sin haber hablado antes de nada del algoritmo de slicing, a los que no sepan de slicing se les quedara el ojete frio aqui. Plantearse quitar esta nota.#JJJ: Esta nota retrasala hasta que hables del algoritmo de slicing. En ese momento puedes decir que precisamente para que hayan dos pasadas se distingue entre parameter-ín y paramneter-out. Alli tendrá sentido y será aclaratorio. Aquí es confusorio. ;-)

**Example 6** (Variable packing and unpacking)**.** Let it be #JJJ: Excelente cancion de los beatles. Buenísima. Pero mejor empieza así: Let $f(x, y)$ be a function with... ;-) a function $f(x, y)$ with two integer parameters #Added: which#JJJ: that modifies the argument passed in its second parameter, and a call $f(a+b, c)$, with parameters passed by reference if possible. The label of the method call node in the CFG would be "`x_in = a + b, y_in = c, f(a + b, c)`#JJJ: ???, `c = y_out`"; method $f$ would have `x = x_in, y = y_in` in the "Start" node and `y_out = y` in the "End" node. The relevant section of the SDG would be: #JJJ: Todo este parrafo y la figura que sigue no se entienden. Hay que reescribirlo y explicarlo más detenidamente, paso a paso. Se supone que este es el ejmplo de la sección. El que va a aclarar las dudas de qué es $x_in$, etc. y de cómo funciona el SDG. Sin embargo, más que aclarar, lía (a uno que no sepa de slicing no le aclara nada). De hecho, para que se entendiera bien, una vez has construido el grafo, estaría bien continuar un poco el ejemplo explicando como las dependencias hacen que lo que hay dentro del método llamado depende (siguiendo los arcos) de lo que hay en el método llamador (o al menos de los parámetros de la llamada). Esto requiere un poco de texto explicativo.




#SSS: Esta figura molaria mas evolutiva si diera tiempo, asi seria casi autoexplicativa: CFG → PDG → SDG. La actual seria el SDG, las otras tendrian poco mas que un nodo y una etiqueta.

## 3.2 Unconditional control flow

Even though the initial definition of the SDG was #Deleted: useful#Added: adequate to compute slices, the language covered was not enough for the typical language of the 1980s, which included (in one form or another) unconditional control flow. Therefore, one of the first #Added: proposed upgrades#Deleted: additions contributed to the algorithm to build #Deleted: system dependence graphs#Added: SDGs was the inclusion of unconditional jumps, such as "break", "continue", "goto" and "return" statements (or any other equivalent). A naive representation would be to treat them the same as any other statement, but with the outgoing edge landing in the corresponding instruction (outside the loop, at the loop condition, at the method's end, etc.). An alternative approach is to represent the instruction as an edge, not a vertex, connecting the previous statement with the next to be executed. #SSS: Juntaria las 2 propuestas anteriores (naive y alternative) en 1 frase, no las separaria, porque despues de leer la primera ya me he mosqueado porque no deciamos ni quien la hacia ni por que no era util. Both of these approaches fail to generate a control dependence from the unconditional jump, as the definition of control dependence (see definition 5) requires a vertex to have more than one successor for it to be possible to be a source of control dependence. From here, there stem two approaches: the first would be to redefine control dependency, in order to reflect the real effect of these instructions —as some authors [5] have tried to do— and the second would be to alter the creation of the SDG to "create" those dependencies, which is the most widely–used solution [3].

The most popular approach was proposed by Ball and Horwitz [3], classifying instructions into three separate categories:

**Statement.** Any instruction that is not a conditional or unconditional jump. #JJJ: #Deleted: It has one outgoing edge in the CFG, to the next instruction that follows it in the program.#Added: Those nodes that represent an statement in the CFG have one outgoing edge pointing to the next instruction that follows it in the program.

**Predicate.** Any conditional jump instruction, such as `while`, `until`, `do-while`, `if`, etc. #JJJ: #Deleted: It has two outgoing edges, labeled *true* and *false*; leading to the corresponding instructions.#Added: In the CFG, those nodes representing predicates have two outgoing edges, labeled *true* and *false*, leading to the corresponding instructions.

**Pseudo–predicates.** Unconditional jumps (e.g. `break`, `goto`, `continue`, `return`); are like predicates, with the difference that the outgoing edge labeled *false* is marked as non–executable#JJJ: —because there is no possible execution where such edge would be possible,#Deleted: , and there is no possible execution where such edge would be possible, according to the definition of the CFG (see Definition ??)—. Originally the edges had a specific reasoning backing them up: the *true* edge leads to the jump's destination and the *false* one, to the instruction that would be executed if the unconditional jump was removed, or converted into a `no op`#SSS: no op o no-op? (a blank operation that performs no change to the program's state). #SSS: {This specific behavior is used with unconditional jumps, but no longer applies to pseudo–predicates, as more instructions have used this category as means of "artificially" #CCC: bad word choice generating control dependencies.#SSS: }No entrar en este jardin, cuando se definio esto no se contemplaba la creacion de nodos artificiales. -Quita el originally, ahora es originally.

#CCC: Pseudo–statements now have been introduced and are used to generate all control edges (for now just the Start method to the End).#JJJ: No entiendo este CCC

As a consequence of this classification, every instruction after an unconditional jump $j$ is control–dependent (either directly or indirectly) on $j$ and the structure containing it (#JJJ: a predicate such as a conditional statement or a loop), as can be seen in the following example.

**Example 7** (Control dependencies generated by unconditional instructions)**.** Figure 3.1 showcases a small program with a `break` statement, its CFG and PDG with a slice in grey#JJJ: No hables aún del slice. Primero presenta el programa, luego los grafos, luego el CS y finalmente el slice. The slicing criterion (line 5, variable $a$) is control dependent on both the unconditional jump and its surrounding conditional instruction (both on line 4#JJJ: ponlos en lineas diferentes)#JJJ: . Therefore, the slice (all nodes in grey) includes the conditional jump and also the conditional exception. Note however that...; even though it is not necessary to include it#SSS: a quien se refiere este it? (in the context of weak slicing).

```java
static void f() {
  int a = 1;
  while (a > 0) {
    if (a > 10) break;
    a++;
  }
  System.out.println(a);
}
```

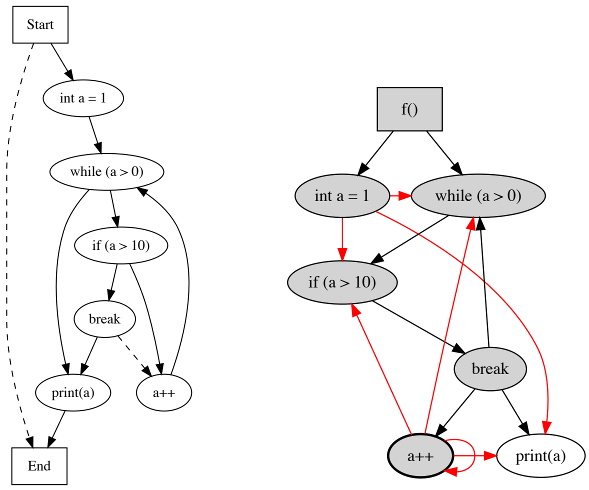



Figure 3.1: A program with unconditional control flow, its CFG (center) and PDG(right).

Note: the “Start” node $S$ is also categorized as a pseudo–statement, with the *false* edge connected to the “End” node, therefore generating a dependence from $S$ to all the nodes inside the method. This removes the need to handle $S$ with a special case when converting a CFG to a PDG, but lowers the explainability of non–executable edges as leading to the “instruction that would be executed if the node was absent or a no–op”.

The original paper#JJJ: que original paper? parece que hablas de alguno que hayas hablado antes, pero el lector ya no se acuerda. Empieza de otra manera... [3] does prove its completeness, but disproves its correctness by providing a counter–example similar to example 8. This proof affects both weak and strong slicing, so improvements can be made on this proposal. The authors postulate that a more correct approach would be achievable if the slice’s restriction of being a subset of instructions were lifted.

**Example 8** (Nested unconditional jumps)**.** #JJJ: Esta frase es dificil de leer. No se entiende hasta leerla dos o tres veces.In the case of nested unconditional jumps where both jump to the same destination, only one of them (the out–most one) is needed #JJJ: El lector no tiene contexto para saber de que hablas. Mejor empieza al reves: Consider the program in Figure 3.2 where we can observe two nested unconditional jumps in lines X and Y. If we slice this program using the dependencies computed according to [] then we compute the slice in light blue. Nevertheless, the minimal slice is composed of the nodes in grey [NOTA: yo no veo los colores. Arreglar la frase si no coincide con los colores]. This means that the slice computed includes unnecessary code (lines 3 and 5 are included unnecessarily). This problem is explained in depth and a solution proposed in Section **??**. Figure 3.2 showcases the problem, with the minimal slice #CCC: have not defined this yet in grey, and the algorithmically computed slice in light blue. Specifically, lines 3 and 5 are included unnecessarily.

#CCC: Add proposals to fix both problems showcased.

```
while (X) {
  if (Y) {
    if (Z) {
      A;
      break;
    }
    B;
    break;
  }
  C;
}
D;
```

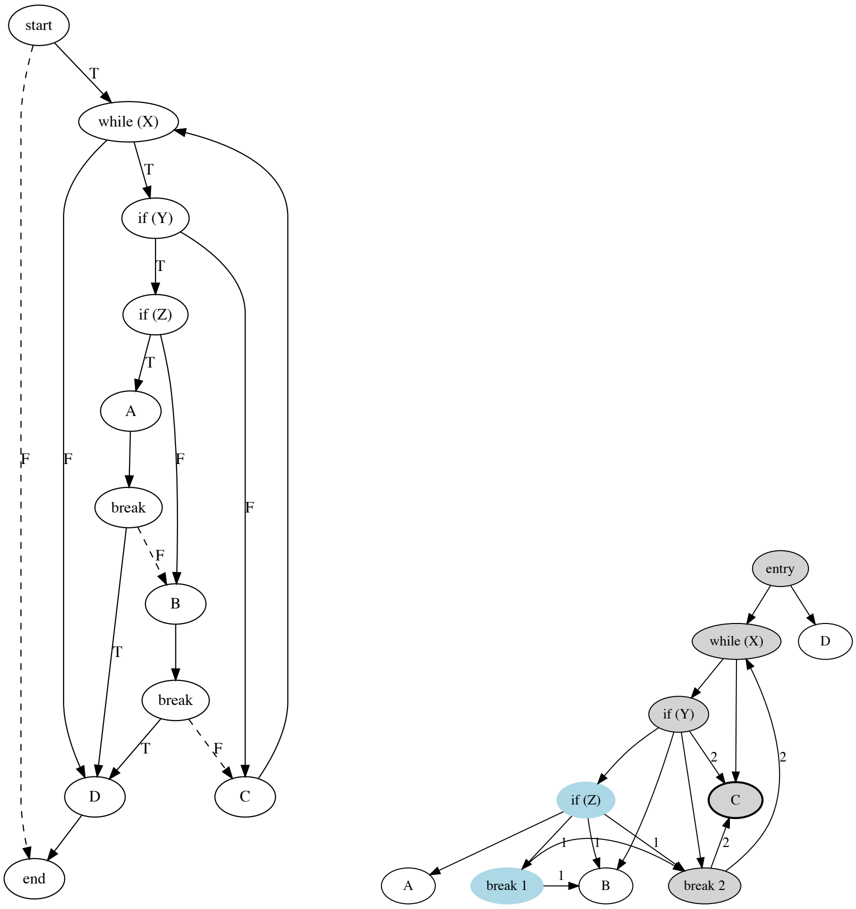



Figure 3.2: A program with nested unconditional control flow (left), its CFG (center) and #JJJ: its PDG (right).

## 3.3 Exceptions

#SSS: Creo que aun no hemos dicho que nuestro target language es Java, creo que ahora seria un buen momento.

Exception handling was first tackled in the context of Java program slicing by Sinha et al. [15], with later contributions by Allen and Horwitz [2]. There exist contributions for other programming languages, which will be explored later (chapter 5) #Deleted: and other small contributions. #SSS: Tal vez cambiaria el orden de estas frases para ir de lo general a lo concreto, diria primero que hay muchas contribuciones que veremos en el chapter 5 y luego que nos vamos a centrar en los planteamientos que abordan el problema para Java, donde las propuestas con mas peso son: tal y tal. The following section will explain the treatment of the different elements of exception handling in Java program slicing.

As seen in section 2.2, exception handling in Java adds two constructs: `throw` and `try-catch`. Structurally, the first one resembles an unconditional control flow statement carrying a value —like `return` statements— but its destination is not fixed, as it depends on the dynamic typing of the value. If there is a compatible `catch` block, execution will continue inside it, otherwise the method exits with the #Deleted: corresponding value as the error #Added: as returned value. The same process is repeated in the method that called the current one, until either the call stack is emptied or the exception is successfully caught. #Deleted: If#Added: Eventually, in case the exception is not caught #Deleted: at all#Added: by any stacked method, the program exits with an error —except in multi–threaded programs, in which case the corresponding thread is terminated. The `try-catch` statement can be compared to a `switch` which compares types (with `instanceof`) instead of constants (with `==` and `Object#equals(Object)` #SSS: esta notacion es obligatoria o podemos decir "... and the `equals` operands"?). Both structures require special handling to place the proper dependencies, so that slices are complete and as correct as #Deleted: can be#Added: possible.

### 3.3.1 `throw` statement

The `throw` statement compounds two elements in one instruction: an unconditional jump with a value attached and a switch to an "exception mode", in which the statement's execution order is disregarded. The first one has been extensively covered and solved; as it is equivalent to the `return` instruction, but the second one requires a small addition to the CFG: there must be an alternative control flow, where the path of the exception is shown. For now#SSS: esto suena muy espanyol no? So far?, without including `try-catch` structures, any exception thrown will exit its method with an error; so a new "Error end" node is needed.#SSS: No me convence esta frase, a ver como os suena esto (aunque no estoy muy convencido de ello) → So far, without including `try-catch` structures, any exception thrown would activate the mentioned "exception mode" and leave its method with an error state. Hence, in order to represent this behaviour, a different exit point (represented with a node called "Error end") need to be defined. #Deleted: T#Added: Consecuently, the pre-existing "End" node is renamed #Added: as "Normal end", #Deleted: but now the#Added: leaving the CFG #Deleted: has#Added: with two distinct sink nodes; which is forbidden in most slicing algorithms. To solve that problem, a general "End" node is created, with both normal and #Deleted: exit#Added: error ends connected to it; making it the only sink in the graph.

In order to properly accommodate a method's output variables (global variables or parameters passed by reference that have been modified), variable unpacking is added to the "Error exit" node; same as the "Exit"#SSS: Exit?End?Vaya cacao llevamos con esto xD node in previous examples. This change constitutes an increase in precision, as now the outputted variables are differentiated#Deleted: ; f#Added: . For example#Added: , a slice which only requires the error exit may include less variable modifications than one which includes both.

This treatment of `throw` statements only modifies the structure of the CFG, without altering the other graphs, the traversal algorithm, or the basic definitions for control and data dependencies. That fact makes it easy to incorporate to any existing program slicer that follows the general model described. Example 9 showcases the new exit nodes and the treatment of the `throw`#SSS: statement? as if it were an unconditional jump whose destination is the "Error exit".

**Example 9** (CFG of an uncaught `throw` statement)**.** Consider the simple Java method on the #Deleted: right#Added: left of figure 3.3; which performs a square root if the number is positive, throwing otherwise a `RuntimeError`. The CFG in the centre illustrates the treatment of `throw`, "normal exit" and "error exit" as pseudo–statements, and the PDG on the right describes the control dependencies generated from the `throw` statement to the following instructions and exit nodes.

```
double f(int x) {
  if (x < 0)
    throw new RuntimeException()
  return Math.sqrt(x)
}
```

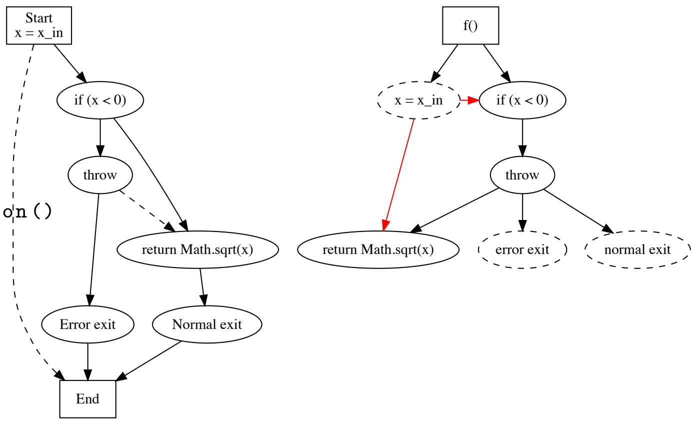



Figure 3.3: A simple program with a `throw` statement #Added: (left), its CFG (centre) and its PDG (#Deleted: left#Added: right).

### 3.3.2 `try-catch-finally` statement

The `try-catch` statement is the only way to stop an exception once it is thrown. It filters #Added: each exception by its type; letting those which do not match any of the catch blocks propagate to #Deleted: another#Added: an external `try-catch`#Deleted: surrounding it#Added: block or #Deleted: outside the method, to the previous #Deleted: one#Added: method in the call stack. On top of that, the `finally` block helps programmers guarantee code execution. It can be used replacing or in conjunction with `catch` blocks. The code placed inside a `finally` block is guaranteed to run if the `try` block has been entered. This holds true whether the `try` block exits correctly, an exception is caught, an exception is left uncaught or an exception is caught and another one is thrown while handling it (within its `catch` block).

#CCC: This would be useful to explain that the new dependencies introduced by the non-executable edges are not "normal" control dependencies, but "presence" dependencies. Opposite to traditional control dependence, where $a \rightarrow^{ctrl} b$ if and only if the number of times $b$ is executed is dependent on the *execution* of *a* (e.g. conditional blocks and loops); this new control dependencies exist if and only if the number of times *b* is executed is dependent on the *presence* or *absence* of *a*; which introduces a meta-problem. In the case of exceptions, it is easy to grasp that the absence of a catch block alters the results of an execution. Same with unconditional jumps, the absence of breaks modifies the flow of the program, but its execution does not control anything. A differentiation seems appropriate, even if only as subcategories of control dependence: execution control dependence and presence control dependence.

The main problem when including `try-catch` blocks in program slicing is that `catch` blocks are not always strictly necessary for the slice (less so for weak slices), but introduce new styles of control dependence #SSS: De esto se habla luego? de estos "new styles"? si es asi acuerdate de referenciarlo forward diciendo donde. Me imagino que es lo que pone en tu comentario de la presence control dependence.; which must be properly mapped to the SDG. The absence of `catch` blocks may also be a problem for compilation, as Java requires at least one `catch` or `finally` block to accompany each `try` block; though that could be fixed after generating the slice, if it is required that the slice be #SSS: be or to be? executable.

A typical#SSS: La tipica o la de la propuesta de Horwitz? Si es la de Horwitz di que ellos lo hacen asi, que ya hemos dicho que es lo mas importante hasta la fecha en Java. representation of the `try` block is as a pseudo-predicate, connected to the first statement inside it and to the instruction that follows the `try` block. This generates control dependencies from the `try` node to each of the instructions it contains. #CCC: This is not really a "control" dependency, could be replaced by the definition of structural dependence.#SSS: Totalmente, pero para decir esto hay que definir la structural dependence, que imagino que estara en la seccion 4. Inside the `try` there can be four distinct sources of exceptions:

**Method calls.** If an exception is thrown inside a method and it is not caught, it will surface inside the `try` block. As *checked* exceptions must be declared explicitly, method declarations may be consulted to see if a method call may or may not throw any exceptions. On this front, polymorphism and inheritance present no problem, as inherited methods must match the signature of the parent method —including exceptions that may be thrown. #Deleted: If#Added: In case *unchecked* exceptions are also considered, method calls could be analysed to know which exceptions may be thrown, or the documentation #Added: could be checked automatically for the comment annotation `@throws` to know which ones #Deleted: are thrown#Added: can be raised.

**`throw` statements.** The least common, but most simple, as it is #Deleted: treated as#Added: equivalent to#SSS: no las tratamos, solo decimos cuales son a `throw` inside a method #SSS: Hemos explicado como se trata un "throw inside un method"? O nos estamos refiriendo a una checked exception en una method call?. The type of the exception may be obvious, as most #CCC: this is a weird claim to make without backup exceptions are built and thrown in the same instruction; but it also may be hidden: e.g., `throw #Added: ((Exception) o#Added: )` where#SSS: por claridad, sino parece que la o forma parte de la frase `o` is a variable of type Object.

#SSS: Este es el caso mas directo de excepcion, un throw a fuego en un try-catch. Yo tal vez lo pondria antes que las method calls.

**Implicit unchecked exceptions.** If *unchecked* exceptions are considered, many common expressions may throw an exception, with the most common ones being trying to call a method or accessing a field of a `null` object (`NullPointerException`), accessing an invalid index on an array (`ArrayIndexOutOfBoundsException`), dividing an integer by 0 (`ArithmeticException`), trying to cast to an incompatible type (`ClassCastException`) and many others. On top of that, the user may create new types that inherit from `RuntimeException`, but those may only be explicitly thrown. Their inclusion in program slicing and therefore in the method's CFG generates extra dependencies that make the slices produced bigger#Added: . For this reason, they are not considered in most of the previous works.

**Errors.** May be generated at any point in the execution of the program, but they normally signal a situation from which it may be impossible to recover, such as an internal JVM error. In general, most programs will not attempt to catch them, and can be excluded in order to simplify implicit unchecked exceptions (any instruction at any moment may throw an Error).

#SSS: Despues de leer las 4 propongo el que me parece el orden ideal de explicacion: (1) throw (2) implicit unchecked (3) method calls (asi puedes aprovechar que ya has hablado de las uncheked ahora mismo y el lector ya ha recordado que eran) (4) errors

All exception sources are treated very similarly: the statement that may throw an exception is treated as a predicate, with the true edge connected to the next instruction #Deleted: were the statement to execute without raising exceptions#Added: of the normal execution; and the false edge connected to all the possible `catch` nodes which may be compatible with the exception thrown.

#Deleted: The case of method calls that may throw exceptions is slightly different, as#Added: Unfortunately, when the exception source is a method call, there is an augmented behaviour that make the representation slightly different, since there may be variables to unpack, both in the case of a normal or erroneous exit. To that end, nodes containing method calls have an unlimited number of outgoing edges: one #Deleted: to leads#Added: that points to a node labelled "normal return", after which the variables produced by any normal exit of the method are unpacked; and all the others #Added: point to any possible catch that may catch the exception thrown. Each catch must then unpack the variables produced by the erroneous exits of the method.

The "normal return" node is itself a pseudo-statement; with the *true* edge leading to the following instruction and #SSS: {the *false* one to the first common instruction between all the paths of length $\geq 1$ that start from the method call —which translates to the instruction that follows the `try` block if all possible exceptions thrown by the method are caught or the "Exit" node if there are some left uncaught.#SSS: }esta frase es larguisima, con aclaraciones en medio y no se entiende.

#Deleted: Carlos: CATCH Representation doesn't matter, it is similar to a switch but checking against types. The difference exists where there exists the chance of not catching the exception; which is semantically possible to define. When a `catch (Throwable e)` is declared, it is impossible for the exception to exit the method; therefore the control dependency must be redefined.

#Deleted: The filter for exceptions in Java's `catch` blocks is a type (or multiple types since Java 8), with a class that encompasses all possible exceptions (`Throwable`), which acts as a catch-all. In the literature there exist two alternatives to represent `catch`: one mimics a static switch statement, placing all the `catch` block headers at the same height, all pending from the exception-throwing exception and the other mimics a dynamic switch or a chain of `if` statements. The option chosen affects how control dependencies should be computed, as the different structures generate different control dependencies by default.

#Deleted:

**Switch representation.** There exists no relation between different `catch` blocks, each exception-throwing statement is connected through an edge labelled false to each of the `catch` blocks that could be entered. Each `catch` block is a pseudo-statement, with its true edge connected to the end of the `try` and the As an example, a `1 / 0` expression may be connected to `ArithmeticException`, `RuntimeException`, `Exception` or `Throwable`. If any exception may not be caught, there exists a connection to the "Error exit" of the method.

**If-else representation.** Each exception-throwing statement is connected to the first `catch` block. Each `catch` block is represented as a predicate, with the true edge connected to the first statement inside the `catch` block, and the false edge to the next `catch` block, until the last one. The last one will be a pseudo-predicate connected to the first statement after the `try` if it is a catch-all type or to the "Error exit" if it #Added: is not#Deleted: isn't.

**Example 10** (Catches.)**.** Consider the #Deleted: following segment of Java code in figure 3.4#Added: (left), which includes some statements #Deleted: that do not use data#Added: without any data dependence (X, Y and Z), and#Added: a method call to `f` that uses `x` and `y`, two global variables. `f` may throw an exception, so it has been placed inside a `try-catch` structure, with a statement in the `catch` that logs the #Added: `error` #Added: token when it occurs. Additionally, #Added: consider the case that when `f` exits #Deleted: without an error#Added: normally, only `x` is modified; but when an error occurs, only `y` is modified.

#Deleted: Note how the pseudo-statements act to create control dependencies between the *true* and *false* edges, such as the "normal return", "catch", "try".#Added: As can be seen in the CFG shown in figure 3.4 (centre), the nodes "normal return", "catch" and "try" are considered as pseudo-statements, and their *true* and *false* edges (solid and dashed arrows respectively) are used to create control dependencies. The statements contained after the function call, inside the `catch` #Added: block, and #Added: inside the `try` block#Deleted: s are respectively control dependent on the aforementioned nodes. Finally, consider the statement `Z`; which is not dependent on any part of the `try-catch` block, as all exceptions that may be thrown are caught: it will execute regardless of the path taken inside the `try` block. #CCC: Consider critiquing the result, saying that despite the last sentence, statements can be removed (the catch) so that the dependencies are no longer the same.

#CCC: From here to the end of the chapter, delete / move to solution chapter

Regardless of the approach, when there exists a catch–all block, there is no dependency generated from the `catch`, as all of them will lead to the next instruction. However, this means that if no data is outputted from the `try` or `catch` block, the catches will not be picked up by the slicing algorithm, which may alter the results unexpectedly. If this problem arises, the simple and obvious solution would be to add artificial edges to force the inclusion of all `catch` blocks, which adds instructions to the slice —lowering its score when evaluating against benchmarks— but are completely innocuous as they just stop the exception, without running any extra instruction.

Another alternative exists, though, but slows down the process of creating a slice from a SDG. The `catch` block is only strictly needed if an exception that it catches may be thrown and an instruction after the `try-catch` block should be executed; in any other case the `catch` block is irrelevant and should not be included. However, this change requires analysing the inclusion of `catch` blocks after the two–pass algorithm has completed, slowing it down. In any case, each approach trades time for accuracy and vice#Deleted: –#Added: versa, but the trade–off is small enough to be negligible.

Regarding *unchecked* exceptions, an extra layer of analysis should be performed to tag statements with the possible exceptions they may throw. On top of that, methods must be analysed and tagged accordingly. The worst case is that of inaccessible methods, which may throw any `RuntimeException`, but with the source code unavailable, they must be marked as capable of

```
try {
  X;
  f();
  Y;
} catch (Exception e) {
  System.out.println("error");
}
Z;
```

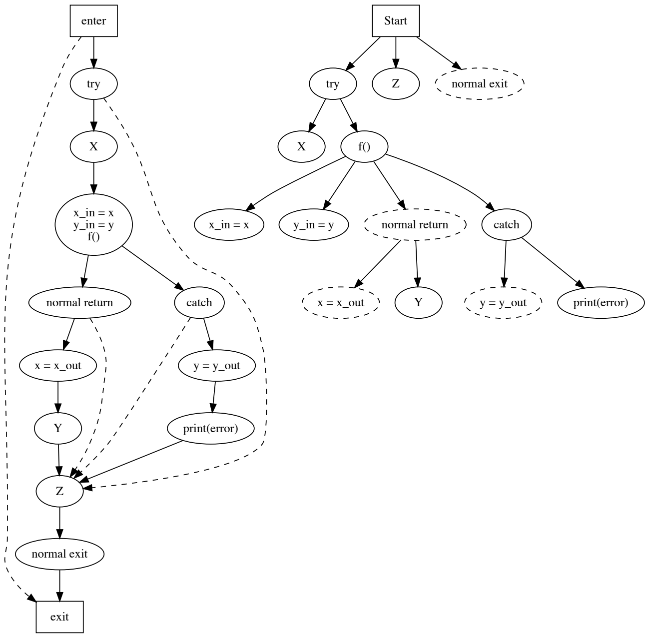



Figure 3.4: A simple example of the representation of `try-catch` structures and method calls that may throw exceptions. #JJJ: Pon quien es el CFG y quien el PDG. Por cierto, el arco del catch a la Z (rama false del catch) no es como los que se habian comentado. Es decir, no va a donde iria la ejecucion si el catch no estuviera.

throwing it. This results on a graph where each instruction is dependent on the proper execution of the previous statement; save for simple statements that may not generate exceptions. The trade–off here is between completeness and correctness, with the inclusion of *unchecked* exceptions increasing both the completeness and the slice size, reducing correctness. A possible solution would be to only consider user–generated exceptions or assume that library methods may never throw an unchecked exception. A new slicing variation that annotates methods or limits the unchecked exceptions #Added: may also#Deleted: to be considered.

Regarding the `finally` block, most approaches treat it properly; representing it twice: once for the case where there is no active exception and another one for the case where it executes with an exception active. An exception could also be thrown here, but that would be represented normally.

#SSS: Mi aportacion aqui es que posiblemente tenemos que restringir la aproximacion del Chapter 4 diciendo que vamos a tratar solo checked exceptions y mencionar al final que las unchecked serian igual pero anyadiendo mas analisis y mas codigo al slice. Sino cada vez que contemos lo que hacemos vamos a tener que estar diciendo: ”y para unchecked noseque...” todo el rato. Cuando presentes la solucion acota el problema y di que vamos a proponer una solucion para checked exceptions y que considera el caso en que no se capture lo que se lanza en el try catch (cosa que puede pasar en java). Eso ya es mejor que la solucion actual

# Chapter 4

# Proposed solution

#JJJ: Antes de nada, felicidades Carlos. En esta sección se ha notado una mejora importante. Sobretodo al introudcir los problemas, los ejemplos, etc. Sigue así!

#JJJ: This chapter features different problems and weaknesses of the current treatment that program slicing techniques use in presence of exceptions. Each problem is described with a counterexample that illustrates the loss of completeness or precision. Finally, for each problem a solution is proposed.

#JJJ: With regards to the problems, Even though the current state of the art considers exception handling, their treatment is not perfect. The mistakes made by program slicers can be classified in two: #JJJ: (1) those that lower the completeness and #JJJ: (2) those that lower the correctness. #JJJ: Remarco el 1 y el 2 porque los referencias mas adelante, lejos, y así se sabe que las referencias vienen aqui.

The first kind is the most important one, as the resulting slices may be incorrect #JJJ: (i.e., the behaviour of the slice is different from the behaviour of the original program)#Deleted: —as in produce different values than the original program— making them invalid for some uses of program slicing. #JJJ: A good example of the effects that these wrong slices may produce happens when they are used for program debugging, but the the error that we want to debug does not appear anymore, or even the slicing criterion cannot be reached due to an uncaught exception. #Deleted: As an example, imagine a slice used for program debugging which does not reach the slicing criterion due to an uncaught exception.

The second kind is less #JJJ: critic#Deleted: important, but still #JJJ: important because a wrong treatment of exceptions can cause the inclusion of wrong dependencies in the slice, thus producing unnecessary long slices that may turn to be useless for some applications#Deleted: useful to address, as the smaller a slice is, the easier it is to use it.

#Deleted: The rest of this chapter features different errors found in the state of the art, each with a detailed description, example, and proposals that solve them.

## 4.1 Unconditional jump handling

The standard treatment of unconditional jumps as pseudo-statements introduces two separate correctness errors: #JJJ: *the subsumption correctness error*, which is relevant in the context of both strong and weak slicing, and the *structure-exiting jump*, #JJJ: which#Deleted: that is only relevant in the context of weak slicing.

### 4.1.1 #JJJ: Problem 1: Subsumption correctness error

This problem has been known since the seminal publication on slicing unconditional jumps [3]: chapter 4 details an example where the slice is bigger than it needs to be, and leave the solution of that problem as an open question to be solved in future publications. A similar example —with `break` statements instead of `goto`— is shown in example 11.

**Example 11** (Example of unconditional jump subsumption [3])**.** Consider the code shown in the left side of figure 4.1. It is a simple Java method containing a `while` statement, from which the execution may exit naturally or through any of the `break` statements (lines 6 and 9). For the rest of statements and expressions, uppercase letters are used; and no data dependencies are considered, as they are not relevant to the problem at hand.

```
1  public void f() {
2    while (X) {
3      if (Y) {
4        if (Z) {
5          A;
6          break;
7        }
8        B;
9        break;
10     }
11     C;
12   }
13   D;
14 }
```

```
1  public void f() {
2    while (X) {
3      if (Y) {
4        if (Z) {
5
6          break;
7        }
8
9        break;
10     }
11     C;
12   }
13
14 }
```

```
1  public void f() {
2    while (X) {
3      if (Y) {
4
5
6
7
8
9        break;
10     }
11     C;
12   }
13
14 }
```

Figure 4.1: A program (left), its computed slice (centre) and the smallest complete slice (right).

Now consider statement `C` (line 11) as the slicing criterion. Figure 4.2 displays the SDG produced for the program, and the nodes selected by the slice. Figure 4.1 displays the computed slice on the centre, and the #JJJ: minimal slice#Deleted: smallest slice possible on the left#JJJ: en realidad hay otro minimal slice si dejamos el otro break y quitamos el que hemos dejado. The inner `break` on line #JJJ: 6#Deleted: 9 and the `if` surrounding it (line #JJJ: 4#Deleted: 7) have been unnecessarily included. Their inclusion would not be specially problematic, if it were not for the condition of the `if` statement #JJJ: (line 4), which may include extra data dependencies #JJJ: that are unnecessary in the slice and that may led to include other unnecessary statements, making the slice even more imprecise#Deleted: , whose only task is to control line 3.

Line 6 is not useful, because whether or not it executes, the execution will continue on line 13 (after the `while`), as guaranteed by line 9, which is not guarded by any condition. Note that `B` is still control-dependent on line #JJJ: 6#Deleted: 5, as it has a direct effect on it, #JJJ: no termino de entender esta frasebut the dependence from line 5 to line 9 introduces useless statements into the slice.

The problem showcased in example 11 can be generalized for any pair of unconditional jump statements that are nested and whose destination is the same. Formally, #JJJ: lo que sigue es bastante lioso. Yo crearia un entorno "problem" (como el de definition o example) y pondria el problema descrito formalmente en ese entorno. Despues, lo aclararia con una breve explicacion similar a la que hay entremezclada con la definicion formalif a program $P$ contains a pair of unconditional jumps without any data (e.g. `goto label`, `continue [label]`, `break [label]`, `return`) $j_A$ and $j_B$ whose destinations (the instruction that will be executed after them) are $A$ and $B$, then $j_B$ is superfluous in the slice if and only if $A = B$ and $j_B$ is inside a conditional

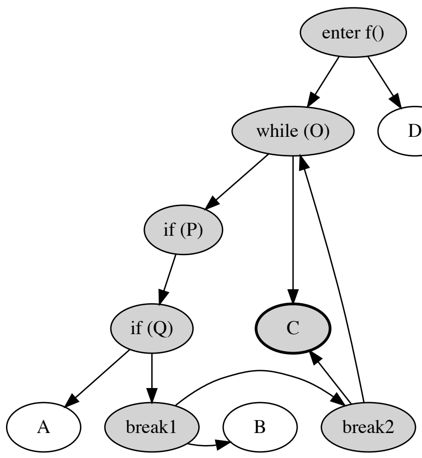



Figure 4.2: The system dependence graph for the program of figure 4.1, with the slice marked in grey, and the slicing criterion in bold.#JJJ: En las condiciones pones O,P,Q en lugar de X,Y,Z

instruction $C$, and $j_A$ follows $C$ (not necessarily immediately). #CCC: Buscar mejor descripcion para la estructura “nested”. #CCC: Maybe use control dependencies between them. Once $j_B$ is included, $C$ will also be included, and so will all of its data dependencies.

#### #JJJ: #Deleted: ProposalA solution for the subsumption correctness error

As only the minimum amount of control edges are inserted into the PDG (according to definition 7), the only edge that can be traverse to include the inner jump ($j_B$) is an edge $j_B \rightarrow^{ctrl} j_A$. An exception can be included when generating the PDG, such that control edges between two unconditional jumps $j_X$ and $j_Y$ whose destinations are $X$ and $Y$ will not be included if $X = Y$.

If the edge is not present, all inner unconditional jumps and their containing structures will be excluded from the slice, unless they are included for another reason.

#JJJ: pon a continuacion un ejemplo solucionando el problema (al menos di como quedaría el SDG)

### 4.1.2 #JJJ: Problem 2: Unnecessary instructions in weak slicing

#JJJ: Esta frase esta mal construidaIn the context of weak slicing, as it is not necessary to behave exactly like the original program. This means that some statements may be removed, even if it means that a loop will become infinite, or an exception will not be caught. The following example describes a specific example which is generalized later in this section.

**Example 12** (Unnecessary unconditional jumps)**.** Consider the code for method `g` on figure 4.3, which features a simple loop with a `break` statement within. The slice in the middle has been created with respect to the criterion (line 6, variable `x`), and includes everything except the print statement. This seems correct, as the presence of lines 4 and 5 determine the number of times line 6 is executed.

However, if #JJJ: one considers#Deleted: you consider weak slicing, instead of strong slicing; the loop's termination stops mattering, lines 4 and 5 are no longer relevant. Without them, the slices produce#JJJ: #Deleted: s an infinite list #JJJ: of natural numbers (0, 1, 2, 3, 4, 5...), but as that is a prefix #JJJ: suena raro que una lista infinita sea un prefijo de 0-9, mas bien es al revesof the original program —which outputs the numbers 0 to 9— the program is still a valid slice (pictured on figure 4.3's right side).

Note that the removal of lines 4 and 5 is only possible if there are no statements in the slice after the `while` statement. If the slicing criterion is line 8, variable `x`, lines 4 and 5 are required to print the value, as without them, the program would loop indefinitely and never execute line 8.

```
1 void g() {
2   int x = 0;
3   while (x > 0) {
4     if (x > 10)
5       break;
6     x++;
7   }
8   System.out.println(x);
9 }
```

```
1 void g() {
2   int x = 0;
3   while (x > 0) {
4     if (x > 10)
5       break;
6     x++;
7   }
8
9 }
```

```
1 void g() {
2   int x = 0;
3   while (x > 0) {
4
5
6     x++;
7   }
8
9 }
```

Figure 4.3: A simple loop with a break statement (left), its computed slice (middle) with respect to line 5, variable `x`, and the smallest weak slice (right) for the same slicing criterion.

If we try to generalize this problem, it becomes apparent that instructions that jump backwards (e.g., `continue`) present a problem, as they may add executions in the middle, not at the end (where they can be disregarded in weak slicing). Therefore, not only has the jump to go forwards, but no instruction can be performed after the jump.

Therefore, a forward jump $j$ (e.g., `break`, `return [value]`, `throw [value]`) whose destination is $X$ is not necessary in a slice $S$ if and only there is no statement $s \in S$ which is after $X$, meaning that there is a path from $X$ to $s$ in the CFG.

As with the previous error, the problem is not the inclusion of the jump and its controlling conditional instruction, but the inclusion of the data dependencies of the condition guarding the execution of the jump.

#### #JJJ: #Deleted: ProposalA solution for the unnecessary instructions in weak slicing

This problem cannot be easily solved, as it is a "dynamic" one, requiring information about the completed slice before allowing the removal of unconditional jumps and their dependencies. This means that the cost of this proposal #JJJ: cannot#Deleted: can not be offloaded to the creation of the SDG as with the previous one.

#JJJ: frase incorrectaOur proposal revolves around temporarily remove edges from the SDG: given an SDG of the form #JJJ: En la definicion de SDG salia esta sextupla?$G = \langle N, E_c, E_d, E_{in}, E_{out}, E_{fc} \rangle$, remove from $E_c$ any edge of the form $x \rightarrow^{ctrl} y | x, y \in N$, where $x$ is an unconditional forward jump; perform the slice normally; and then —if there is any statement after the destination of $x$ in the slice— restore the edges removed in the first step and recompute the slice. The slice would still be linear, because each node would be visited at most once; but the algorithm has a higher complexity, and the removal and restoration of the control edges has a cost; albeit small.

#JJJ: pon a continuacion un ejemplo solucionando el problema

## 4.2 The `try-catch` statement

In this section we present an example where the current approximation for the `try-catch` statement fails to capture all the correct dependencies and excludes from the slice some statements which are necessary for a complete slice (both weak and strong). After that, we generalize the set of cases where that is a problem and its possible appearances in real-life development. Finally, we propose a solution which properly represent all the dependencies introduced by the `try-catch`, focusing on producing complete strong slices.

#### The types of control dependence

#CCC: this subsection snippet could go in another place

Even though it continues to be used for control dependence, definition 5 does not have the same meaning when applied to conditional instructions and loops as it has when applied to unconditional jumps and other complex structures, such as the `switch` and `try-catch` statements.

Originally, the definition of control dependence signified that the execution of a statement affected whether or not another one executed (or kept executing). In contrast, unconditional jumps, and `try-catch` statements' execution do not affect the following instructions; its presence or absence is what generates the control dependency. For those instructions, control dependencies are still generated with the same edges, but require the addition of extra edges to the CFG [3, 2].

### 4.2.1 The control dependencies of a `catch` block

In the current approximation for exception handling [2], `catch` blocks do not have any outgoing dependence leading anywhere except the instructions it contains. This means that, as showcased in chapter 1, the only way a `catch` statement may appear in a slice is if there is a data dependency or one of the statements inside it is needed.

The only occasion in which `catch` blocks generate any kind of control dependency is when there is an exception thrown that is not covered by any of the `catch` blocks, and the function may exit with an exception. In that case, the instructions after the `try-catch` block are dependent on an uncaught exception not being thrown.

But, compared to the treatment of unconditional exceptions does not match the treatment of `try-catch` statement: unconditional jumps have a non-executable edge to the instruction that would be executed in their absence; `catch` statements do not.

**Example 13** (`catch` statements' outgoing dependencies)**.** Consider the code shown in figure 4.4, which depicts a `try-catch` where method `f`, which may throw an exception, is called. The function may throw either a `ExceptionA`, `ExceptionB` or `Exception`-typed exception; and the `try-catch` considers all three cases, logging the type of exception caught. Additionally, `f` accesses and modifies a global variable `x`.

The CFG and PDG associated to that code is depicted in figure 4.5. As can be seen, the only two elements that are dependent on any `catch` are the log statement and the unpacking of `x`. If the following statement used `x` in any way, all `catch` statements would be selected, otherwise they are ignored, and not deemed necessary. It is true that they are normally not necessary; i.e., if the slicing criterion was placed on `next` (line 10), the whole `try-catch` would be rightfully ignored; but there exist cases where `f()` (line 2) would be part of the slice, and the absence of `catch` statements would result in an incomplete slice.

**Example 14** (Incorrectly ignored `catch` statements)**.** Consider the code in figure 4.6, in which a method is called twice: once inside a `try-catch` statement, and a second time, outside. `f` also

```
try {
  f();
} catch (ExceptionA e) {
  log("Type␣A");
} catch (ExceptionB e) {
  log("Type␣B");
} catch (Exception e) {
  log("Exception");
}
next;
```

Figure 4.4: A snippet of code of a call to a method that throws exceptions and `catch` statements to capture and log them.

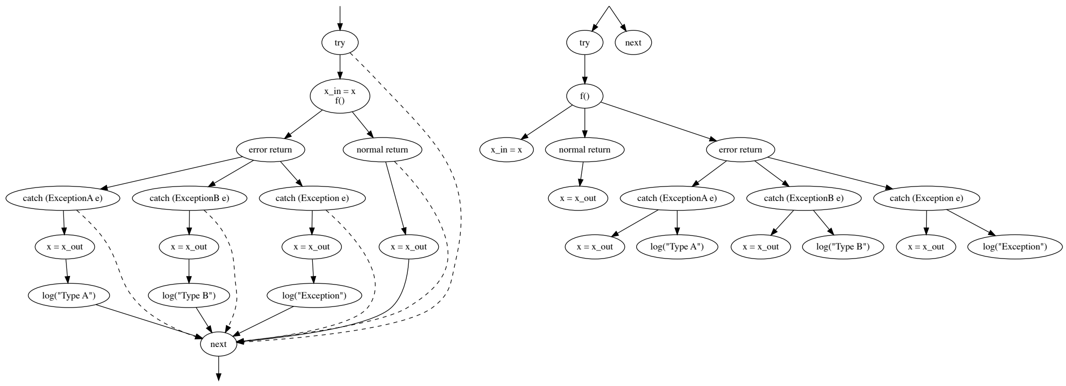



Figure 4.5: CFG (left) and PDG (right) of the code shown in figure 4.4.

accesses and modifies variable `x`, which is redefined before the second call to `f`. Exploring this example, we demonstrate how line 3 will be necessary but not included in the slice.

Figure 4.7 displays the program dependence graph for the snippet of code on the left side of figure 4.6. Data dependencies are shown in red, and summary edges in blue. The set of nodes filled in grey represent the slice with respect to a slicing criterion in method `f` (line 4, `x`). In the slice, both calls to `f` and its input (`x_in = x`) are included, but the `catch` block is not present. The execution of the slice may not be the same: if no exception is thrown, there is no change; but if `x` was odd before entering the snippet, an exception will be thrown and not caught, exiting the program prematurely.

### A solution for the `catch`'s lack of control dependencies

`catch` statements should be handled like unconditional jumps: a non-executable edge should connect them to the instruction that would run if they were absent. For `catch` statements, the non-executable edge would connect them to the `catch` that contains the most immediate supertype (or multiple); or to the error exit, if no other `catch` could catch the same exception. This would create a tree-like structure among `catch` statements, with the root of each tree connected to the "error exit" of the method. This would generate dependencies between `catch` statements, and more importantly, dependencies from the `catch` statements to the instructions that follow

```
try {
  f();
} catch (Exception e) {
  log("error");
}
x = 0;
f();
```

```
void f() throws Exception {
  if (x % 2 != 0)
    throw new Exception();
  x++;
}
```

Figure 4.6: A method that may throw exceptions (`f`), called twice, once surrounded by a `try-catch` statement, and another time after it. On the right, the definition of `f`.

the `try-catch` statement.

Unfortunately, this creates the same behaviour as with unconditional jumps: all the instructions that follow a `try-catch` structure is dependent on the presence of the `catch` statements, which in turn are dependent on all the statements that may throw exceptions. In practice, the inclusion of any statement after a `try-catch` statement would require the slice to include all `catch` statements, the statements that may throw exceptions, and all the statements required by control or data dependencies. This is a huge number of instructions just for including the `catch` statements.

Our solution makes slices complete again, but makes them much less correct. As a solution for the incorrectness, we could insert an additional requirement when including `catch` blocks: if they are included because of their control dependencies on instructions outside the `try-catch`, they need to satisfy an additional condition before being in the slice: have a node in the slice which may throw a compatible exception. In order to achieve this, control dependencies whose source is a `catch` node and its destination is outside that same `catch` are coloured green and labelled (2). Additional edges are added between every `catch` and any statement that may throw a compatible exception; are also coloured green, and labelled (1). When traversing the graph, only include `catch` statements if they are reached through an unlabelled edge or if they are reached by at least one edge with each label (1 and 2). #CCC: Add que solo se pueda atravesar uno de los arcos verdes (una vez llegas a un nodo a traves de un arco verde, no continuas recorriendo arcos verdes) #CCC: optimizacion 2, que los nodos catch se repitan para cada funcion tenga los suyos propios. El contenido del catch es comun a todos, pero las cabeceras y el despempaquetamiento de variables es individual para cada funcion. De este modo no se coge a todas las funciones. Tambien se podria emplear como alternativa a los arcos etiquetados, creando un set de nodos que no tienen padre, y sirven exclusivamente para que las instrucciones futuras dependan de ellas.

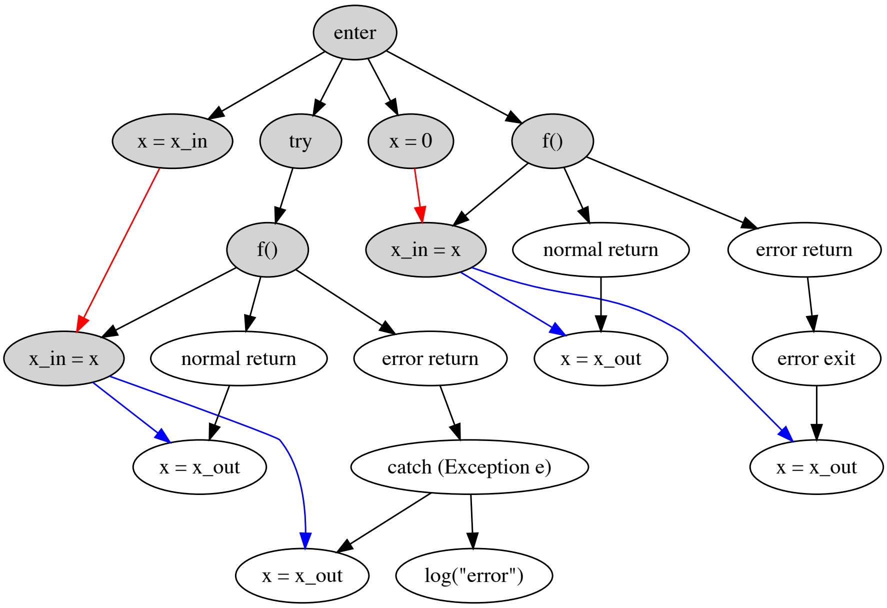



Figure 4.7: The system dependence graph of the left snippet of figure 4.6. `f` and the edges that connect to it are not shown for simplicity.

# Chapter 5

# Related work

Slicing was proposed [17] and improved until the proposal of the current system (the SDG) #CCC: (citation). Specifically in the context of exceptions, multiple approaches have been attempted, with varying degrees of success. There exist commercial solutions for various programming languages: #CCC: name them and link. In the realm of academia, there exists no definite solution. One of the most relevant initial proposal#Added: s [2], although not the first one [15, 16] to target Java specifically.

It uses the existing proposals for *return*, *goto* and other unconditional jumps to model the behavior of *throw* statements. Control flow inside *try-catch-finally* statements is simulated, both for explicit *throw* and those nested inside a method call. The base algorithm is presented, and then the proposal is detailed as changes. Unchecked exceptions are considered but regarded as "worthless" to include, due to the increase in size of the slices, which reduces their effectiveness as a debugging tool. This is due to the number of unchecked exceptions embedded in normal Java instructions, such as `NullException` in any instance field or method, `IndexOutOfBoundsException` in array accesses and countless others. On top of that, handling *unchecked* exceptions opens the problem of calling an API to which there is no analyzable source code, either because the module was compiled before-hand or because it is part of a distributed system. The first should not be an obstacle, as class files can be easily decompiled. The only information that may be lost is variable names and comments, which #Added: do not#Deleted: don't affect a slice's precision, only its readability.

Chang and Jo [10] present an alternative to the CFG by computing exception-induced control flow separately from the traditional control flow computation, but go no further into the ramifications it entails for the PDG and the SDG.

Jiang et al. [8] describe#Deleted: s a solution specific for the exception system in C++, which differs from Java's implementation of exceptions. They reuse the idea of non-executable edges in *throw* nodes, and introduce handling *catch* nodes as a switch, each trying to catch the exception before deferring onto the next *catch* or propagating it to the calling method. Their proposal is center#Added: ed around the IECFG (Improved Exception Control-Flow Graph), which propagates control dependencies onto the PDG and then the SDG. Finally, in their SDG, each normal and exceptional return and their data output are connected to all *catch* statements where the data may have arrived, which is fine for the example they propose, but could be inefficient if the method has many different call nodes.

Others [12] have worked specifically on the C++ exception framework. #CCC: remove or expand.

Finally, Hao [9] introduced a Object-Oriented System Dependence Graph with exception handling (EOSDG), which represented a generic object-oriented language, with exception handling capabilities. Its broadness allows for the EOSDG to fit into both Java and C++. It uses concepts from Jiang [8], such as cascading *catch* statements, while adding explicit support for virtual calls, polymorphism and inheritance.

lternative explanation of [2], with counter xample. Maybe should move the counter example backwards.

---

In her#JJJ: their? paper #Added: [?], Horwitz #JJJ: et al.? suggests treating exceptions in the following way:

- Statements are divided into statements, predicates (loops and conditional blocks) and pseudo-predicates (return and throw statements). Statements only have one successor in the CFG, predicates have two (one when the condition is true and another when false), pseudo-predicates have two, but the one labeled "false" is non-executable. The non-executable edge connects to the statement that would be executed if the unconditional jump was replaced by a "nop".
- *try-catch-finally* blocks are treated differently, but it has fewer dependencies than needed. Each catch block is control-dependent on any statement that may throw the corresponding exception. The #JJJ: ???

#JJJ: Crea un entorno example

Example

```
void main() {
  int x = 0;
  while (true) {
    try {
      f(x);
    } catch (ExceptionA e) {
      x--;
    } catch (ExceptionB e) {
      System.err.println(x);
    } catch (ExceptionC e) {
      System.out.println(x);
    }
    System.out.println(x);
  }
}

void f(x) {
  x--;
  if (x > 10)
    throw new ExceptionA();
  else if (x == 0)
    throw new ExceptionB();
  else if (x > 0)
    throw new ExceptionC();
  x++;
  System.out.println(x);
}

static class ExceptionA extends ExceptionC {}
static class ExceptionB extends Exception  {}
static class ExceptionC extends Exception  {}
```

In this example we can explore all the errors found with the current state of the art. #JJJ: Seria mucho más claro si tenemos un grafo con la soluciones propuesta para cada problema.

The first problem found is the lack of `catch` statements in the slice, as no edge is drawn from the catch. Some of the catch blocks will be included via data dependencies, but some may not be reached, though they are still necessary if the slice includes anything after a caught exception. Therefore, an extra control dependency must be introduced, in order to always include a “catch” statement in the slice if the “throw” statement is in the slice. In the example, only the catch statement from line 20 will be included #JJJ: con que criterio? no has definido el ejemplo. El lector no sabe como interpretar esta figura, and if ExceptionC or ExceptionB were thrown, they would not be caught. That would not be a problem if the function $f$ was not executed again, but it is, making the slice incorrect.

## Chapter 6

# Conclusion

#CCC: todo

# Bibliography


[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[2] Matthew Allen and Susan Horwitz. Slicing java programs that throw and catch exceptions. *SIGPLAN Not.*, 38(10):44–54, June 2003.

[3] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pages 206–222, London, UK, UK, 1993. Springer-Verlag.

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

[5] Sebastian Danicic, Richard Barraclough, Mark Harman, John Howroyd, Ákos Kiss, and Michael Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theoretical Computer Science*, 412:6809–6842, 11 2011.

[6] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[7] Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical report, In Proc. of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1994.

[8] S. Jiang, S. Zhou, Y. Shi, and Y. Jiang. Improving the preciseness of dependence analysis using exception analysis. In *2006 15th International Conference on Computing*, pages 277–282. IEEE, Nov 2006.

[9] H. Jie, J. Shu-juan, and H. Jie. An approach of slicing for object-oriented language with exception handling. In *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, pages 883–886, Aug 2011.

[10] Jang-Wu Jo and Byeong-mo Chang. Constructing control flow graph for java by decoupling exception flow from normal flow. pages 106–113, 05 2004.

[11] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.

[12] Prakash Prabhu, Naoto Maeda, and Gogul Balakrishnan. Interprocedural exception analysis for c++. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 583–608, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Thomas Reps and Wuu Yang. The semantics of program slicing and program integration. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, pages 360–374, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[14] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3), June 2012.

[15] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 348–357. IEEE, Nov 1998.

[16] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 432–441. IEEE, May 1999.

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