



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



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

1	Introduction	3
1.1	Motivation	3
1.2	Contributions	5
2	Background	7
2.1	Program slicing	7
2.1.1	Computing program slices with the system dependence graph	10
2.1.2	Program slicing metrics	11
2.1.3	Variations and applications of program slicing	13
2.2	Exception handling in Java	14
2.2.1	Exception handling in other programming languages	15
3	Main explanation?	18
3.1	First definition of the SDG	18
3.2	Unconditional control flow	23
3.3	Exceptions	27
3.3.1	<code>throw</code> statement	27
3.3.2	<code>try-catch-finally</code> statement	28
4	Proposed solution	33
4.1	Unconditional jump handling	33
4.1.1	#JJJ: Problem 1 : Subsumption correctness error	34
4.1.2	#JJJ: Problem 2 : Unnecessary instructions in weak slicing	36
4.2	The <code>try-catch</code> statement	37
4.2.1	The control dependencies of a <code>catch</code> block	38
5	Related work	41
6	Conclusion	44

Chapter 1

Introduction

1.1 Motivation

Program slicing is a technique for program analysis and transformation whose main objective is to extract from a program the set of statements that affect a specific statement and set of variables, called a *slicing criterion* [27, 26]. It answers the question “Which parts of a program affect a set of variables in a specific statement?” The program obtained by program slicing is called a *slice*, and it has many uses, such as debugging [8], program specialization [19], software maintenance [10], code obfuscation [18], etc. This technique was originally defined [27] for a simple imperative programming language, but now can be used with practically all programming languages and paradigms.

Example 1 (Program slicing applied a simple Java method). Consider the code shown on the left side of figure 1.1, which is a simple method written in Java. If that method is sliced with respect to the slicing criterion (line 5, variable *x*), the slice would be the program on the right. The *if* and *print* statements would be excluded from the slice, as they do not affect the value of *x*. As a test, the execution of line 5 on both programs would yield the same result —assuming both the original program and the slice are executed with the same input value.

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

Figure 1.1: A simple Java method (left) and its slice w. r. t. slicing criterion (line 5, *x*).

As depicted in example 1, slices are subsets of the original program. In the most general form, the execution of slices produces the same values in the slicing criterion as the original program would. In other words, the slice criterion behaves identically in the slice as in the original. Some uses of program slicing, such as program specialization, require the slices to be executable, which is useful to extract an independent process from a bigger program or software library. Other uses do not, as the slices are used to find the complete set of dependencies of a slicing criterion.

Though it may seem a really powerful technique, many programming languages lack a mature program slicer which covers the whole language. Even commonly widespread languages like Java

does not have a complete program slicer that is publicly available, or documented in the literature; which makes it difficult to use program slicing where it may be needed. Nevertheless, there exist commercial program slicers that cover Java, such as CodeSurfer¹.

Building a program slicer is not a simple task, requiring a considerable amount of analysis to obtain a valid slice. Smaller slices are preferable, but even more difficult to create. In Java specifically many situations lead to several scenarios, such as arrays, polymorphism and inheritance, and exception handling that are quite difficult to analyze. This is the reason there does not exist a universal solution for all the existent problems in the field of program slicing. Conversely, many approaches are usually proposed to solve the same slicing problem. Program slicing is used in so many applications —debugging, program comprehension, parallelization, dead code removal— that any improvement to the state of the art improves those processes.

Among others, there is an area that has been investigated, but does not have a definitive solution yet: exception handling. Example 2 shows how, even using the latest developments to handle exceptions in program slicing [2, 14], the slice produced is not valid.

Example 2 (Program slicing with exceptions). Consider figure 1.2: the Java program on the left has been sliced (on the right) using Allen et al.’s proposal [2]; with respect to the slicing criterion (line 17, variable `a`).

<pre> 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 } </pre>	<pre> 1 void f(int x) throws Exception { 2 try { 3 g(x); 4 } 5 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 } </pre>
---	--

Figure 1.2: A simple Java program with exception (left) and its slice w. r. t. line 17, variable `a` (right).

As a test of the validity of the slice, we can execute both (with the initial call being `f(0)`). We can define the *execution history* as the list of instructions executed by a program [17]. As an example, the execution log of `g(1)` is 13, 14, 17, and the execution log of `g(0)`, 13, 14, 15. When the program is executed from the call `f(0)`, the execution history of the original program (left) is: 1, 2, 3, 13, 14, 15, 4, 5, 8, 10, 13, 14, 17. The slicing criterion executes once: `a` has value 1. In contrast, the execution history for the slice is 1, 2, 3, 13, 14, 15. Method `g` throws an exception, which is not caught, and the program ends with an error, stopping abruptly before reaching the slicing criterion.

The problem in this example is that the `catch` block in line 4 is not included. This is because —according to the system dependence graph [11] computed using Allen et al.’s algorithm [2] and

¹Created by GrammaTech. For more information, consult their website at <https://www.gramatech.com/>

shown in Figure 1.3 below— it does not influence the execution of line 17. The graph displays the statements of the methods as nodes; and the dependencies between statements as edges. Some nodes have its outline dashed; as they do not correspond to a statement, but are needed by the algorithm. The node associated with the slicing criterion is marked in bold and the nodes that represent the slice are filled in grey. Note that there are some edges between both methods that are not shown. The only relevant ones (the ones traversed to create the slice) are shown, and the rest are hidden for clarity.

The graph traversal will be explained later, but the basic rule is that edges are traversed backwards starting from the slicing criterion. Any node that is reached is part of the slice, the rest can be disregarded.

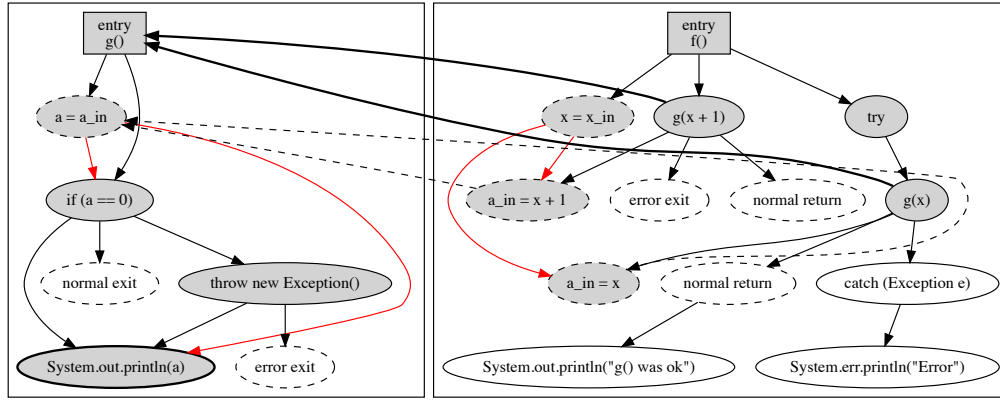


Figure 1.3: The system dependence graph for the method shown in Figure 1.2.

#CCC: mover el grafo y la explicación a después del background; el porqué y la solución se presenta en sección X???

Example 2 is a contribution of this thesis, because it showcases an important error in the current state of the art. This example is later generalized (see chapter 4), as under some conditions all `catch` statements are ignored, regardless of if it is needed or not. 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 et al. [2] did not tackle this problem, as for some examples the `catch` statement is included or unnecessary.

A real-life, commonly used instance of example 2 is the writing of any information to a file or a database; or any other instruction that has no data output (excluding side effects) and may throw an exception.

1.2 Contributions

The main contribution of this thesis is a new approach for program slicing with exception handling for Java programs. Our approach extends the existing techniques proposed by Allen et al. [2]. It is able to generate valid slices for all cases considered in their work, but it also provides a solution to other cases not contemplated by them. For the sake of completeness and in order

to explain the process that led us to this solution, we first summarize the fundamentals of program slicing and its terminology; delving deeper in the progress of program slicing techniques related to exception handling.

The rest of this thesis is structured as follows: chapter 2 summarizes the theoretical background required in program slicing and exception handling, chapter 3 analyzes each structure used in exception handling and explores the already available solution. Chapter 4 provides a list of problems that occur in the state of the art, detailing the scope and importance of each one, and proposes an appropriate solution, chapter 5 provides a bird's eye view of the current state of the art, and finally, chapter 6 concludes the thesis and explores future avenues of work, such as improvements or optimizations that have not been explored in our solution.

Chapter 2

Background

Before delving into the specific problems that exist in program slicing currently, let's explore the surface of this thesis' relevant fields: program slicing and exception handling. The last one will be focused specifically on the Java programming language, but could be generalized to other popular programming languages which feature a similar exception handling system (e.g., Python, JavaScript, C++).

2.1 Program slicing

This section provides a series of definitions and background information so that future definitions can be grounded in a common foundation. [#CCC: ampliar intro?](#)

Definition 1 (Program slicing). *Program slicing* is the process of extracting a slice S given a program P and a slicing criterion SC .

Definition 2 (Slicing criterion). Given a program P , composed of statements and containing variables $x_1, x_2 \dots x_n \in \text{vars}$, a *slicing criterion* is a tuple $SC = \langle s, v \rangle$ where $s \in P$ is a single statement that belongs to the program, and v is a set of variables from P . Each variable in v may not appear in s .

Definition 3 (Slice). Given a program P and a slicing criterion $SC = \langle s, v \rangle$, a *slice* is a subset of statements of P ($S \subset P$), which behaves like the original program P , when considering the values of the variables in v in statement s .

Definition 4 (Execution history). Given a program P , composed of a set of statements $S = \{s_1, s_2, s_3 \dots s_n\}$, and a set of input values I , the *execution history* of P given I is the list of statements H that is executed, in the order that they were executed.

Until now, the concept of slicing has been centred around finding the instructions that affect a variable. That is the original definition, but as time has progressed, variations have been proposed, with the one described in definitions 1, 2 and 3 is called *static backward slicing*. It is also the one that will be used throughout this thesis, though the errors detected and solutions proposed can be easily generalized to others. The different variations are described later in this chapter, but there exist two fundamental dimensions along which the slicing problem can be proposed [23]:

- *Static* or *dynamic*: slicing can be performed statically or dynamically. *Static slicing* [23] produces slices that consider all possible executions of the program: the slice will be correct

regardless of the input supplied. In contrast, *dynamic slicing* [17, 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 execution history that corresponds to one instance of the selected statement, making it much more specific. It may help find *#CCC: idk if I need the “to”* a bug related to indeterministic behaviour—such as a random or pseudo-random number generator—but, despite selecting the same slicing criterion in the same program, the slice must be recomputed for each set of input values or execution considered. *#CCC: Talk about quasi-static as a middle ground?*

- *Backward or forward: backward slicing* [23] looks for the statements that affect the slicing criterion. It sits among the most commonly used slicing technique. In contrast, *forward slicing* [4] computes the statements that are affected by the slicing criterion. There also exists a middle-ground approach called *chopping* [13], which is used to find all the 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 seminal definition of program slicing by Weiser [27], the most studied variation of slicing has been *static backward slicing*, which has been defined in previous sections of this thesis. That definition can be split in two sub-types, *strong* and *weak* slices, with different levels of requirements and uses in different fields.

Definition 5 (Strong static backward slice [26]). Given a program P and a slicing criterion $SC = \langle s, v \rangle$, S is a *strong static backward slice* of P with respect to SC if S fulfils the following properties:

1. S is an executable program.
2. $S \subseteq P$, or S is the result of removing 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 así, 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 6 (Weak static backward slice [22]). *#JJJ: Si esa cita no es, entonces puedes usar la de Binkley: [5]* Given a program P and a slicing criterion $SC = \langle s, v \rangle$, S is the *weak static backward slice* of P with respect to SC if S fulfils the following properties:

1. S is an executable program.
2. $S \subseteq P$, or S is the result of removing 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 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: $\forall i \in I, v \in V \rightarrow seq(i, v, P) \text{ Pref } seq(i, v, S)$ 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

Original program	1	2	6	-	-
Slice A	1	2	6	-	-
Slice B	1	2	6	24	120
Slice C	1	1	1	1	1

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

todos los inputs posibles para P . Por ahí irían los tiros creo yo. #SSS: Formalización 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 definición del paper de POI testing (Sergio sabe cuál es).

Both definitions (5 and 6) 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 weak. #SSS: Josep? #JJJ: para Strong se puede poner a Weiser. Para Weak se puede poner a Binkley [5]). Most do not differentiate them, or acknowledge the other variant, because most publications focus on one variant exclusively. Therefore, although the definitions come from different authors, the *weak* and *strong* nomenclature employed here originates from a control dependency analysis by Danicic [7], 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*.

Different applications of program slicing use the option that fits their needs, though *weak* is used if possible, because the resulting slices are smaller statement-wise, and the algorithms used tend to be simpler. Of course, if the application of program slices requires the slice to behave exactly like the original program, then *strong* slices are the only option. As an example, debugging uses weak slicing, as it does not matter what the program does after reaching the slicing criterion, which is typically the point where an error has been detected. In contrast, program specialization requires strong slicing, as it extracts features or computations from a program to create a smaller, standalone unit which performs in the exact same way.

Along the thesis, we indicate which kind of slice is produced with each problem detected and technique proposed.

Example 3 (Strong, weak and incorrect slices). Consider table 3, which displays the sequence of values or execution history obtained with respect to different slices of a program and the same slicing criterion.

The first row stands for the original program, which computes 3!.

Slice A’s execution history is identical to the original and therefore it is a strong slice.

Slice B’s execution history does not stop after producing the same first 3 values as the original: it is a weak slice. An instruction responsible for stopping the loop may have been excluded from the slice.

Slice C is incorrect, as the execution history differs from the original program in the second column. It seems that some dependency has not been accounted for and the value is not updating.

#CCC: The following paragraph has already been repeated in previous sections, mainly the motivation. Consider its removal and the addition of citations to the previous mention.

#JJJ: Even though the original proposal by Weiser [27] focussed on an imperative language, program slicing is a language-agnostic technique. Program slicing is a language-agnostic technique, but the original proposal by Weiser [27] 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 [11], 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 debería haber sobre 30 casi, si no mas. #JJJ: Si. Muchas de esas referencias puedes sacarlas de los ultimos surveys de slicing.

2.1.1 Computing program slices with the system dependence graph

There exist multiple program representations, data structures and algorithms that can be used to compute a slice, but the most efficient and broadly used data structure is the *system dependence graph* (SDG), introduced by Horwitz et al. [12]. It is computed from the program’s source code, and once built, a slicing criterion is chosen and mapped on the graph, then the graph is traversed using a specific algorithm, and the slice is obtained. Its efficiency relies on the fact that, for multiple slices performed on the same program, the graph generation process is only performed once. Performance-wise, building the graph has quadratic complexity ($\mathcal{O}(n^2)$), and its traversal to compute the slice has linear complexity ($\mathcal{O}(n)$); both with respect to the number of statements in the program being sliced.

The SDG is a directed graph, and as such it has a set of nodes, each representing a statement in the program —barring some auxiliary nodes introduced by some approaches— and a set of directed edges, which represent the dependencies among nodes. Those edges represent several kinds of dependencies —control, data, calls, parameter passing, summary.

To create the SDG, first a *control flow graph* (CFG) is built for each method in the program, some dependencies are computed based on the CFG. With that data, a new graph representation is created, called the *program dependence graph* (PDG) [20]. Each method’s PDG is then connected to form the SDG. For a simple visual example, see Example 4 below, which briefly illustrates the intermediate steps in the SDG creation. The whole process is explained in detail in section 3.1.

Once the SDG has been created, a slicing criterion can be mapped on the graph and the edges are traversed backwards starting. The process is performed twice, the first time ignoring a specific kind of edge, and the second, ignoring another kind. Once the second pass has finished, all the nodes visited form the slice.

Example 4 (The creation of a system dependence graph). #SSS: Este ejemplo da demasiados detalles en cuanto a los grafos. Consider the code provided in Figure 2.1, where a simple Java program containing two methods (`main` and `multiply`) is displayed.

```

1 void main() {
2     multiply(3, 2);
3 }
4
5 int multiply(int x, int y) {
6     int result = 0;
7     while (x > 0) {
8         result += y;
9         x--;
10    }
11    System.out.println(result);
12    return result;
13 }
```

Figure 2.1: A simple Java program with two methods.

Now turn your attention to Figure 2.2#CCC: is this too personal? the second person is used in other places, but not as directly: a CFG has been created for each method. The CFG has a unique source node (without incoming edges) and a unique sink node (without outgoing edges), named “Entry” and “Exit”. In between, the statements are structured according to all possible executions that could happen.

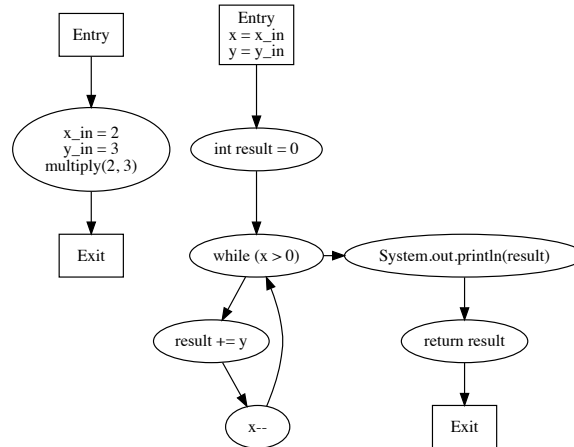


Figure 2.2: The control flow graphs for the code in Figure 2.1.

Next is Figure 2.3, which is a reordering of the CFG’s nodes according to the dependencies between statements: the PDG. Finally, both PDGs are connected into the SDG.

2.1.2 Program slicing metrics

In the area of program slicing, there exist many slicing techniques and tools implementing them. This fact has created the need to classify them by defining a set of metrics. These metrics are commonly associated to some features of the generated slices, or to the resources used by the slicing tool. The following list details 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 techniques and implemented tools set to achieve at least the generation of complete slices. There exists a trivial way of achieving completeness, by 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 (polymorphism, global variables, arrays, etc.), programming languages or paradigms a slicing tool is able to cover. There are slicing tools (publicly published or commercially available) for most popular programming languages, from C++

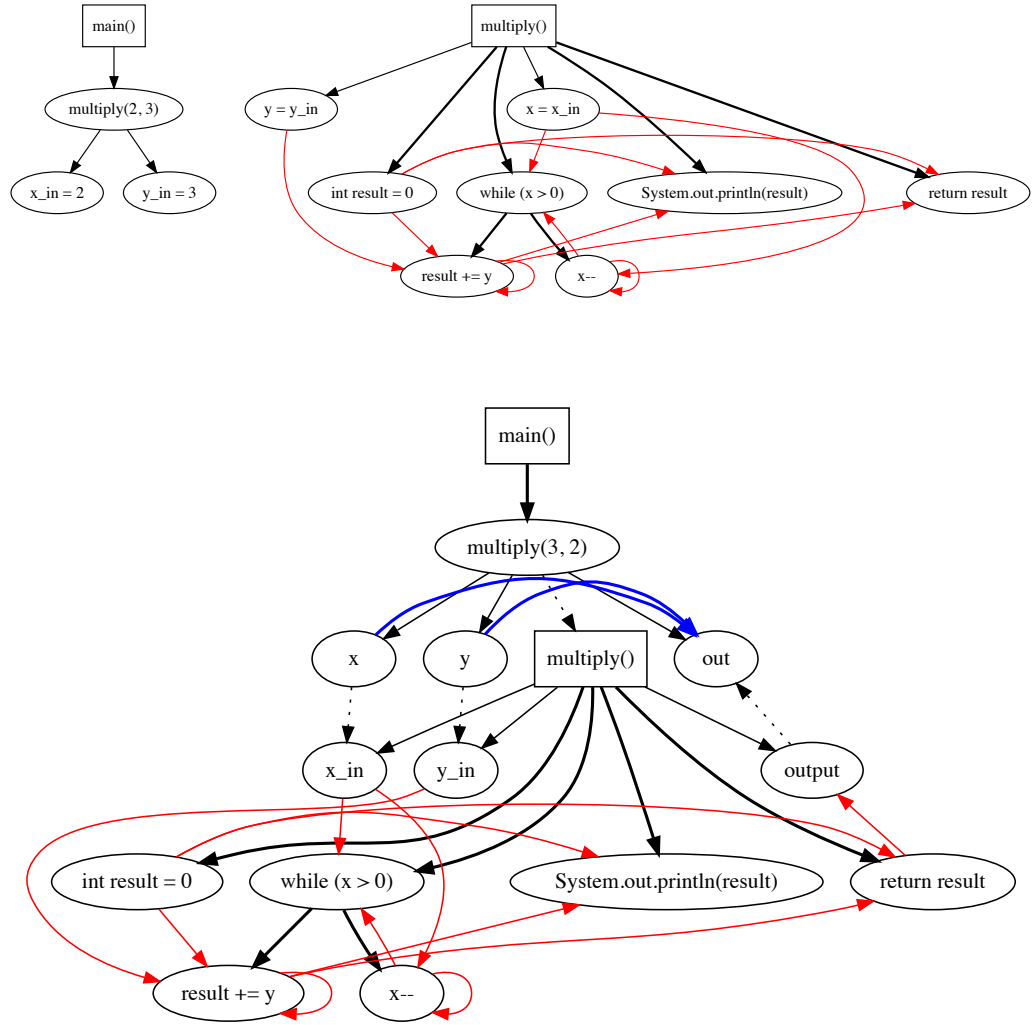


Figure 2.3: The program dependence graphs (above) and system dependence graph (below) generated from the code in Figure 2.1.

to Erlang. Some slicing techniques only cover a subset of the targeted language, and as such are less useful, but can be a stepping stone in the betterment of the field. There also exist tools that cover multiple languages or that are language-independent [6]. A small set-back of language-independent tools is that they are not as efficient in other metrics.

Resource consumption. Speed and memory consumption for the graph generation and slice creation. As previously stated, slicing is a two-step process: building a graph and traversing it, with the first process being quadratic and the second linear (in time). Proposals that build upon the SDG try to keep traversal linear, even if that means making the graph bigger or slowing down its building process.

Though this metric may not seem as important as others, program slicing is not a simple analysis. On top of that, some applications of software slicing like debugging constantly change the program and slicing criterion, which makes faster slicing software preferable for them.

Memory consumption is less relevant, mainly due to its availability, but could become a concern in big systems with millions of lines of code. [#CCC: Check this.](#)

2.1.3 Variations and applications of program slicing

As stated before, there are many uses for program slicing: program specialization, software maintenance, code obfuscation... but there is no doubt that program slicing is first and foremost a debugging technique. Program slicing can also be performed with small variations on the algorithm or on the meaning of “slice” and “slicing criterion”, so that it answers a slightly or totally different question. Each variation of program slicing answers a different question and serves a different purpose:

Backward static. Used to obtain the lines that affect the slicing criterion, normally used on a line which contains an incorrect value, to track down the source of the bug.

Forward static [9]. Used to obtain the lines affected by the slicing criterion, used to perform software maintenance: when changing a statement, slice the program w.r.t. that statement to discover the parts of the program that will be affected by the change.

Chopping static. Obtains both the statements affected by and the statements that affect the selected statement. [#CCC: Add application and verify question.](#)

Dynamic. Can be combined with any of the previous variations, and limits the slice to an execution history, only including statements that have run in a specific execution. The slice produced is much smaller and useful, but must be recomputed each time. It can be used for debugging when the input values that cause the error are known.

Quasi-static. In this slicing variant, some input values are given, and some are left unspecified: the result is a slice sized 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. It is another middle ground between static and dynamic slicing, similarly to quasy-static slicing. Likewise, it can offer a slightly bigger slice than pure dynamic slicing while keeping the scope focused on the slicing criterion and the set of executions.

There exist many more, which have been detailed in surveys of the field, such as [23], which analyzes the different dimensions that can be used to classify slicing techniques.

2.2 Exception handling in Java

Exception handling is common in most modern programming languages. It generally consists of a few new instructions used to modify the normal execution flow and later return to it. Exceptions are used to react to an abnormal program behaviour (controlled or not), and either solve the error and continue the execution, or stop the program gracefully. 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:

Throwable. An interface that encompasses all the exceptions or errors that may be thrown. Its two main implementations are **Error** for internal errors in the Java Virtual Machine and **Exception** for normal errors. The first ones are generally not caught, as they indicate a critical internal error, such as running out of memory, or overflowing the stack. The second kind encompasses the rest of exceptions that occur in Java. All exceptions can be classified as either *unchecked* (those that extend **RuntimeException** or **Error**) or *checked* (all others, may inherit from **Throwable**, but typically they do so from **Exception**). Unchecked exceptions may be thrown anywhere without warning, whereas checked exceptions, 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 (the **main** method) is popped, and the execution of the program ends abruptly. #CCC: Review [stopped here](#).

try. This statement contains a block of statements and one or more **catch** clauses and/or a **finally** block. All exceptions thrown in the statements contained or any methods called will be processed by the list of catches.

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.

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

Language	% usage	Language	% usage
JavaScript	69.7	C	17.3
HTML/CSS	63.1	Ruby	8.9
SQL	56.5	Go	8.8
Python	39.4	Swift	6.8
Java	39.2	Kotlin	6.6
Bash/Shell/PowerShell	37.9	R	5.6
C#	31.9	VBA	5.5
PHP	25.8	Objective-C	5.2
TypeScript	23.5	Assembly	5.0
C++	20.4		

Table 2.2: The most commonly used programming languages by professional developers³

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 Overflow². 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? 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#, PowerShell⁴, 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

²<https://stackoverflow.com>

³Data from <https://insights.stackoverflow.com/survey/2019/#technology--programming-scripting-and-markup-languages>

⁴Only since version 2.0, released with Windows 7.

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 runs, 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 **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 saved, 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 5 (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 authors⁵. 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

⁵For 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 ~~a method~~ ~~the main data structure for program representation used in the~~ ~~for program slicing~~ ~~area. It~~ ~~that~~ was first proposed by Horwitz, Reps and Blinkey [11] ~~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. ~~Volvemos a poner las siglas y su significado? CFG? PDG? ya se han puesto antes~~ The ~~system dependence graph (SDG~~ ~~)~~ 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 ~~each~~ CFG to PDG and then to ~~the final~~ SDG. The only modification possible is the redefinition of a ~~n already defined~~ dependency or the addition of new kinds of dependence.

The language covered by the initial proposal ~~was~~ ~~is~~ ~~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) ~~:~~ ~~, i.e.,~~ ~~no se si i.e., queda bien aqui :/~~ the basic features of an imperative programming language. The ~~control flow graph was~~ ~~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 (~~e.g.,~~ assignments, procedure calls) and predicates, whose execution may lead to one of multiple —though traditionally two— ~~different paths~~ (~~e.g.,~~ conditional instructions). ~~S~~ ~~While~~ statements are connected sequentially to the next instruction ~~.~~ ~~P~~ ~~, on the contrary,~~ predicates have two outgoing edges, each ~~of them~~ connected to the first statement that should be executed ~~,~~ according to the result of evaluating the conditional expression in the guard of the predicate.

Definition 7 (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 $e = (n_1, n_2) \mid n_1, n_2 \in N$. #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 digas 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 aqui Edges 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 formalmente, Debe ser TODO formalmente 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 definiendo In 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 8 (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 9 (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 10 (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 11 (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. $N' = N \setminus \{End\}$
2. $(a, b) \in E_c \iff a, b \in N' \wedge a \xrightarrow{ctrl} b \wedge \nexists c \in N' . a \xrightarrow{ctrl} c \wedge c \xrightarrow{ctrl} b$
3. $(a, b) \in E_d \iff a, b \in N' \wedge a \xrightarrow{data} b$

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 $n_1 \xrightarrow{ctrl} n_2$ **#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 $n_1 \xrightarrow{data} n_2$. Additionally, if a node n does not have any incoming control edges, it has a “default” control edge $e = (Start, n)$; so that “Start” is the only source node of the graph.

Note: **#JJJ: dentro de una definicion no pueden haber notas. Esto va fuera**the 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.

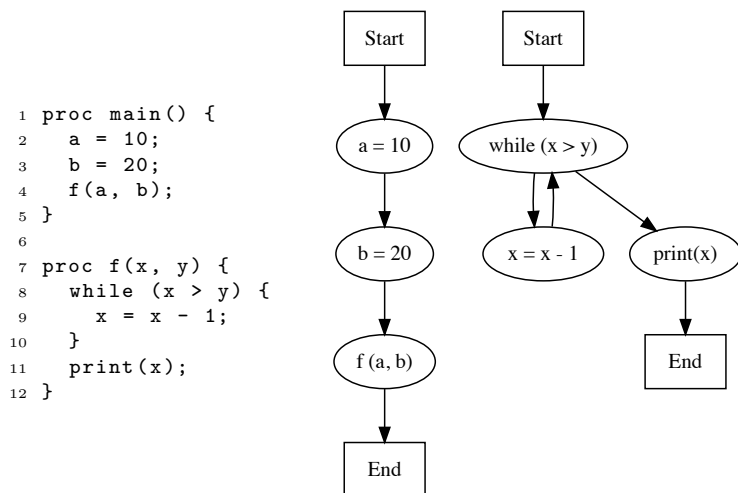
#SSS: creo que en la definicion de CFG y PDG tiene que quedar mas claro que hay varios por programa (uno por funcion), para que esta ultima frase cobre mas sentido.

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

Definition 12 (System dependence graph). Given a program P composed of a set of n methods $M = \{m_0 \dots m_n\}$ and their associated PDGs (each method m_i has a PDG $G_{PDG}^i = \langle N^i, E_c^i, E_d^i \rangle$), 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 .

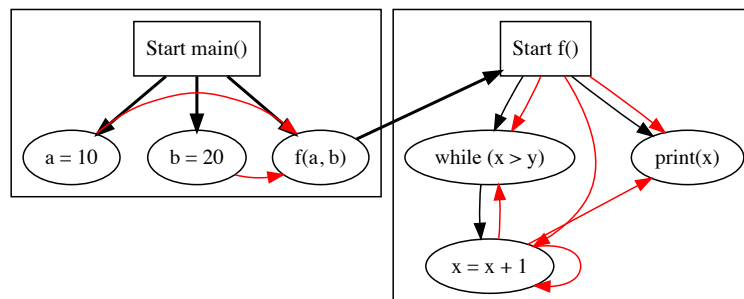
#JJJ: Arreglar esta definicion como la del PDG. Ahora mismo es totalmente informal. Deberia definirse encima del PDG. Es decir, una SDG es la conexion adecuada de varios PDGs, uno por método. Y solo definir lo nuevo: call arcs, parameter-in arcs, parameter-out arcs y summary arcs. The *system dependence 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 6 (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:



#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 problem~~ everything ~~#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:

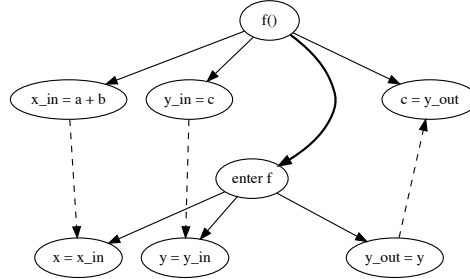
CFG. In each CFG, global variables read or modified and parameters are added to the label of the “Start” node in assignments of the form $par = par_{in}$ for each parameter and $x = x_{in}$ for global variables. Similarly, global variables and parameters modified are added to the label of the “End” node 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 $par_{in} = \text{exp}$ 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 definicion 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 ojeté 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-in y paramneter-out. Allí tendrá sentido y será aclaratorio. Aquí es confusorio. ;-)~~

Example 7 (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 9) 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 [7] 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 originalmente, ahora es originalmente.

#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 8 (Control dependencies generated by unconditional instructions). Figure 3.1 shows 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).

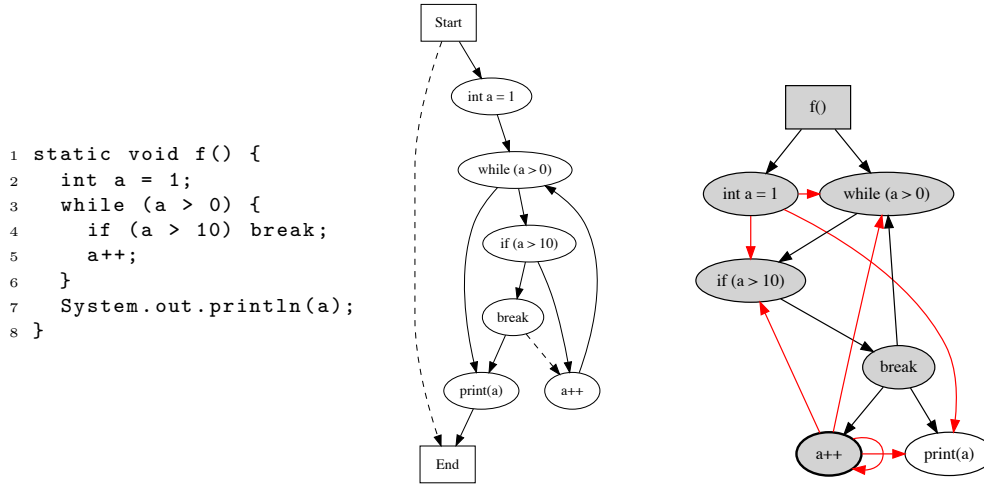


Figure 3.1: A program with unconditional control flow, its CFG (center) and PDG(right).

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 9. 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 9 (Nested unconditional jumps). #JJJ: Esta frase es difícil 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 revés: 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.

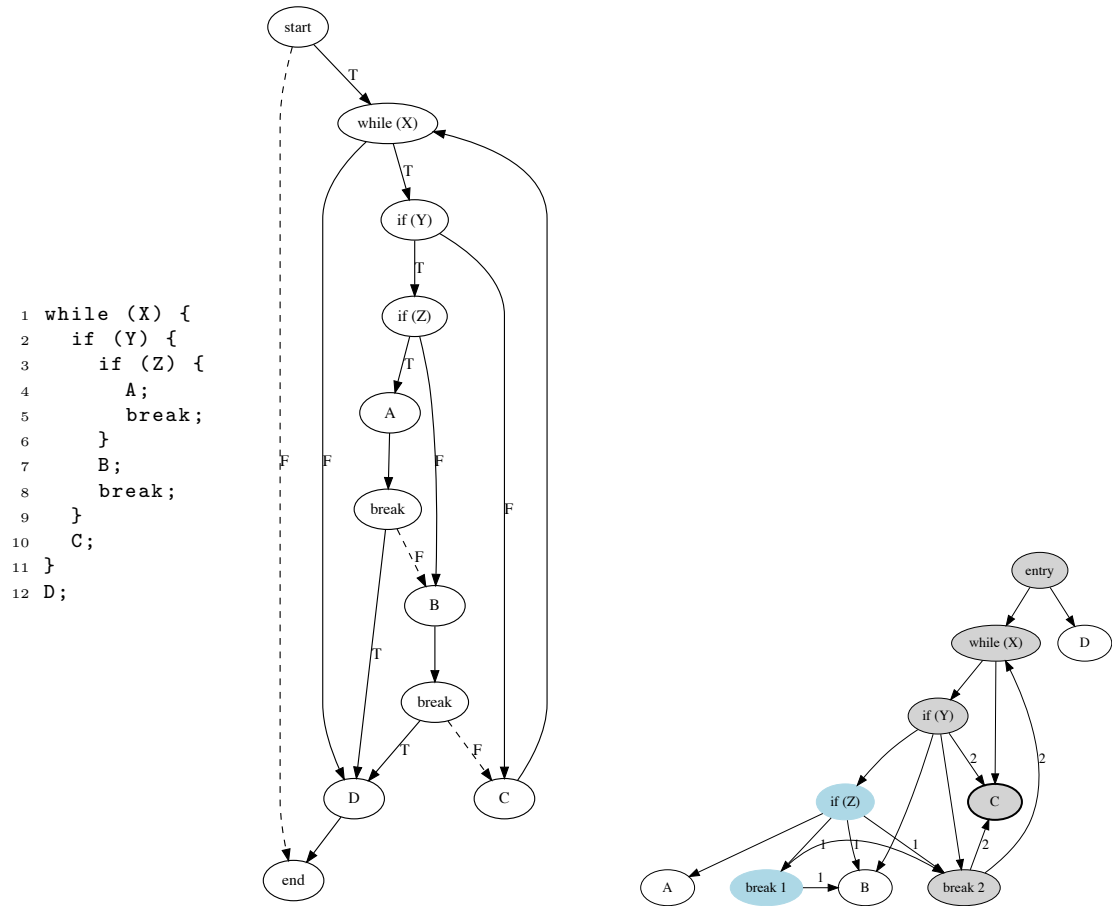


Figure 3.2: A program with nested unconditional control flow (left), its CFG (center) and [#JJJ: its](#) PDG (right).

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. [24], 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

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 unchecked 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 ≥ 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 larguissima, 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 false edge to the next `catch` block. 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 is not.

Example 11 (Catches.). Consider the following segment of Java code in figure 3.4 (left), which includes some statements that do not use data without any data dependence (X, Y and Z), and 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 error token when it occurs. Additionally, consider the case that when `f` exits without an error normally, only `x` is modified; but when an error occurs, only `y` is modified.

Note how the pseudo-statements act to create control dependencies between the true and false edges, such as the “normal return”, “catch”, “try”. 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` block, and inside the `try` block 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. 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.

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 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

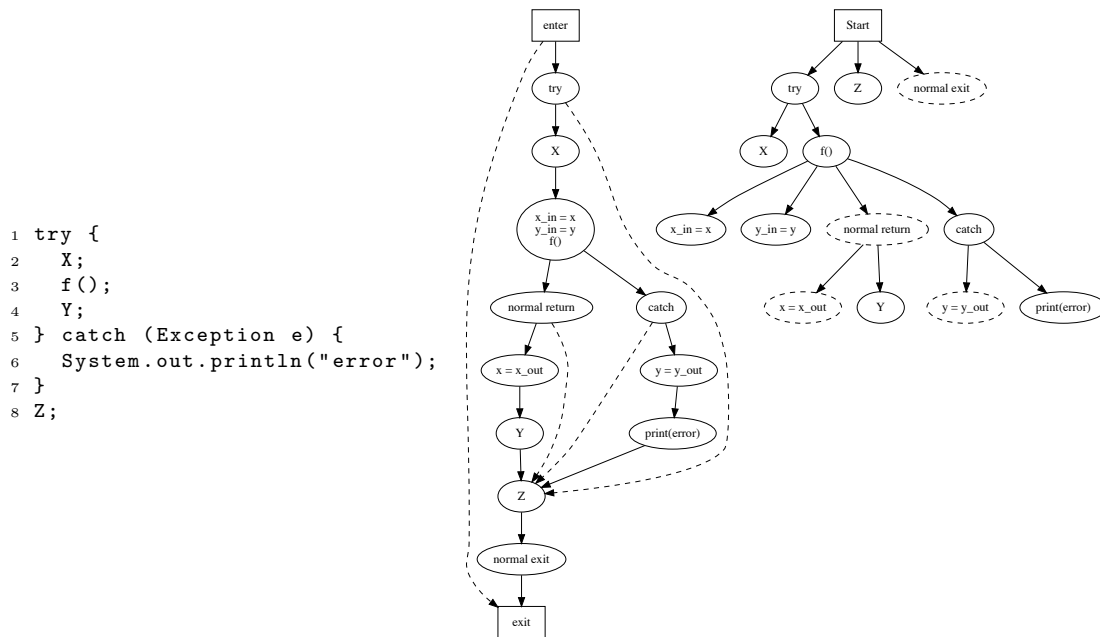


Figure 3.4: A simple example of the representation of **try-catch** structures and method calls that may throw exceptions. #JJJ: Pon quien es el CFG y quien el PDG. Por cierto, el arco del catch a la Z (rama false del catch) no es como los que se habian comentado. Es decir, no va a donde iria la ejecucion 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. Sobre todo al introducir 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 #SSS: Los propios autores lo comentaban? Si es asi no digo nada xD 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 #SSS: similar a quien? Es similar o el mismo con breaks? yo tal vez diria analogous.example —with `break` statements instead of `goto`— is shown in example 12.

Example 12 (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 #SSS: impacta que ahora digamos statements or expressions cuando llevamos todo el rato diciendo instructions. Casi diria que es la primera vez que nos referimos a expressions. Yo dejaria statements o instructions, uppercase letters are used; and no data dependencies are considered, as they are not relevant to the problem at hand.

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

Figure 4.1: A program (left), its computed slice (centre) and the #SSS: {smallest complete#SSS: } minimal slice (right).#SSS: echar un pelin a la derecha, los numeros quedan fuera del margen

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#SSS: Si decimos minimal slice tenemos que haber dicho que es en algun sitio, al menos definirlo. Lo hemos dicho?. 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 frase#SSS: yo entiendo que se refiere a que el inner break (6) tiene control sobre `B` (8), pero no sobre `C`. Yo creo que si que se entiende bienbut the dependence #Deleted: from line 5 to line 9#SSS: demasiados line tal line cual#Added: between both `break` statements introduces useless statements into the slice.

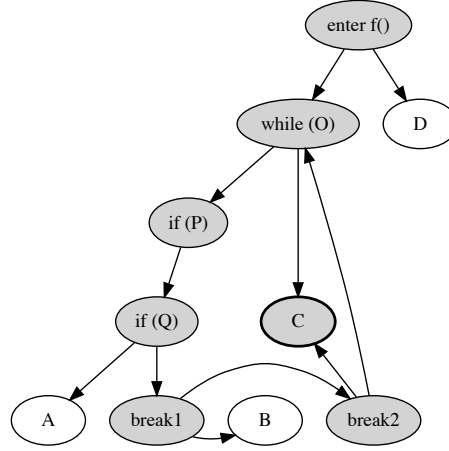


Figure 4.2: The system dependence graph for the program of figure 4.1, with the slice marked in grey, and the slicing criterion in bold. **#JJJ: En las condiciones pones O,P,Q en lugar de X,Y,Z** **#SSS: eso es porque la liaste en el brainstorming con la O P Q, y lo rompiste todo, Josep's fault!! xD**

The problem showcased in example 12 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 formal** if a program P contains a pair of unconditional jumps without any data **#Added: information** (e.g. `goto label`, `continue [label]`, `break [label]`, `return`) **#SSS: yo pondria 1, no los 4, que sino ya no es e.g. xD** 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 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. **#SSS: Esta definicion tiene varios vacios, estaba intentando proponer algo pero hay que definir varios conjuntos y es una definicion condicional del SC... propongo intentar hacer una mejor definicion entre los 3 el lunes**

#SSS: Dejo esto a medias a ver si partiendo de eso sacamos algo:

#SSS: Let P be a program, C_{block} a block contained in a conditional statement C of P , and j_A and j_B two unconditional jumps of P without any data (e.g. `goto label`) which jump to the statements A and B respectively.

#SSS: $j_B \in \text{Superfluous in the slice w.r.t. SC} \iff A = B \wedge j_B \in C_{Block} \wedge \nexists \text{ path } E_P \in \{CFG_P \text{ from } j_B \text{ to } SC\} . j_A \notin E_P$

#JJJ: #Deleted: Proposal A solution for the subsumption correctness error

As only the minimum amount of control edges are inserted into the PDG (according to definition 11), 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 quedaria el SDG)

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

#JJJ: Esta frase esta mal construida In the context of weak slicing, as #Deleted: it is not necessary to behave exactly like the original program #Added: shown in chapter 3??, the slicing criterion is not forced to behave in exactly the same way that the original program . This means that some statements may be removed, even if it #Deleted: means that a loop will become infinite #Added: results in an infinity loop execution, or an #Deleted: exception will not be caught #Added: uncaught exception behaviour. The following example describes a specific #Deleted: example #Added: scenario which is generalized later in this section.

Example 13 (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 #Added: slicing 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...) #SSS: {, but as that is a prefix #JJJ: suena raro que una lista infinita sea un prefijo de 0-9, mas bien es al reves of 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). #SSS: }. Fortunately, this represents no inconvenience in the context of weak slicing, since the values given to the slicing criterion for the original program —which is a list with the numbers 0 to 9— is a prefix of the values generated by the slice, fulfilling the requirements of definition ?? Creo que esto estaba en una definicion.

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 #Deleted: is #Added: was line 8, variable `x`, lines 4 and 5 #Deleted: are #Added: would be required to print the value, as without them, the program would loop indefinitely and never execute line 8.

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

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 **#Added:** if there is no statement $s \in S$ which is after X , meaning that there is a path from X to s in the CFG. **#SSS:** *Si formalizamos mas arriba aqui habria que hacer lo propio.*

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: Proposal A 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 incorrecta* Our proposal **#Deleted:** *revolves around temporarily remove* **#Added:** *is related to the temporal removal of* edges from the SDG: given an SDG of the form **#JJJ:** *En la definicion de SDG salia esta sextupla?* $G = \langle N, E_c, E_d, E_{in}, E_{out}, E_{fc} \rangle$, **#Added:** *we remove* from E_c any edge of the form $x \xrightarrow{ctrl} y \mid x, y \in N$, where x is an unconditional forward jump; **#Added:** *then,* **#Deleted:** *perform* the slice **#Added:** *is performed* normally; and **#Deleted:** *then* **#Added:** *finally* —if there is any statement **#Added:** *located* after the destination of x in the slice— **#Added:** *we restore the edges removed in the first step and recompute the slice.* **#SSS:** *no habia una solucion mejor que esta?, suena un poco a parche poco convincente* 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 9 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 14 (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`.

```
1 try {  
2   f();  
3 } catch (ExceptionA e) {  
4   log("Type_A");  
5 } catch (ExceptionB e) {  
6   log("Type_B");  
7 } catch (Exception e) {  
8   log("Exception");  
9 }  
10 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.

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 15 (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 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;

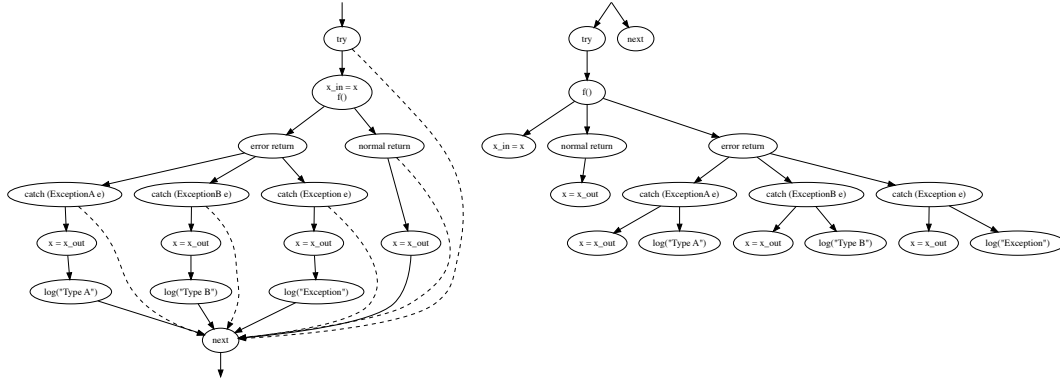


Figure 4.5: CFG (left) and PDG (right) of the code shown in figure 4.4.

```

1 try {
2   f();
3 } catch (Exception e) {
4   log("error");
5 }
6 x = 0;
7 f();

```

```

1 void f() throws Exception {
2   if (x % 2 != 0)
3     throw new Exception();
4   x++;
5 }

```

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`.

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 super-type (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 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

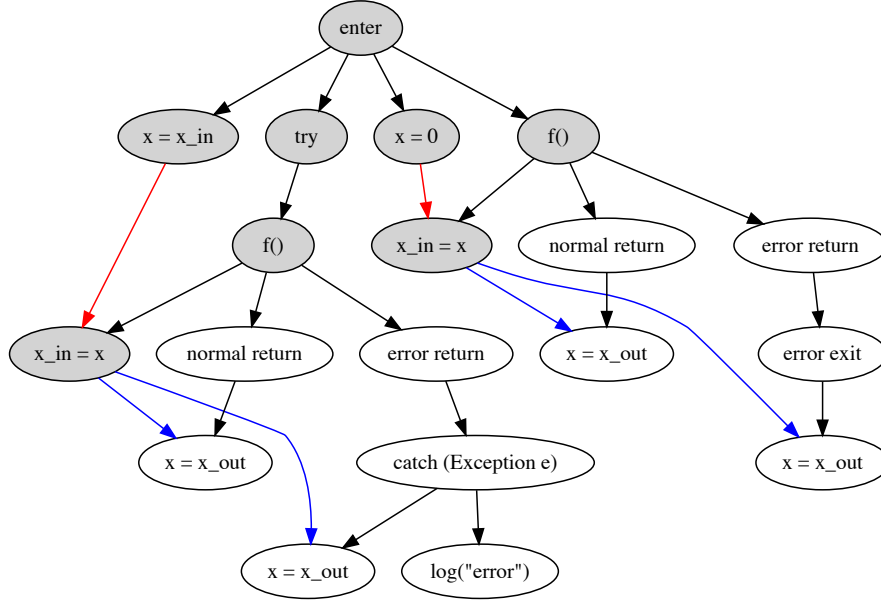


Figure 4.7: The system dependence graph of the left snippet of figure 4.6. `f` and the edges that connect to it are not shown for simplicity.

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.

Chapter 5

Related work

Slicing was proposed [27] 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 [24, 25] 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 `NullPointerException` 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 [16] 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. [14] 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 centered ~~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 [21] have worked specifically on the C++ exception framework. ~~CCC: remove or expand~~.

Finally, Hao [15] 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 [14], such as cascading *catch* statements, while adding explicit support for virtual calls, polymorphism and inheritance.

Alternative explanation of [2], with counter example. 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

```

1 void main() {
2     int x = 0;
3     while (true) {
4         try {
5             f(x);
6         } catch (ExceptionA e) {
7             x--;
8         } catch (ExceptionB e) {
9             System.err.println(x);
10        } catch (ExceptionC e) {
11            System.out.println(x);
12        }
13        System.out.println(x);
14    }
15 }
16
17 void f(x) {
18     x--;
19     if (x > 10)
20         throw new ExceptionA();
21     else if (x == 0)
22         throw new ExceptionB();
23     else if (x > 0)
24         throw new ExceptionC();
25     x++;
26     System.out.println(x);
27 }
28
29 static class ExceptionA extends ExceptionC {}
30 static class ExceptionB extends Exception {}
31 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] David Binkley and Keith Brian Gallagher. Program Slicing. *Advances in Computers*, 43(2):1–50, April 1996.
- [6] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent Program Slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 109–120, New York, NY, USA, 2014. ACM.
- [7] 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.
- [8] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. *SIGSOFT Softw. Eng. Notes*, 21(3):121–134, May 1996.
- [9] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug 1991.
- [10] Ákos Hajnal and István Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software Maintenance*, 24(1):67–82, 2012.
- [11] 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.
- [12] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.

- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.
- [18] Anirban Majumdar, Stephen J. Drape, and Clark D. Thomborson. Slicing obfuscations: Design, correctness, and evaluation. In *Proceedings of the 2007 ACM Workshop on Digital Rights Management, DRM '07*, pages 70–81, New York, NY, USA, 2007. ACM.
- [19] Claudio Ochoa, Josep Silva, and Germán Vidal. Lightweight program specialization via Dynamic Slicing. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP '05*, pages 1–7, New York, NY, USA, 2005. ACM.
- [20] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Software Engineering Notes*, 9(3):177–184, 1984.
- [21] 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.
- [22] Thomas Reps and Wu 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.
- [23] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3), June 2012.
- [24] 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.
- [25] 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.
- [26] Frank Tip. A survey of Program Slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
- [27] 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.