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

Program slicing is an analysis technique that can be applied to practically all programming languages. However, in the presence of exception handling, current program slicing software has a precision problem. This project tackles the problem of program slicing with exception handling, analysing the problem from a general perspective (for any kind of exception system), but focusing our efforts in the object-oriented paradigm, specifically the Java language.

In this thesis, we study the currently available solutions to the problem, and we propose a generalization that includes at least the `try-catch` and `throw` statements. We provide detailed descriptions, generalizations and solutions for two problems that increase the size of slices and one problem that greatly reduces the precision of slices. The solutions we propose produce slices that guarantee completeness and are as correct as possible, given the restrictions set by the exception handling system.

The analysis performed and solutions proposed are specific for the Java programming language, but are general enough that they can be ported effortlessly to other programming languages with similar exception handling capabilities. They are also specific for static backward slicing, but are likewise compatible with other variants of program slicing.

Resumen

La fragmentación de programas es una técnica de análisis de programas que puede ser aplicada prácticamente a todos los lenguajes de programación. Sin embargo, en presencia de excepciones, los fragmentadores de programas tienen un problema de precisión. Este proyecto aborda el problema de la fragmentación de programas en presencia de excepciones, analizando el problema desde una perspectiva general (para cualquier tipo de sistema de excepciones), pero concentrando nuestros esfuerzos en el paradigma de la orientación a objetos, más específicamente en el lenguaje Java.

En esta tesis, estudiamos las soluciones existentes al problema planteado, y proponemos una generalización que incluye por lo menos las instrucciones `try-catch` y `throw`. Damos descripciones detalladas, generalizaciones y soluciones a dos problemas que aumentan innecesariamente el tamaño de los fragmentos de programa y un problema que reduce bastante la precisión. Las soluciones que proponemos producen fragmentos que garantizan la completitud y son tan correctos como es posible, dadas las restricciones marcadas por el sistema de manejo de excepciones.

El análisis realizado y las soluciones propuestas son específicas para el lenguaje de programación Java y su sistema de manejo de excepciones, pero también son lo suficientemente generales como para poder ser empleadas en otros lenguajes de programación que posean un sistema de excepciones. También son específicas para la fragmentación estática hacia atrás, pero, del mismo modo, son compatibles con otras variantes en la fragmentación de programas.

Resum

La fragmentació de programes es una tècnica que pot ser aplicada practicament a qualsevol llenguatge de programació. No obstant això, en presència de tractament d'excepcions, el software actual de fragmentació de programes presenta problemes de precisió. Aquest projecte aborda el problema de fracmentació de programes amb tractament d'excepcions, analitzant el problem des d'una perspectiva general (per a qualsevol tipus de sistema d'excepcions), però centrant els nostres esforços en un paradigma orientat a objectes, specificament per al llenguatge de programació Java. Tot i així, la solució es encara suficientment general com per a aplicarla a altres paradigmes i llenguatges de programació.

En aquesta tesi, estudiem les actuals solucions disponibles per al problema, i proposem una generalització que inclueix al menys les instruccions `try-catch` i `throw`. Proveïm descripcions detallades, generalitzacions i solucions per a dos problemes que incrementen el tamany dels fragments de programa obtinguts i un problema que redueix enormement la precisió dels fragments calculats. Les solucions que proposem produeixen fragments de programa que garanteixen completitut i que con el mes correctes posibles, donat el conjunt de restriccions del sistema de tractament d'excepcions.

Els anàlisis realitzats i les solucions proposades son específiques per al llenguatge de programació Java, pero son suficientment generals per a ser exportades sense esforç a altres llenguatges de programació ambs un sistema de tractament d'excepcions similar. Les solucions també son específiques per a fragmentació de programes estàtica cap arrare, pero son igualment compatibles amb altres variantes de la fragmentació de programes.

Contents

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* [29, 28]. 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 [9], program specialization [21], software maintenance [11], code obfuscation [20], etc. This technique was originally defined [29] 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 $\langle 5, x \rangle$ (which represents variable x in line 5), 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.

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

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

Figure 1.1: A simple Java method (left) and its slice w.r.t. slicing criterion $\langle 5, x \rangle$.

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 CodeSonar[1].

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 there are several scenarios, such as arrays, polymorphism and inheritance, and exception handling that are quite difficult to analyse. This is the reason why a universal solution does not exist for all the problems in the field of program slicing. Conversely, there are many approaches 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.

Even though the original proposal by Weiser [29] focused on an imperative language, program slicing is a language-agnostic technique. Since then, the literature has been expanded by dozens of authors, that have described and implemented program slicing for more complex structures, such as uncontrolled control flow [12], exception handling [3]; and for other programming paradigms, such as object–oriented languages [19].

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 [3, 15], 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 [3]; with respect to the slicing criterion $\langle 17, a \rangle$.

```
 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 }
```

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

Figure 1.2: A simple Java program with exception (left) and its slice w.r.t. $\langle 17, a \rangle$ (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 [18]. 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

[1]Created by GrammaTech. For more information, consult their website at `https://www.gramatech.com/`

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 [12] computed using Allen et al.'s algorithm [3] and 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.

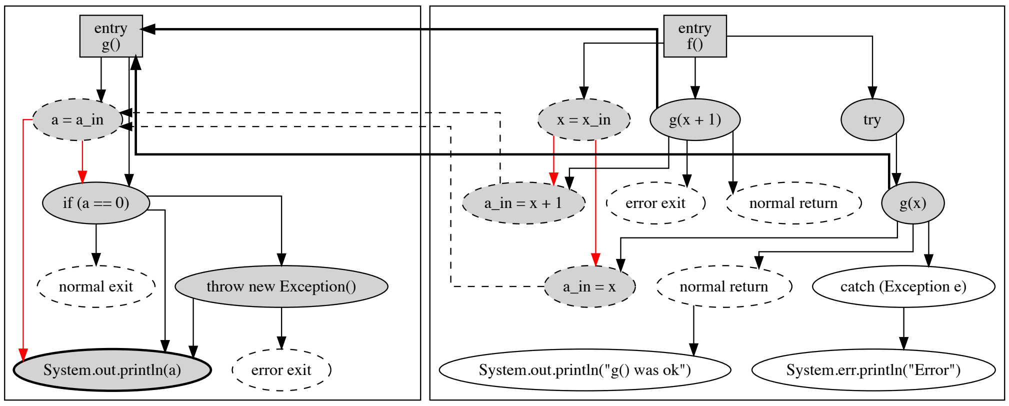


Figure 1.3: The system dependence graph for the method shown in Figure 1.2.

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. [3] 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. [3]. 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 leaded 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.

Definition 1 (Slicing criterion)**.** Given a program P, composed of statements and containing variables $x_1, x_2...x_n \in$ vars, a *slicing criterion* is a tuple $\langle s, v \rangle$ where $s \in P$ is a single statement that belongs to the program, and v is a set of variables from P.

The reader should note that the variables in v may not appear in s.

Definition 2 (Execution history)**.** Given a program P, composed of a set of statements $S = \{s_1, s_2, s_3...s_n\}$, 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.

Program slicing is the process of extracting a slice given a program and a slicing criterion. A *slice* is a subset of statements of a program which behaves like the original program, at the slicing criterion.

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. The variation described until now 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 [25]:

- *Static* or *dynamic*: slicing can be performed statically or dynamically. *Static slicing* [25] produces slices that consider all possible executions of the program: the slice will be correct regardless of the input supplied. In contrast, *dynamic slicing* [18, 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 bugs 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.

- *Backward* or *forward*: *backward slicing* [25] looks for the statements that affect the slicing criterion. It sits among the most commonly used slicing technique. In contrast, *forward slicing* [5, 10] computes the statements that are affected by the slicing criterion. There also exists a middle-ground approach called *chopping* [14], 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 [29], 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. First, though, we need to introduce and additional concept: the sequence of values.

Definition 3 (Sequence of values [23])**.** Let P be a program and $\langle s, v\rangle$ be a slicing criterion of P. $\mathit{seq}(P, s, v)$ is the sequence of values the slicing criterion v is evaluated to, at s, during the execution of P.

Definition 4 (Strong static backward slice [29, 10])**.** 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 possible input, $\mathit{seq}(P, s, v) = \mathit{seq}(S, s, v)$.

Definition 5 (Weak static backward slice [6])**.** Given a program P and a slicing criterion $\langle s, v\rangle$, 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 possible input, $\mathit{seq}(P, s, v)$ is a prefix of $\mathit{seq}(S, s, v)$.

Both Definition 4 and Definition 5 are used throughout the literature. Most publications do not differentiate them, as they work with one of them without acknowledging the other variant. Therefore, although the definitions come from different authors, the *weak* and *strong* nomenclature employed throughout this thesis originates from a control dependence analysis by Danicic [8], 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,

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.

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 obtained with respect to different slices of a program and the same slicing criterion.

The first row stands for the original program's sequence of values, which computes 3!.

Slice A's sequence of values is identical to the original and therefore it is a strong slice.

Slice B's sequence 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 sequence differs from the original program in the second column. It seems that some dependence has not been accounted for and the value is not updating.

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. [13]. 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) [22]. 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)**.** Consider the code provided in Figure 2.1, where a simple Java program containing two methods (`main` and `multiply`) is displayed.

```
void main() {
  multiply(3, 2);
}

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

Figure 2.1: A simple Java program with two methods.

Figure 2.2 contains one CFG per method. Each CFG has a unique source node (without incoming edges) and a unique sink node (without outgoing edges), named “Enter” and “Exit”. In between, the statements are structured according to all possible executions that could happen according to Java’s semantics.

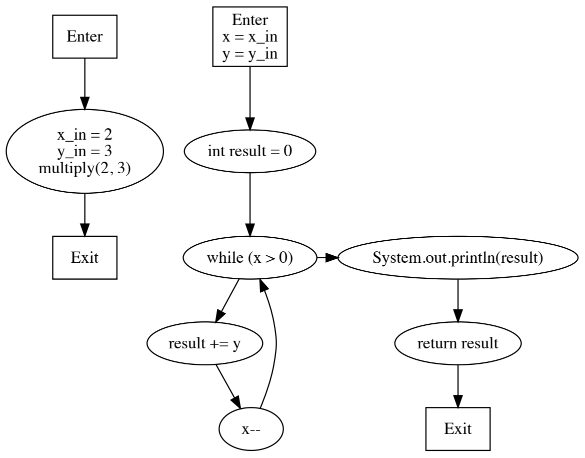
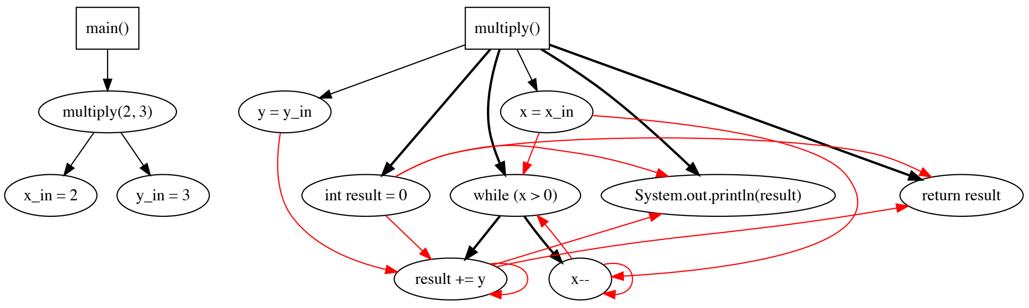


Figure 2.2: The control flow graphs for the code in Figure 2.1.

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



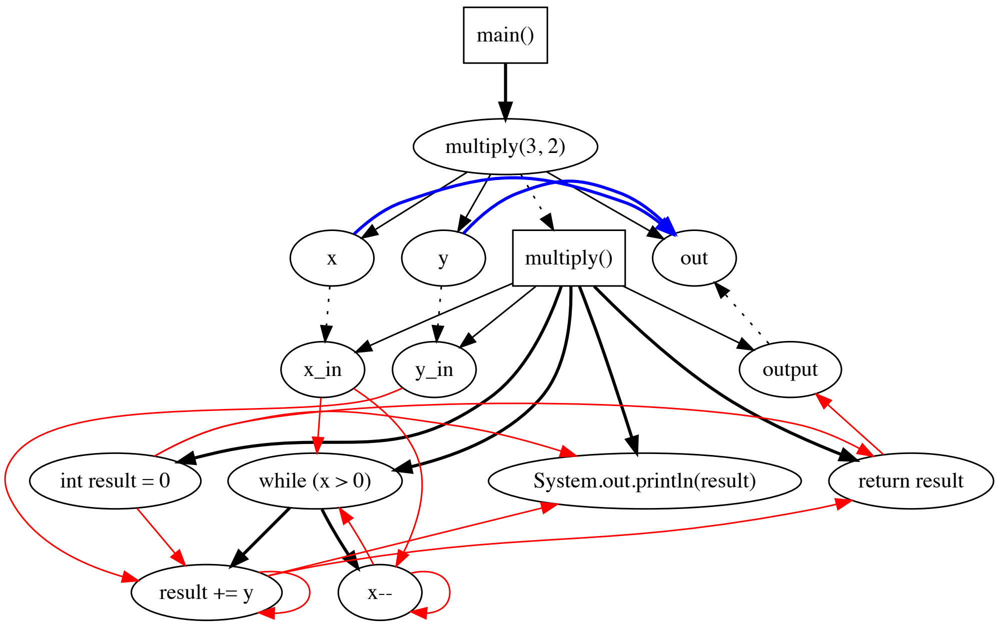


Figure 2.3: The program dependence graphs (above) and system dependence graph (below) generated from the code in Figure 2.1.

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++ 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 [7]. A small set-back of language-independent tools is that they are not as efficient in other metrics.

Performance. 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 lineal (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.

Regarding memory consumption, it is not currently a problem, given that the amount available in most workstations and servers is enough to run any slicing algorithm. It could become a concern in big programs with millions of lines of code, or in embedded systems, where memory is scarce.

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. 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. Given two slicing criteria, it obtains the intersection between the statements affected by the first criterion and the statements that affect the second criterion. It is mainly used for debugging applications.

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 quasi-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 [25].

2.2 Exception handling

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.

2.2.1 Exception handling in Java

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. A type 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` statement for its type or any super type, the control flow will continue in the first statement inside the `catch` statement. 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.

try. This statement contains a block of statements and one or more `catch` statement and/or a `finally` statement. All exceptions thrown in the statements contained or any methods called will be processed by the list of `catch` statements. If no `catch` matches the type of the exception, the exception propagates to the `try` block that contains the current one, or, in its absence, the method that called the current one.

catch. Contains two elements: a variable declaration, whose type must extend from `Throwable`, and a block of statements to be executed when an exception of a matching type is thrown. The type of a thrown exception T_1 matches the type of a `catch` statement T_2 if one of the following is true: (1) $T_1 = T_2$, (2) T_1 extends T_2, (3) T_1 extends $T \wedge T$ matches T_2. `catch` statements are processed sequentially, although their order does not matter, due to the restriction that each type must be placed after all of its subtypes. When a matching `catch` is found, its block is executed and the rest are ignored. Variable declarations may be of multiple types `(T1|T2 e)`, when two unrelated types of exception must be caught and the same code executed for both. If there is an inheritance relationship, the parent suffices.[1]

finally. Contains a block of statements that will always be executed, no matter what, if the *try* is entered. It is used to tidy up, for example closing I/O streams. The `finally` statement 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` 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.

2.2.2 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 programming languages feature some kind of error system, which 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 list of languages to be analysed has been extracted from the results of a survey performed by the programming Q&A website Stack Overflow[2]. The survey contains a question about the technologies used by professional developers in their work, and from that list we have extracted those languages with more than 5% usage in the industry. Table 2.2 displays the list and its source. All languages displayed there feature an exception system similar to Java's, except for Bash, Assembly, VBA, C and Go[3].

The exception systems that are similar to Java are mostly all the same, featuring a `throw` statement (i.e. `raise` in Python), `try-catch`-like structure, and most include a `finally` statement that may be appended to `try` blocks. The difference resides in the value passed by the exception. In programming languages with inheritance and polymorphism, the value is restricted to any type that extends a generic error type (e.g. `Throwable` in Java). The exceptions are filtered using types. In languages without inheritance, the value is an arbitrary one (e.g. JavaScript, TypeScript), with the exceptions being filtered using a boolean condition or pattern

[1]Only available from Java 7 onward. For more details, see `https://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html` (retrieved November 2019).

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

[3]PowerShell only features an exception system since version 2.0, released alongside Windows 7.

[4]From a survey on software developers by StackOverflow. Source: `https://insights.stackoverflow.com/survey/2019/#technology-_-programming-scripting-and-markup-languages` (retrieved November 2019).

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

Table 2.2: The most commonly used programming languages by professional developers[4]

to be matched (e.g. JavaScript). In both cases there exists a way to indicate that all possible exceptions should be caught, regardless of type and content.

Regarding the languages that do not offer an exception handling mechanism similar to Java's, error-handling is covered by a variety of systems, which are briefly detailed below.

Bash. The popular Bourne Again SHell features no exception system, apart from the user's ability to check 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. In essence, traps allow the programmer to react to a user or system–sent signal, or an exit run from within the Bash environment. When a trap is activated, its code run, and the signal does not proceed and stop the program. This does not replace a fully featured exception system, but `bash` programs tend to be short, with programmers preferring the efficiency of C or the commodities of other high–level languages when the task requires it.

VBA. Visual Basic for Applications is a scripting programming language based on Visual Basic that is integrated into Microsoft Office to automate small tasks, such as generating documents from templates, making advanced computations that are impossible or slower with spreadsheet functions, etc. The only error–correcting system it has is the directive `On Error` x, where x can be 0—lets the error crash the program—, `Next`—continues the execution as if nothing had happened—or a label in the program—the execution jumps to the label in case of error. The directive can be set and reset multiple times, therefore creating artificial `try-catch` blocks, but there is no possibility of attaching a value to the error, lowering its usefulness.

C. In C, errors can also be controlled via return values, but some instructions featured in it can be used to create a simple exception system. `setjmp` and `longjmp` are two instructions which set up and perform inter–function jumps. The first makes a snapshot of the call stack in a buffer, and the second returns to the position where the buffer was safe, destroying the current state of the stack and replacing it with the snapshot. Then, the execution continues from the evaluation of `setjmp`, which returns the second argument passed to `longjmp`. Example 5 shows this system in action.

Example 5 (User-built exception handling system in C).
Consider Figure 2.4: 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

```c
int main() {
  if (!setjmp(ref)) {
    res = safe_sqrt(x, ref);
  } else {
    // Handle error
    printf /* ... */
  }
}
```

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

Figure 2.4: A simple `main` method (left) with an emulated `try-catch` and a method that computes a square root (left), emulating a `throw` statement if the number is negative.

`safe_sqrt` is run, returning the second argument of `longjmp`, and therefore entering the else block in the `main` method.

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

[5]For more details on Go's design choices, see `https://golang.org/doc/faq#exceptions` (retrieved November 2019).

Chapter 3

Program slicing with exception handling

3.1 First definition of the SDG

The SDG is the most common data structure for program representation in the field of program slicing. It was first proposed by Horwitz et al. [12] and, since then, many approaches to program slicing have based their models on it. It builds upon the existing CFG, which represents the control flow between the statements of a method. Then, it creates a PDG using the CFG's vertices and the dependencies computed from it. The SDG is finally built from the assembly of the different method's PDGs, linking each method call to its corresponding definition. Because each graph is built from the previous one, new statements and statements 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 an already defined dependence or the addition of new kinds of dependence.

The seminal appearance of the SDG covers a simple imperative programming language, featuring procedures and basic statements like calls, variable assignments, arithmetic and logic operators and conditional statements (branches and loops).

Definition 6 (Control Flow Graph (based on [2]))**.** Given a method M, which contains a list of statements $s = \{s_1, s_2, ...\}$, the *control flow graph* of M is a directed graph $G = \langle N, E \rangle$, where:

- $N = s \cup \{\text{Enter}, \text{Exit}\}$: a set of nodes such that for each statement s_i in s there is a node in N labelled with s_i and two special nodes "Enter" and "Exit", which represent the beginning and end of the method, respectively.

- E is a set of edges of the form $e = (n_1, n_2) \,|n_1, n_2 \in N$. There exist edges between normal statements, in the order they appear in the program: the "Enter" node is connected to the first statement, which in turn is connected to the second, etc. Additionally, conditional statements (i.e., `if`) have two outgoing edges: one towards the first statement executed if the condition evaluates to *true* and another towards the first statement if the condition evaluates to *false.*

Most algorithms, in order to generate the SDG, mandate the "Enter" node to be the only source and the "Exit" node to be the only sink in the graph. In general, expressions are not evaluated when generating the CFG; so an `if` conditional statement will two outgoing edges regardless the condition value being always true or false (e.g., `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. First, though, we need to define the concept of postdominance in a graph, as it is necessary in the definition of control dependence:

Definition 7 (Postdominance [28])**.** Let $C = (N, E)$ be a CFG. $b \in N$ *postdominates* $a \in N$ if and only if b is present on every possible sequence from a to "Exit".

From the previous definition, given that the "Exit" node is the only sink in the CFG, every node will have a path to it, so it follows that any node postdominates itself.

Definition 8 (Control dependence [12])**.** Let $C = (N, E)$ be a CFG. $b \in N$ is *control dependent* on $a \in N$ ($a \rightarrow^{ctrl} b$) if and only if b postdominates one but not all of $\{n \mid (a, n) \in E, n \in N\}$ (a's successors).

It follows that a node with less than two outgoing edges cannot be the source of control dependence.

Definition 9 (Data dependence [12])**.** Let $C = (N, E)$ be a CFG. $b \in N$ is *data dependent* on $a \in N$ ($a \rightarrow^{data} b$) if and only if a may define a variable x, b may use x and there exists in C a sequence of edges from a to b where x is not defined.

Data dependence was originally defined as flow dependence, and subcategorized into loop-carried and loop-independent flow-dependencies, but that distinction is no longer used to compute program slices with the SDG. 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 now be built by replacing the edges from the CFG by data and control dependence edges. The first tends to be represented as a thin dashed line or a thin solid coloured line; and the latter as a thin solid black line. In the examples, data and control dependencies are represented by red and black solid lines, respectively.

Definition 10 (Program dependence graph)**.** Given a method M, composed of statements $S = \{s_1, s_2, ...s_n\}$ and its associated CFG $C = (N, E)$, the *program dependence graph* (PDG) of M is a directed graph $G = \langle N', E_c, E_d \rangle$, where:

1. $N' = N \setminus \{\text{Exit}\}$
2. $(a, b) \in E_c \iff a, b \in N' \land (a \rightarrow^{ctrl} b \lor a = \text{Enter}) \land \nexists c \in N' \,.\, a \rightarrow^{ctrl} c \land c \rightarrow^{ctrl} b$ (*control edges*)
3. $(a, b) \in E_d \iff a, b \in N' \land a \rightarrow^{data} b$ (*data edges*)

Regarding the graphical representation of the PDG, the most common one 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. An example of the creation of the PDGs of a program's methods can be seen in Example 6.

Example 6 (Creation of a PDG from a simple program)**.** Consider the program shown on the left side of Figure 3.1, where two procedures in a simple imperative language are shown. The CFG that corresponds to each procedure is shown on the right side.

Then, the nodes of each CFG are rearranged, according to the control and data dependencies, to create the corresponding PDGs. Both are shown in Figure 3.2, each bounded by a rectangle.

Before creating the SDG by joining the different PDGs, we must consider the treatment of method calls and their data dependencies.

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

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

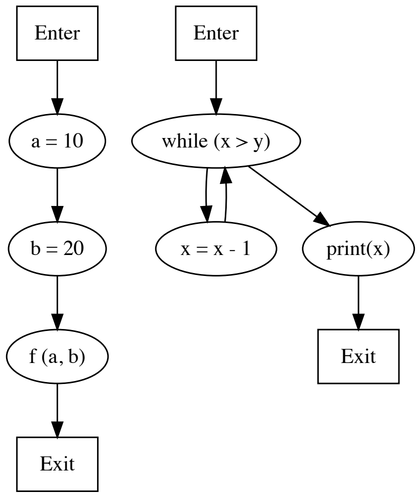


Figure 3.1: A simple imperative program composed of two procedures (left) and their associated CFGs (right).

Method calls and data dependencies

Although it is not imperative, since the inception of the SDG, data input and output from method calls[1]has been treated with special detail. A similar system is used for a method input (parameters) and output (return value) as with the global variables it can access (static variables and fields from a class in Java). Method calls can access global variables and modify them, and to that end we must add fictitious nodes that represent variable input and output from the methods in both the method calls and their declarations. This proposal can also be extended to those programming languages that pass parameters by reference instead of the more common pass-by-value. Java objects and arrays can also be analysed more deeply, as even though Java passes parameters by value, modifications to fields of an object or elements of an array affect the original object or array.

In practice, the following modifications are made to the different graphs:

CFG. The CFG's structure is not modified, as the control flow is not altered by the treatment of variables. Instead, some labels are extended with extra information, which is later used in the PDG's creation. Specifically, the "Enter" node, the "Exit" node and nodes that contain method calls are modified:

Enter. Each global variable that is used or modified and every parameter are appended to the node's label in assignments of the form $par = par_{in}$ in the case of parameters and $x = x_{in}$ in the case of global variables. These lines are the input information, and will become the input nodes.

End. Each global variable that is modified and every parameter whose modification can be read by the caller are prepended to the node's label. The assignments take the

[1]Method calls in this thesis will refer to Java method calls, but most if not all the details provided apply to functions, procedures and other routines.

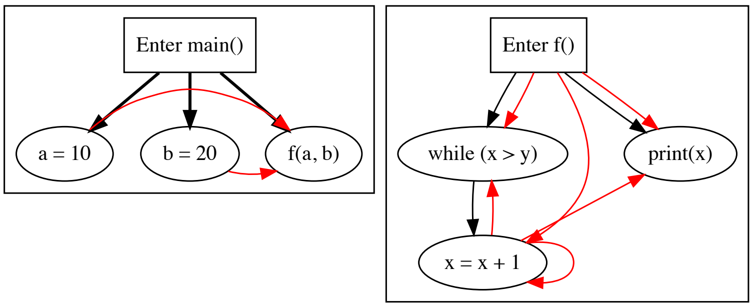


Figure 3.2: The PDG that corresponds to the program from Figure 3.1.

form $x_{out} = x$ for both. The method's output is also added, if the method will return a value, as `output`. These lines constitute the output information, and will be transformed into output nodes.

Method call. Each method call must be preceded by the input information and followed by the output information of the corresponding method. The input takes the form $par_{in} = \text{exp}$ for each parameter and $x_{in} = x$ for each global variable x. The output is always of the form $x = x_{out}$, except for the output of the function, which is labelled `output`.

PDG. Each node augmented with input or output information in the CFG is now split into multiple nodes: the original label ("Enter", "Exit" or function call) is the main node and each assignment contained in the input and output information is represented as a new node, which is control-dependent on the main one.

Now that method calls are properly handled, the SDG can be defined as the combination of PDGs, with the addition of four dependencies that connect the method calls and their definitions.

Definition 11 (System dependence graph)**.** Given a program P, composed of a set of methods $M = \{m_0...m_n\}$ and their associated PDGs—each method m_i has a $PDG^i = \langle N^i, E_c^i, E_d^i \rangle$. The *system dependence graph* (SDG) of P is a graph $G = \langle N, E_c, E_d, E_{call}, E_{in}, E_{out}, E_{sum} \rangle$ where:

1. $N = \bigcup_{i=0}^{n} N^i$
2. $E_c = \bigcup_{i=0}^{n} E_c^i$
3. $E_d = \bigcup_{i=0}^{n} E_d^i$
4. $(a, b) \in E_{call}$ if and only if a is a statement that contains a call and b is a method "Enter" node of the function or method called by a. (a, b) is a *call edge.*
5. $(a, b) \in E_{in}$ if and only if a and b are input nodes which refer to the same variable or parameter, $m_{call} \to^{ctrl} a \land m_{enter} \to^{ctrl} b \land (m_{call}, m_{enter}) \in E_{call}$ (m_{call} is a method call, m_{enter} is an "Enter" node). (a, b) is a *parameter-input* or *param-in edge.*

6. $(a, b) \in E_{out}$ if and only if a and b are output nodes which refer to the same variable or to the output, $m_{enter} \rightarrow^{ctrl} a \wedge m_{call} \rightarrow^{ctrl} b \wedge (m_{call}, m_{enter}) \in E_{call}$ (m_{call} is a method call, m_{enter} is an "Enter" node). (a, b) is a *parameter-output* or *param-out edge.*
7. $(a, b) \in E_{sum}$ if and only if a is an input node and b is an output node, $m_{call} \rightarrow^{ctrl} a \wedge m_{call} \rightarrow^{ctrl} b$, m_{call} is a node that contains a method call and there is a path from a to b. (a, b) is a *summary edge.*

Regarding call edges, in programming languages with ambiguous method calls (those that have polymorphism or pointers), there may exist multiple outgoing call edges from a statement with a single method call. To avoid confusion, the "Enter" nodes of each method are relabelled with their method's name.

Example 7 (The creation of a system dependence graph)**.** For simplicity, we explore a single small method that is called by another. Let $f(x, y)$ be a method with two integer parameters that modifies the argument passed in its second parameter. Its code is displayed in Figure 3.3. It also uses a global variable z. A valid call to f could be $f(a + 1, b)$, with parameters passed by reference when possible.

```
void f(int x, int y) {
  z += x;
  y++;
}
```

Figure 3.3: A simple method that modifies a parameter and a global variable.

The CFG is very simple, with the addition of the parameter information to the labels of the nodes. The aforementioned method call would be labelled as "$z_{in} = z$, $x_{in} = a + 1$, $y_{in} = b$, $f(a + 1, b)$, $b = y_{out}$, $z = z_{out}$", with the inputs, the actual call and the outputs.

The PDG seems more complicated, but can be pieced together piece by piece. In Figure 3.4, the PDG is the graph below and including the node "Enter f". First, the input and output information is extracted into nodes, and placed in order. The input nodes will generate data dependencies (shown in red) to the statements inside the method, and those in turn to the output nodes. All statements are control-dependent on the "Enter" node, as there are no conditional expressions.

Finally, if we connect the PDG of the method that contains the method call $f(a + 1, b)$ to the method's PDG we obtain the SDG (where shown partially, as the method containing the method call has not been detailed). There are param-in and param-out dependencies (shown with dashes), which connect each input node from the method call to its corresponding node from the method declaration (and vice versa for the outputs). There is also the call edge, which connects the actual call to the declaration, and finally there are the summary edges, which of course summarize the dependencies that exist between the input and output nodes inside the method.

3.2 Creating slices with the SDG

Once a SDG has been built, it can be traversed to create slices, without the need to rebuild it unless the underlying program changes. The traversal process is actually consists of two passes:

The node that corresponds to the statement in the slicing criterion is selected as the initial node. From there, all edges except for *param-in* are traversed backwards. All nodes encountered

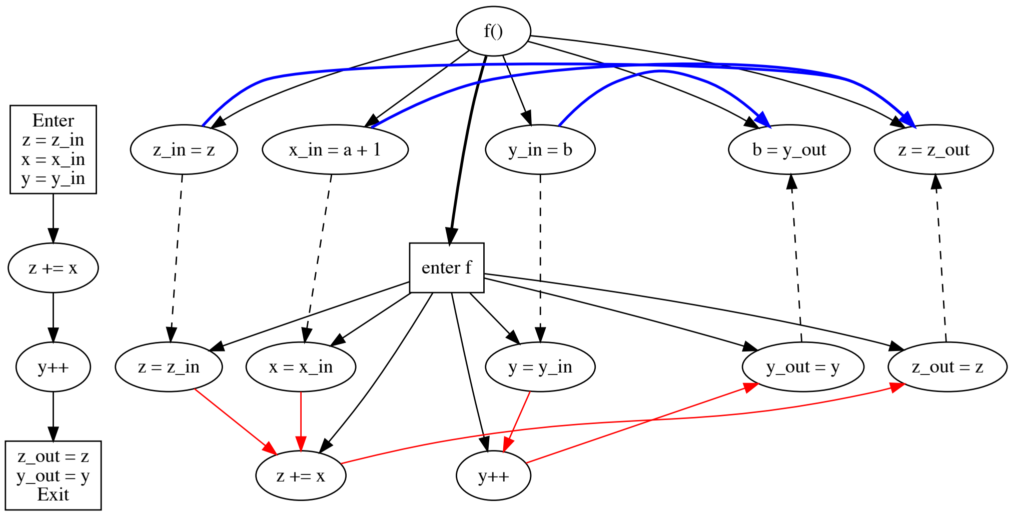


Figure 3.4: The CFG of f from Figure 3.3 (left) and its SDG (right).

are added to a set (the slice). When all possible edges have been traversed, the second pass begins, ignoring *param-out* edges, adding the nodes found to the aforementioned set. When the process has ended, the set of nodes encountered during the two-pass traversal constitutes the slice.

Along this thesis there are some examples where the SDG has been sliced, filling the nodes in grey and marking the slicing criterion in bold. Some are Example 2, Example 8, Example 11 and Example 14.

3.3 Unconditional control flow

Even though the initial definition of the SDG was 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 additions contributed to the algorithm to build 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 statement (e.g., outside the loop); or, alternatively, to represent the statement as an edge, not a vertex, connecting the previous statement with the next to be executed. Both of these approaches fail to generate a control dependence from the unconditional jump, as the definition of control dependence (see Definition 8) 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 dependence, in order to reflect the real effect of these statements—as some authors have done [8]—and the second would be to alter some step of the SDG’s construction to introduce those dependencies.

The most popular approach follows the latter option (modifying the SDG’s construction), and was proposed by Ball et al. [4]. It classifies statements into three separate categories:

Statement. Any statement that is not a conditional or unconditional jump. In the CFG, their nodes have one outgoing edge pointing to the next statement that follows them in the program.

Predicate. Any conditional jump statement, such as `while`, `until`, `do-while`, `if`, etc. In the CFG, nodes representing predicates have two outgoing edges, labelled *true* and *false*, leading to the statements that would be executed with each result of the condition evaluation. As mentioned before, in general no evaluation is performed on the conditions, so every conditional statement has two outgoing edges, even if the condition is trivially *true* or *false* (e.g., $1 = 1$ or *false*).

Pseudo-predicates. Unconditional jumps (i.e. `break`, `goto`, `continue`, `return`); are treated like predicates, with the difference that the outgoing edge labelled *false* is marked as non-executable—because there is no possible execution where such edge would be possible, according to the definition of the CFG (see Definition 6). For unconditional jumps, the *true* statement leads to the statement that will be executed after the jump is performed, and the *false* edge to the statement that *would* be executed if the jump was skipped or turned into a no-operation.

In future sections, other statements will make use of the pseudo-predicate structure (two outgoing edges, one non-executable), but using a different definition to place the non-executable edge. Therefore, the behaviour described for unconditional jumps is not universal for all statements classified as pseudo-statements.

As a consequence of this classification, every statement after an unconditional jump j is control-dependent on it, as can be seen in the following example.

Example 8 (Control dependencies generated by unconditional jumps)**.** Consider the program on the left side of Figure 3.5, which contains a loop and a `break` statement. The figure also includes the CFG and PDG for the method, showcasing the data and control dependencies of the statements. The slicing criterion $\langle 6, a\rangle$ is control dependent on both the unconditional jump and its surrounding conditional statement. Therefore, the slice (all nodes coloured in grey) includes both. They are necessary to terminate the loop, but they could be excluded in the context of weak slicing: the loop does not need to terminate, the slice can keep producing values.

3.4 Exceptions

Exception handling was first tackled in the context of Java program slicing by Sinha et al. [26], with later contributions by Allen and Horwitz [3]. There exist contributions for other programming languages, which will be explored later in chapter 5. This section explains 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. The `try-catch` statement can be likened to a `switch` which compares types (using the `instanceof` operator) instead of constants. Both structures require special handling to place the proper dependencies, so that slices are complete and as correct as possible.

3.4.1 `throw` statement

The `throw` statement compounds two elements in one statement: an unconditional jump with a value attached and a switch to an "exception mode", in which the statement's execution order

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

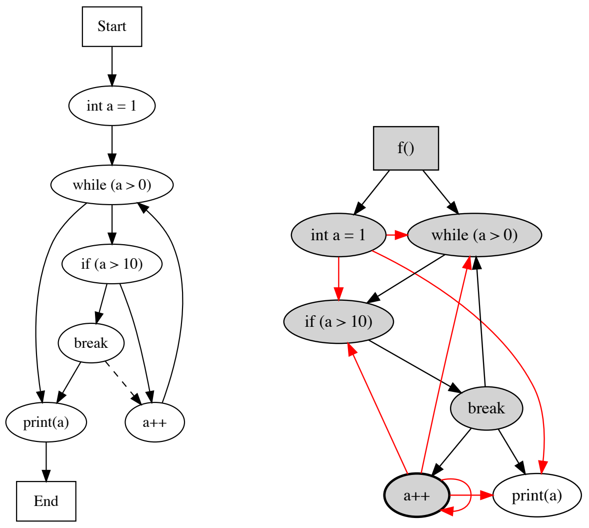


Figure 3.5: A program with unconditional control flow, its CFG (center) and PDG(right).

is disregarded. The first one has been extensively covered and solved; as it is equivalent to the `return` statement, but the second one requires a small addition to the CFG: there must be an alternative control flow for the error to flow throw until it is caught or the program terminates.

So far, without including `try-catch` structures, any exception thrown will activate the aforementioned "exception mode" and leave its method with an error state. Hence, in order to model this behaviour, a different exit point (represented with a node labelled "Error exit") needs to be defined. Consequently, the pre-existing "Exit" node is renamed to "Normal exit". Now we face the problem that CFGs may have two distinct sink nodes, something which is forbidden in most slicing algorithms. To solve that problem, a general "Exit" node is created, with both "Normal exit" and "Error exit" connected to it, which makes it the new sink of the CFG.

In order to properly accommodate a method's output variables (global variables or parameters passed by reference that have been modified), variable unpacking must be moved from "Exit" to both "Normal exit" and "Error exit". This duplicates some nodes, but allows some of those duplicated to be removed. Therefore, this change constitutes an increase in precision, as now the outputted variables are differentiated. For example, a slice which only requires the "Error exit" may include less variable modifications than one which includes both.

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

Example 9 (CFG of an uncaught `throw` statement)**.** Consider the simple Java method on the left of Figure 3.6; which performs a square root on a global variable x if the number is positive, otherwise throwing a `RuntimeError`. The CFG in the centre illustrates the treatment of `throw` as a pseudo-statement and the new nodes "Normal exit" and "Error exit". The PDG on the right describes the control dependencies generated from the `throw` statement to the following statements and exit nodes.

```
void f() {
  if (x < 0)
    throw new RuntimeException()
  x = Math.sqrt(x)
}
```

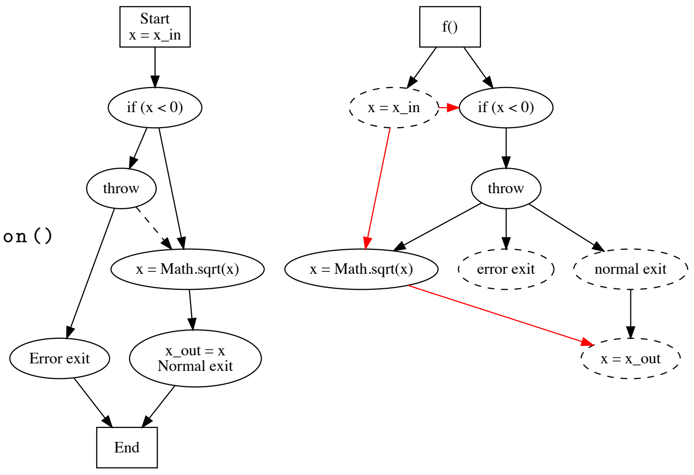


Figure 3.6: A simple program with a **throw** statement (left), its CFG (centre) and its PDG (right).

3.4.2 `try-catch-finally` statement

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

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 control dependencies that 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 should be executable.

Allen et al.'s representation of the `try` block is as a pseudo-predicate, connected to the first statement inside it and to the statement that follows the `try` block. This generates control dependencies from the `try` node to each of the statements it contains. Inside the `try` there can be four distinct sources of exceptions:

`throw` statements. The least common, but most simple to treat, because the exception is always thrown. The only problem may come from the ambiguity of the exception's type. For example, in the statement `throw ((Throwable) o)`, where `o` is a variable of type Object, the real type of the exception is unknown.

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. For this reason, they are not considered in most of the previous works. This does not mean that they require special treatment in the graph, they just need to be identified in all instructions that may generated them.

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. In case *unchecked* exceptions are also considered, method calls could be analysed to know which exceptions may be thrown, or the documentation could be checked automatically for the comment annotation `@throws` to know which ones can be raised. This is the most common way an exception appears inside a `try-catch` statement.

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 statement at any moment may throw an Error). Therefore, most slicing software ignores them. Similarly to implicit unchecked exceptions, they do not need special treatment, but their identification is costly and can complicate the SDG until every instruction is dependent on the correct execution of the previous one; which is true in a technical sense but not in most practical applications of program slicing.

All exception sources (except `throw` statements) are treated very similarly: the statement that may throw an exception has an outgoing edge the next statement. Then, there is an outgoing edge to each `catch` statement whose type may be compatible with the exception raised. The nodes that represent `try` and `catch` statements are both pseudo-predicates: the *true* edge leads to the first statement inside them, and the *false* edge leads to the first instruction after the `try-catch` statement.

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 that points to an auxiliary node labelled "normal return", in which the output variables produced by any normal exit of the method are placed. Each catch must then be labelled with the output variables produced by the erroneous exits of the method.

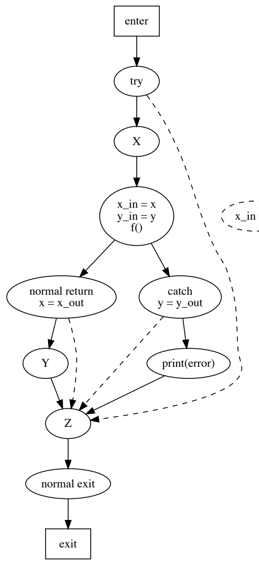
The "normal return" node is itself a pseudo-statement. The *true* edge is connected to the following statement, and the *false* one to the first common statement between all the paths of non-zero length start from the method call. The most common destinations for the *false* edge are (1) the first statement after the `try-catch` (if all exceptions that could be thrown are caught) and (2) the "Error exit" of the method (if some exception is not caught).

Example 10 (Code that throws and catches exceptions.)**.** Consider the segment of Java code in Figure 3.7 (left), which includes some statements 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 a message when it occurs. Additionally, consider the case that when f exits normally, only x is modified; but when an error occurs, only y is modified.

As can be seen in the CFG shown in Figure 3.7 (centre), the nodes “Normal return”, “catch” and “try” are considered as pseudo-statements, and their *true* and *false* edges (solid and dashed respectively) are used to create control dependencies. The statements contained after the function call, inside the `catch` statement and inside the `try` statement are respectively controlled by the aforementioned nodes.

Finally, consider the statement `Z`; which is not dependent on any part of the `try-catch` statement, as all exceptions that may be thrown are caught: it will execute regardless of the path taken inside the `try` block.

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



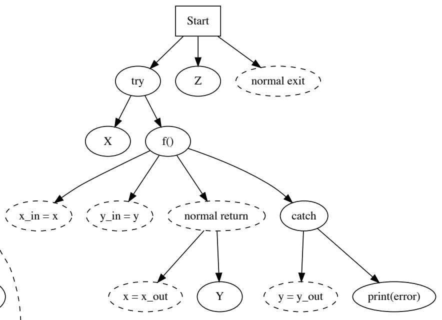


Figure 3.7: A simple program with a method call that could throw an exception (left), its CFG (centre) and its PDG (left).

Chapter 4

Improving the SDG for exception handling

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.

Regarding 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: (1) those that lower the completeness and (2) those that lower the correctness.

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

The second kind is less critical, but still 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.

4.1 Unconditional jump handling

The standard treatment of unconditional jumps as pseudo-statements introduces two separate correctness errors (type 2): *the subsumption correctness error*, which is relevant in the context of both strong and weak slicing, and the *structure-exiting jump*, that is only relevant in the context of weak slicing.

4.1.1 Problem 1: Subsumption correctness error

This problem has been known since the seminal publication on slicing unconditional jumps [4]: the paper's chapter 4 (page 219) 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. An analogous example—with `break` statements instead of `goto`—is shown in Example 11.

Example 11 (An unconditional jump subsumption [4])**.** 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 conditional expressions, uppercase letters are used; and no data dependencies are considered, as they are not relevant to the problem at hand.

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

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

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

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

Now consider statement `C` (line 11) as the slicing criterion. Figure 4.2 displays the SDG produced for the program, and the nodes selected by the slice. Figure 4.1 displays the computed slice on the centre, and one of the minimal slices on the left. The inner `break` on line 6 and the `if` surrounding it have been unnecessarily included. Their inclusion would not be specially problematic, if it were not for the condition of the `if` statement, which may include extra data dependencies that are unnecessary in the slice and that may lead to include other unnecessary statements, making the slice even more imprecise.

Line 6 is not useful because regardless of whether it executes, the execution will continue on line 13 (after the `while`), as guaranteed by the other `break` statement on line 9, which is not guarded by any condition. Note that `B` is still control-dependent on line , as it has a direct effect on it, but the dependence between both `break` statements introduces useless statements into the slice.

The problem showcased in Example 11 can be generalized as Problem 1 for any pair of unconditional jump statements that are nested and whose destination is the same.

Problem 1 (Subsumption correctness error)**.** Let a and b be two distinct unconditional jump statements without data whose destination c is the same. Any control edge that connects them is superfluous and includes unnecessary statements in the slices produced.

A solution for the subsumption correctness error

As only the minimum amount of control edges are inserted into the PDG (according to Definition 10), it is only necessary to remove the edge described in Problem 1 in order to improve the correctness of the algorithm. This removal must be performed after the SDG has been build, in order to avoid the reappearance of transitive dependencies that are excluded by the PDG's definition.

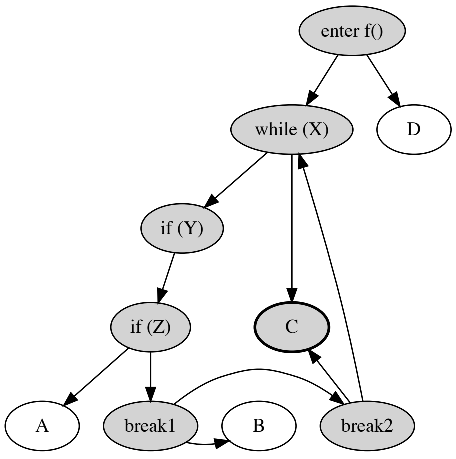


Figure 4.2: The system dependence graph for the program of Figure 4.1, with the slice marked in grey, and the slicing criterion in bold.

4.1.2 Problem 2: Unnecessary instructions in weak slicing

In the context of weak slicing, as 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 results in an infinity loop execution, or an uncaught exception behaviour. The following example describes a specific scenario which is generalized later in this section.

Example 12 (Unnecessary unconditional jumps)**.** Consider the code for method `g` on Figure 4.3, which features a simple loop with a `break` statement within. The slice in the middle has been created with respect to the slicing criterion $\langle 6, x\rangle$, 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 one considers 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 an infinite list of natural numbers (0, 1, 2, 3, 4, 5...). As the original program's output (the numbers 0 to 9) is a prefix of the natural numbers, the program is still a valid slice (pictured on Figure 4.3's right side). The sequences of values fulfil the requirements of Definition 5.

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

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.

Problem 2 (Unnecessary instructions in weak slicing)**.** Let j be an unconditional jump to X. j is not necessary in a slice S if there is no statement present in S that may be executed after

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

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

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

Figure 4.3: A simple loop with a break statement (left), its computed slice (middle) with respect to $\langle 6, x\rangle$, and the smallest weak slice (right) for the same slicing criterion.

X in the original program.

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.

A solution for the unnecessary instructions in weak slicing

After the slice has been completed, the unconditional jumps are identified. Those jumps after whose destination there are no more instructions are removed, and the slice recomputed. This is repeated until there are no more unconditional jumps that fulfil the condition expressed in Problem 2.

The complexity of this solution is higher than the previous one, as it raises the traversal of the slice from a linear time with respect to the number of nodes to $\mathcal{O}(nm)$, where n is the number of nodes and m is the number of unconditional jumps. That is a worst case estimation, as most cases will be close to linear time.

4.2 The `try-catch` statement

In this section we present an example where the current approach used to handle `try-catch` statements fails to capture all the correct dependencies, excluding from the slice some statements that are necessary for a complete slice (both weak and strong). After that, we generalize the set of cases where the lack of completeness (kind 1) is a problem and its possible appearances in real-life development. Finally, we propose a solution that properly represents all the dependencies introduced by the `try-catch`, focusing on producing complete strong slices.

4.2.1 Problem 3: The lack control dependencies of `catch` statements

In the current approach for exception handling [3], `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 the slicing criterion is inside the catch block, or if the value of a variable defined inside the catch block is needed (reaching it by data dependence).

The only occasion in which `catch` blocks generate any kind of control dependence 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 control dependent on every `catch` statement.

But, compared to the treatment of unconditional jumps, the lack of `catch` statements is not treated: unconditional jumps have a non-executable edge to the instruction that would be executed in their absence; `catch` statements do not.

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

```
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 the code of Figure 4.4is depicted in Figure 4.5[1]. 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.

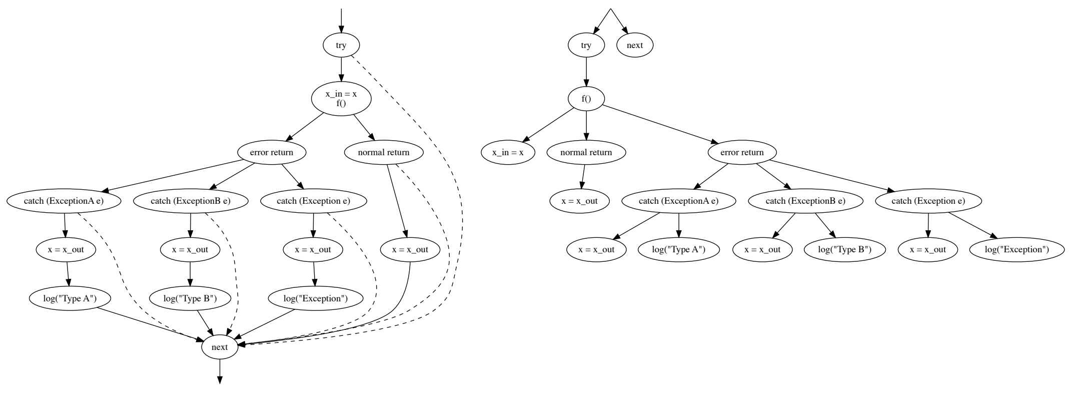


Figure 4.5: CFG (left) and PDG (right) of the code shown in Figure 4.4.

[1]For the sake of clarity, in the PDG of Figure 4.5, `log` function calls have been represented as a single node instead of their full node structures.

Example 14 (Incorrectly ignored `catch` statements)**.** Consider the code in Figure 4.6, in which the method `f` is called twice: once inside a `try-catch` statement, and a second time, outside of it. As it happened in Example 13, `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.

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

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

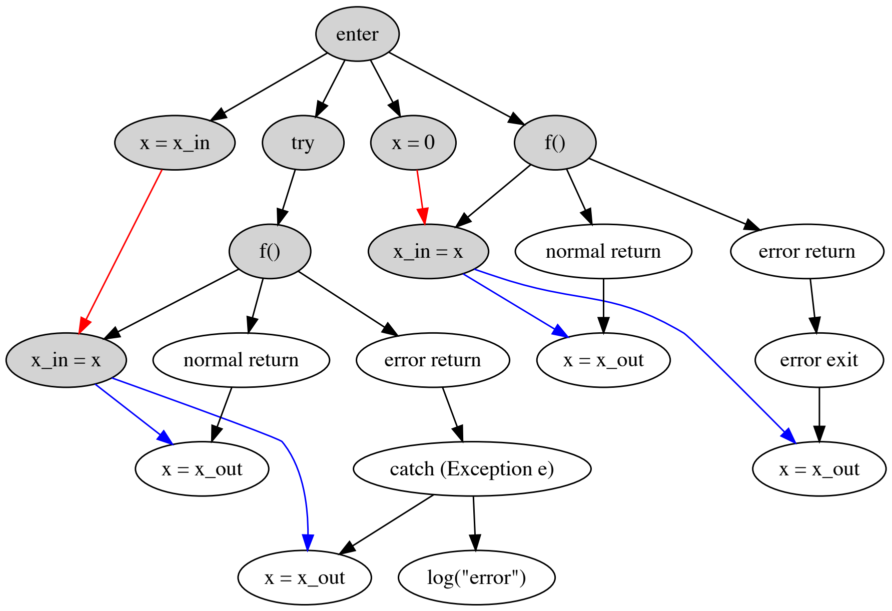


Figure 4.7: The SDG of the left snippet of Figure 4.6. `f` and the associated inter-procedural edges are not shown for simplicity.

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

In order to solve the drawback exposed above, we propose the `catch` statements to be handled as unconditional jumps: a non-executable edge should connect them to the instruction that would run if they were absent. There are two possibilities: 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 creates 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.

We propose two separate solutions in order to reduce the number of statements introduced:

1. Make the inclusion of `catch` statements conditional on not one but two dependencies: a statement that throws an exception is present in the slice but also there is a statement that needs the exception to be caught. This would place the minimum amount of `catch` statements, with the cost of a slower program slicer.

2. Represent each `catch` statement in multiple nodes, one per method that may lead to it. This would minimize the number of method calls that are included when the corresponding `catch` block is included, but it may increase considerably the amount of nodes in the SDG.

Both solutions need to be studied further before being implemented, but at least the slices produced are complete, even if some correctness is lost along the way.

Chapter 5

Related work

Program slicing was proposed [29] and iteratively improved until the proposal of the currently most used program representation structure, the SDG. Specifically, in the context of exceptions, multiple approaches have been attempted with varying degrees of success. In the realm of academia, there exists no definite solution. One of the most relevant initial proposals was Allen and Horwitz ([3]), although it was not the first one targeting the Java programming language specifically ([26, 27]).

In [3], Allen et al. benefit from the existing proposals for *return*, *goto* and other unconditional jumps [12] to model the behaviour of *throw* statements. Control flow inside *try-catch-finally* statements was simulated, both for explicit *throw* and all possible *throws* nested inside a method call. In that work, unchecked exceptions were considered but regarded as "worthless" to include, due to the increase in size of the slices, which reduces their effectiveness. The reason for this decision, was the number of unchecked exceptions embedded in normal Java instructions, such as `NullException` in any instance field or method, `IndexOutOfBoundsException` in array accesses and countless others, which would entail an exhaustive analysis of the code looking for every potential instruction that may arise all kinds of unchecked exceptions. On top of that, handling *unchecked* exceptions opens the problem of calling an API to which there is no analysable source code, either because the module was compiled before-hand or because it is part of a distributed system.

Chang et al. [17] 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. [15] describe 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 centred 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 call nodes. Prabhu et al. [24] have worked specifically on the C++ exception framework, but without producing any notable improvement to the field that could be applicable to Java.

Finally, Jie et al. [16] introduced an 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 [15], such as cascading *catch* statements, while adding explicit support for virtual calls, polymorphism and inheritance. Despite its reach, it does not solve the original underlying problems displayed in Allen's approach [3], which is why our thesis is centred around Allen's contribution.

Chapter 6

Conclusions

Program slicing is a powerful technique to extract subsets of statements from a program, which behave as the original with respect to a slicing criterion. In the past four decades, different techniques have been proposed and matured, among which the system dependence graph is the most popular. The SDG has been implemented for various programming languages and paradigms, but it does not have a definitive complete and correct solution yet.

Specifically, in the field of exception handling, there have not been significant advances since the beginning of the millennium [3], as later works have made minimal progress without finding any errors in Allen's proposal.

In this thesis, we show that the current treatment of exception handling constructs, such as `try-catch-finally` and `throw` is correct and complete only for some cases. We identify three distinct problems where the slices generated had lost correctness and completeness. We provide counter-examples to back up the problems, and generalize them to show the conditions necessary for them to surface.

An important contribution of our work has been the solutions proposed for each of the problems identified. Each exchanges a small amount of performance for an improvement in correctness or completeness. The solutions have been proposed specifically for Java's exception handling system, but are valid for almost any other programming language with a similar exception system (which can be seen in detail on section 2.2.2).

Future work

Our work does not end here, we are currently studying some improvements and applications related to our proposal, hereunder we enumerate some of them:

- Implementation of the solutions proposed, so that they can be benchmarked against the previous state of the art, and used to build better program slicers. The implementation could be done in Java or another language with a similar exception-catch system. The solutions that improve correctness at the price of slice speed could be implemented as optional for the user to execute, as to avoid increasing the slicing software's temporal complexity.

- Improved correctness for the `try-catch` statement. The solution proposed in chapter 4 does solve the lack of completeness in the treatment of `catch` statements, at the cost of including many more `catch` statements that are really necessary. This is not the most desirable outcome, but it can be improved by developing and implementing the measures suggested at the end of the solution: make the inclusion of the `catch` statements conditional upon two dependencies instead of one and represent `catch` statements multiple times, so that the method calls that throw errors may be selectively included.

- Redefinition or specialization of control dependence: the system dependence graph is centred around the definitions of control and data dependence. The meaning of control dependence has slowly shifted as more kinds of statements have been included in program slicing, but its definition has remained almost constant for three decades. Unconditional jumps and `catch` statements introduce a new kind of control dependence which is not the traditional "b is control dependent on a if the execution of a affects whether or not b executes", but "the *presence* of a affects whether or not b executes". This constitutes a substantial change that has not been reflected, and is the source of many problems when handling both unconditional jumps and `catch` statements.

Bibliography

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

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

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

[4] 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.

[5] 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.

[6] David Binkley and Keith Brian Gallagher. Program Slicing. *Advances in Computers*, 43(2):1–50, April 1996.

[7] 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.

[8] 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.

[9] 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.

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

[11] Ákos Hajnal and István Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software Maintenance*, 24(1):67–82, 2012.

[12] 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.

[13] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[20] 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.

[21] 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.

[22] 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.

[23] Sergio Pérez, Josep Silva, and Salvador Tamarit. Automatic Testing of Program Slicers. *Scientific Programming*, vol. 2019, Article ID:1–15, February 2019.

[24] 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.

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

[26] 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.

[27] 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.

[28] Frank Tip. A survey of Program Slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[29] 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.