UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Diseño y desarrollo de un fragmentador de programas para Java basado en el System Dependence Graph

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Departamento de Sistemas Informáticos y Computación

Autor: Javier Costa Rosa
Tutor: Josep Francesc Silva Galiana
Valencia, curso 2019-2020

Resumen

La fragmentación de programas es una técnica de análisis de software muy útil en muchos ámbitos relacionados con la ingeniería del software, especialmente en la depuración de código. No obstante, a pesar de lo beneficiosa que puede llegar a ser, no todos los lenguajes de programación disponen de un fragmentador público que aborde la totalidad del lenguaje y que integre los últimos avances en esta técnica.

En el caso de Java sí existen herramientas de análisis estático comerciales como CodeSonar. Sin embargo, aunque también existen herramientas de libre acceso como Indus, WALA, Kaveri, etc., ninguna de estas está suficientemente actualizada, o incluso son dependientes de otras o de algunos entornos de desarrollo, como puede ser Eclipse.

Disponer de un fragmentador de código puede facilitar en gran medida tareas relacionadas con la ingeniería del *software*. En el caso de la depuración de código, por ejemplo, un error obtenido en un punto cualquiera del programa es mucho más sencillo de identificar si se consiguen descartar algunas partes del programa que se sabe a ciencia cierta que no tienen relevancia alguna en el error obtenido, optimizando tiempo y recursos.

Lo que se pretende en el presente trabajo es diseñar y desarrollar desde cero una herramienta de fragmentación de código estática que sea pública, actualizada y de libre acceso para el lenguaje Java. Esto incluye todo el ciclo de desarrollo: Desde el diseño conceptual, pasando por el análisis del código, la construcción de diversas representaciones necesarias como el CFG (*Control Flow Graph*), PDG (*Program Dependence Graph*) y SDG (*System Dependence Graph*), entre otros, hasta la obtención del fragmento y su exportación a fichero. Todo ello tratando de desarrollar una herramienta robusta, mantenible y extensible, de forma que se pueda actualizar y ampliar su funcionalidad de forma fácil y sencilla en el futuro.

Abstract

Program slicing is a program analysis technique that is useful in many areas related to software engineering, especially in program debugging. However, despite its benefits, not all programming languages have a public and up-to-date program slicer.

In case of Java, there do exist commercial, static analysis tools like CodeSonar. However, although some freely available tools also exist, like Indus, WALA, Kaveri, etc., none of them is up-to-date, or even they depend on others or some development environments like Eclipse.

Having a program slicer can simplify some tasks related to software engineering. In case of code debugging, for example, an error obtained in any point of the program can be traced more easily if some certainly irrelevant parts of the program can be discarded, optimizing time and resources.

The aim of this work is to design and develop a public, up-to-date and freely accesible static program slicing tool from scratch, for the Java language. That includes the whole development cycle: From the conceptual design, through code analysis, graph building such as CFG (Control Flow Graph), PDG (Program Dependence Graph) and SDG (System Dependence Graph), among others, to the computation of the final fragment and its exportation. All of this trying to develop a robust, maintenable and extensible tool, that can be updated and extended in the future.

Resum

La fragmentació de programes és una tècnica d'anàlisi de software molt útil en molts àmbits relacionats amb l'enginyeria software, especialment en la depuració de codi. No obstant, tot i lo beneficiosa que pot arribar a ser, no tots els llenguatges de programació disponen d'un fragmentador públic que aborde la totalitat del llenguatge i que integre els últims avanços d'aquesta tècnica.

En el cas de Java, sí existeixen ferramentes d'anàlisi estàtic comercials com CodeSurfer. No obstant, encara que existeixen algunes ferramentas de lliure accés com Indus, WALA, Kaveri, etc. ningú d'aquestes està completament actualitzada, o dependeixen d'altres ferramentas o d'algú entorn de desenvolupament com Eclipse.

Disposar d'un fragmentador de codi pot facilitar molt algunes tasques relacionades amb la engenyeria del *software*. En el cas de la depuració de codi, per example, un error obtingut en qualsevol punt del programa és molt més senzill d'identificar si es consegueix descartar altres parts del programa que es sap que no tenen rellevància, optimitzant el temps i els recursos.

En aquest treball es pretén disenyar i desenvolupar desde zero una ferramenta de fragmentació de codi estàtica que siga pública, actualitzada i de lliure accés per a Java. Aquest inclou tot el cicle de desenvolupament: desde el diseny conceptual, pasant per l'anàlisi del codi, la construcció de diverses representacions necesarias com el CFG (*Control Flow Graph*), PDG (*Program Dependence Graph*) y SDG (*System Dependence Graph*), entre altres, fins i la obtenció del fragment i la seua exportació. Tot això tratant de desenvolupar una ferramenta robusta, mantenible y extensible que es puga actualitzar i ampliar la seua funcionalitat de manera fàcil y senzilla en el futur.

Palabras clave— Program Slicing, Java, System Dependence Graph

Índice general

Capítulo 1

Introducción

1.1. Visión general

El concepto de fragmentación de programas (en inglés, *Program slicing*) fue introducido originalmente por Mark Weiser en [1] y [2], donde define un fragmento (o *slice*, en inglés) como un programa ejecutable que se compone de partes extraídas de un programa y que tienen el mismo comportamiento (producen los mismos valores) que el programa original en un punto dado. Este punto se conoce como criterio de fragmentación (en inglés, *slicing criterion*), y más concretamente, hace referencia a un punto del código y a una o más variables del programa que será el punto de partida para realizar el cálculo. Esta técnica es muy utilizada en ámbitos como la depuración de código [9, 10, 11, 12], mantenimiento de código [14], detección de errores [33], especialización de programas [34], etc. El resultado del cálculo de un fragmento no es único. Esto significa que para el mismo criterio de fragmentación, pueden existir diferentes fragmentos válidos (i.e., que tienen el mismo comportamiento que el programa original). Sin embargo, siempre es preferible que el fragmento contenga el menor número de instrucciones posible; aunque, generalmente, estos suelen ser más difíciles de calcular. El fragmento ideal, por tanto, sería el que contenga el menor número de instrucciones, lo que se conoce como fragmento mínimo, aunque, como el propio Weiser [2] demostró, calcularlo no es siempre posible, pues se trata de un problema indecidible.

La definición original de la fragmentación de programas era estática [2]. Se define como estática porque no tiene en cuenta ninguna ejecución del programa en particular, sino que intenta cubrir todas las posibles ejecuciones. Existen otras variaciones que han surgido después, como la fragmentación dinámica (*dynamic slicing*), que permite calcular un fragmento considerando una ejecución concreta del programa (i.e., un conjunto de valores de entrada), lo que produce un fragmento resultante más preciso para el conjunto de valores de entrada especificado.

```
1  public void foo() {
2      int x = 1;
3      int y = 5;
4      if (x < 0) {
5          x = x + 1
6      }
7      if (y > 0) {
8          y = y + 1;
9      }
10     System.out.print(x);
11     System.out.print(y);
12 }
```

```
1  public void foo() {
2
3      int y = 5;
4
5
6
7      if (y > 0) {
8          y = y + 1;
9      }
10
11     System.out.print(y);
12 }
```

Figura 1.1: Ejemplo de un fragmento (derecha) del programa original (izquierda) con respecto al criterio de fragmentación ⟨11, y⟩

La figura 1.1 muestra el resultado del cálculo de un fragmento de un método de Java con respecto al criterio de fragmentación ⟨11, y⟩. Todas las instrucciones que no tienen ninguna relación con la y en la línea 11 son eliminadas en el fragmento (ya que nuestro criterio de fragmentación hace referencia a la y), mientras que el comportamiento de ambos programas sigue siendo exactamente el mismo (al ejecutar la línea 11, ambos programas mostrarán el mismo valor para y).

1.2. Motivación

La fragmentación de programas puede ser una técnica muy beneficiosa en muchas áreas de la producción del *software*. Una de ellas es la depuración de programas, donde este tipo de técnicas ayudan en gran medida al programador. Por ejemplo, si un error es encontrado en la salida de algún módulo del programa, es mucho más fácil inspeccionar las instrucciones que estén relacionadas con él, en lugar de hacerlo con otras partes que no tienen relación alguna y que, por tanto, representan una pérdida de tiempo. El fragmentador ideal sería aquel que es capaz de obtener únicamente las instrucciones relevantes para encontrar el origen del error, lo que ayuda en gran medida al programador, teniendo menos instrucciones que inspeccionar. Dado que encontrar el fragmento mínimo es un problema indecidible, incluso para el mejor de los fragmentadores de programas es posible que algunas instrucciones irrelevantes se incluyan en el fragmento final. Aún así, inspeccionar este fragmento resulta un trabajo mucho más fácil que inspeccionar el programa completo.

Aunque se usa principalmente en la depuración de código, la fragmentación de programas también se usa en ámbitos tan variados como son la paralelización de programas [13], la diferenciación de programas [15, 16], el mantenimiento del software [14], el testeo [17, 18] y la ingeniería inversa [19, 20].

El proceso de obtención de un fragmento no es trivial. Se ha de analizar e interpretar el código del programa, lo que implica tratar con la idiosincrasia de cada lenguaje, como su sintaxis, sus propias estructuras de datos, etc. para realizar una abstracción que permita trabajar sin tener que preocuparse de todo esto. Esto hace que no pueda existir un fragmentador universal o, dicho de otra forma, válido para cualquier lenguaje.

A pesar de todos los beneficios que profiere esta técnica, la mayoría de lenguajes de programación no disponen de un fragmentador completo, actualizado y público que cubra todas las características del lenguaje. Existen algunas herramientas públicas desactualizadas, aunque sí que existen herramientas comerciales que hacen uso de esta técnica, como CodeSonar[35].

1.3. Objetivos

El objetivo principal de este trabajo es la implementación de un fragmentador que genere fragmentos completos, que sea público y de código abierto. Como caso de estudio, se escogerá Java como lenguaje para el cual desarrollar el fragmentador, cubriendo un subconjunto de características compatibles hasta el JDK11. Además, el fragmentador debe tener tanto una estructura como unas APIs bien definidas, de modo que sea fácil extender su funcionalidad en el futuro para cubrir más características de Java o incluso dar soporte a otros lenguajes.

Capítulo 2

Contexto

Antes de empezar a describir el diseño y los detalles de implementación del fragmentador, es necesario introducir un poco de contexto, ahondando un poco más en la fragmentación de programas, definiendo unos conceptos previos, y revisando otros trabajos relacionados.

2.1. Fragmentación de programas

La fragmentación de programas fue definida en sus orígenes como estática y "hacia atrás" (*backward static slicing*). Estática, porque el fragmento cubre todas las posibles ejecuciones del programa. Hacia atrás, porque desde el criterio de fragmentación la pregunta que se realiza es: ¿Qué instrucciones afectan a estas variables en esta línea de código? Por tanto, se buscan todas las posibles instrucciones que puedan afectar a ese punto del programa.

Después de que Weiser definiera su concepción de la fragmentación de programas, otras variaciones empezaron a surgir. En 1985, Bergeretti y Carré [3] introdujeron la fragmentación "hacia delante" (*forward slicing*), que consiste en obtener lo contrario a la fragmentación hacia atrás. Es decir, en lugar de obtener las instrucciones que puedan afectar a las variables en un determinado punto, trata de obtener aquellas instrucciones que pueden verse afectadas por dicho punto. Esta variación resulta útil para algunas tareas de transformación de código, como la eliminación de código muerto (i.e., eliminar código del programa que no tiene ninguna utilidad) o como el mantenimiento de software. En particular, permite saber qué partes del código se verán afectadas por un cambio en un punto dado.

En 1988, Korel y Laski [4] definieron la fragmentación dinámica, que se diferencia de la fragmentación estática en que, en lugar de generalizar para cualquier posible ejecución del programa, se obtiene un fragmento específico para una ejecución concreta. Por supuesto, esto requiere incluir la información de los valores de entrada para esa ejecución en el criterio de fragmentación. Esta técnica permite obtener un fragmento del programa más preciso, pero, como desventaja, solo es válido para unos valores de entrada concretos. Si estos cambian, el fragmento deja de ser válido.

A modo de resumen, finalmente se puede decir que realmente existen dos dimensiones principales[1] de aplicación de la fragmentación de programas:

- Estática o dinámica, si intenta cubrir todas las posibles ejecuciones de un programa o, en cambio, solo una única ejecución (con unos valores de entrada concretos).
- Hacia atrás o hacia delante, si busca las instrucciones previas que puedan afectar a un determinado punto del programa o, en cambio, busca las instrucciones posteriores que pueden verse afectadas por dicho punto.

Este trabajo está centrado en la fragmentación estática hacia atrás. Esto quiere decir que el fragmentador final producirá fragmentos que cubrirán todas las posibles ejecuciones del programa que analice, y se compondrá de un conjunto de partes del programa que afecten al punto definido por el criterio de fragmentación.

Mediante el estudio de ciertos análisis que se pueden realizar a un programa, como son el análisis del flujo de control y de las dependencias tanto de control como de datos, se pueden realizar interpretaciones que representen el programa en cuestión de forma abstracta.

La técnica de fragmentación de programas, en sus inicios, estaba pensada para lenguajes procedurales. No obstante, se pueden aplicar los mismos principios, en este caso, a métodos de Java. Existen dos tipos de alcance a la hora de obtener un fragmento. El primero es un alcance intraprocedural (i.e., dentro de un mismo procedimiento/método), y el segundo es un alcance interprocedural (i.e., dentro del mismo procedimiento/método y, además, entre procedimientos/métodos).

2.1.1. Fragmentación de programas intraprocedural

La fragmentación intraprocedural hace referencia al tipo de fragmentación que se realiza dentro de un único procedimiento (o, en este caso, método). Para ello, es necesario construir el *Program Dependence Graph* (PDG) [21], el cual es un grafo dirigido cuyos nodos son sentencias o expresiones (predicados u operadores) y que dispone de dos tipos de arcos: Por un lado, un tipo de arco representa los valores de datos de los que dependen las operaciones de un nodo (dependencias de datos) y, por otro, las condiciones de control de las que depende la ejecución de las operaciones del nodo (dependencias de control).

Para poder construir el PDG, en primer lugar es necesario construir una representación del flujo de control del programa. Esto es posible hacerlo mediante el *Control Flow Graph* (CFG), que se trata de un grafo dirigido donde cada nodo representa una sentencia del programa, y el orden de los nodos indica el orden en el que las sentencias son ejecutadas.

Definición 1 (Control Flow Graph [21])**.** Un *Control Flow Graph* es un grafo dirigido G en el que se cumple que:

- Existe un nodo único de entrada START y un nodo único de salida STOP

[1] En [23] se puede encontrar una discusión sobre las dimensiones que afectan a esta técnica.

- Cada nodo en el grafo tiene, como máximo, dos sucesores
- Para cada nodo N en G, existe un camino de START a N y de N a STOP

Una vez obtenido el CFG, y a partir de este, se pueden obtener las dependencias de control y de datos. Las dependencias de control están definidas en base al concepto de postdominancia entre nodos.

Definición 2 (Postdominancia [21])**.** Dado un CFG G, un nodo V es postdominado por un nodo W en G si cada camino dirigido de V a STOP (sin incluir V) contiene W.

Definición 3 (Dependencia de control [21])**.** Dado un CFG G y dos nodos X e Y de G, Y es dependiente por control de X si y solo si:

1. existe un camino dirigido P de X a Y con cualquier Z en P (excluyendo X e Y) postdominado por Y y
2. X no está postdominado por Y.

Definición 4 (Dependencia de datos [6])**.** Dado un CFG G y dos nodos X e Y de G, Y es dependiente por datos de X si y solo si:

1. X es un nodo que define una variable v
2. Y es un nodo que usa v
3. Existe un camino en G de X a Y en el que no se define v

Una vez finalizada la construcción del PDG, la obtención del fragmento se realiza, en primer lugar, localizando el nodo correspondiente al criterio de fragmentación. Una vez localizado el nodo, obtener el fragmento es trivial, pues consiste únicamente en recorrer todos los arcos desde el nodo hacia atrás, recursivamente para cada nodo visitado. El conjunto de los nodos visitados representan el fragmento final.

```
1  public static void foo() {
2      int x = 1;
3      int y = 10;
4
5      int z = 0;
6
7      while (x > y) {
8          z = z + x;
9      }
10
11     System.out.println(z);
12 }
```

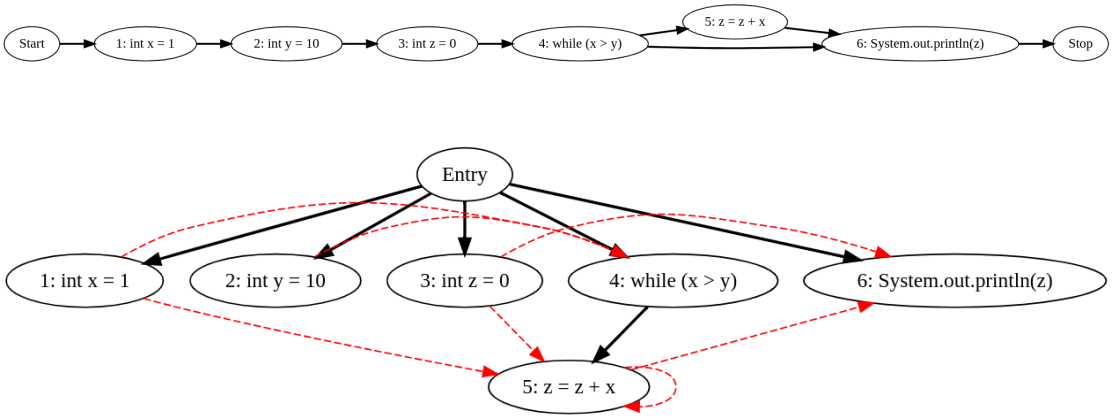

Figura 2.1: Ejemplo de método en Java (arriba), con sus correspondientes CFG (centro) y PDG (abajo)

En la figura 2.1, se muestra un método escrito en Java con sus correspondientes CFG y PDG construidos. En el CFG, el control de flujo es representado mediante arcos. En el PDG, las dependencias de control se representan mediante arcos negros, y las dependencias de datos mediante arcos intermitentes rojos.

No obstante, estas representaciones no son capaces de representar saltos incondicionales (como *break*, *continue* y *return*), puesto que éstos poseen una representación en el CFG que no da lugar a las dependencias de control correctas. Por tanto, para poder analizar e incluir este tipo de sentencias, es necesario construir un grafo aumentado que extiende el PDG llamado PPDG (*Pseudo-predicate Program Dependence Graph*), que incluye estas sentencias, y que se construye haciendo uso del ACFG (*Augmented Control Flow Graph*), una extensión del CFG creada para representar correctamente los saltos incondicionales.

El ACFG es un grafo que se construye de la misma forma que el CFG, únicamente con dos diferencias[5]: En primer lugar, se tratan las sentencias de salto como pseudo-predicados. Cada sentencia de salto está representada por un nodo con dos arcos de salida: Un arco de control de flujo, y otro arco no ejecutable. El arco no ejecutable representa un flujo inexistente en el programa, pero en conjunto mejoran la creación de las dependencias de control. De cara a las dependencias de datos, los arcos no ejecutables son ignorados. Por otra parte, las etiquetas son tratadas como sentencias separadas, de modo que cada etiqueta es representada en el ACFG como un nodo con un único arco de salida a la sentencia a la que se refiere.

El PPDG extiende el PDG con dos modificaciones: una nueva definición de las dependencias de control (véase la Definición 5) y una alteración al algoritmo de fragmentación.

Definición 5 (Dependencia de control en presencia de pseudo-predicados[5])**.** Un nodo N es dependiente por control de un nodo M si y solo si N postdomina, en el CFG, uno pero no todos los nodos sucesores de M en el ACFG.

Los pasos para crear el PPDG son los siguientes[5]:

1. Construir el ACFG, en base a la definición de ACFG descrita anteriormente
2. Construir el PPDG ignorando los arcos no ejecutables del ACFG al calcular las dependencias de datos, y calcular las dependencias de control según la definición 5

Una vez creado el PPDG, se puede realizar fragmentación sobre él de la misma forma que sobre el PDG, teniendo en cuenta la siguiente restricción: no se deben atravesar los arcos de control cuyo destino sea un pseudo-predicado p, salvo que p sea el criterio de fragmentación.

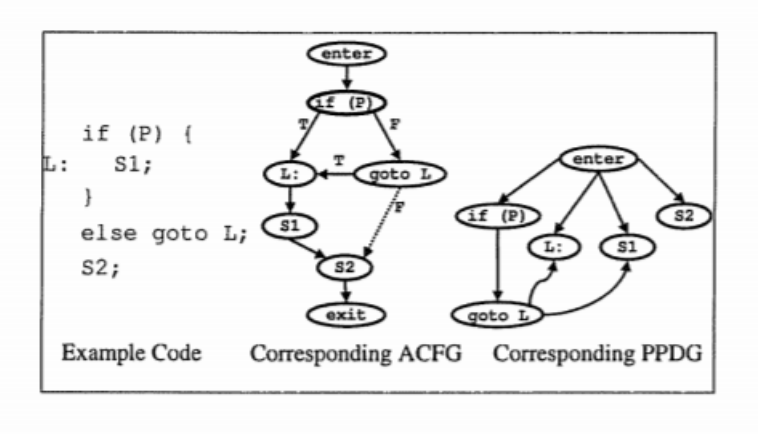

Figura 2.2: Ejemplo de un código con su respectivo ACFG (centro) y PPDG (derecha)

2.1.2. Fragmentación de programas interprocedural

La fragmentación interprocedural consiste en la fragmentación de un programa que se compone de un procedimiento inicial y uno o más procedimientos auxiliares. Este tipo de fragmentación se realiza mediante la construcción del *System Dependence Graph* (SDG). En concreto, el SDG, introducido por Horwitz et al. [6], es un grafo que extiende el PDG y que representa programas en un lenguaje que incluye procedimientos y llamadas a procedimientos, añadiendo dos tipos de arcos: En primer lugar, un tipo de arco que representa una dependencia entre una llamada a procedimiento y el propio procedimiento al que se llama. En segundo lugar, un tipo de arco que representa dependencias transitivas entre llamadas. Según su definición [6], fue ideado para modelar programas escritos en lenguajes que tengan las siguientes propiedades[2]:

[2]Estas propiedades son compatibles con un lenguaje como Java, asumiendo la equivalencia método = procedimiento, procedimientos que no devuelven valores = métodos *void* y paso por valor en lugar de valor-resultado

- El sistema completo está compuesto por un programa principal y un conjunto de procedimientos auxiliares.
- Los procedimientos terminan con sentencias *return*, que no incluye una lista de variables (no devuelven ningún valor).
- Los parámetros se pasan por valor-resultado.

Para representar las llamadas a procedimientos, se crea un nuevo tipo de arco desde el nodo de la llamada al nodo del procedimiento.

Para representar el paso de parámetros, se añaden unos nodos adicionales. En concreto, en el lado de la llamada se añade un par de nodos por cada argumento de la llamada. Estos nodos toman el nombre de *actual_in*, para los valores de los argumentos de entrada, y *actual_out*, para los valores de salida de cada argumento. Dichos nodos representan sentencias de asignación que copian el valor real de los argumentos a unas variables temporales y desde las variables temporales, respectivamente [6]. De forma similar, los parámetros del procedimiento se representan mediante dos nodos por cada parámetro. Estos nodos tienen el nombre de *formal_in* y *formal_out*, y representan sentencias de asignación donde se copia el valor de las variables temporales de llamada y a las variables temporales de llamada, respectivamente. Adicionalmente, se añaden nuevos arcos, *parameter_in* y *parameter_out*, que representan la correspondencia de un *actual_in* a un *formal_in*, y de un *formal_out* a un *actual_out*. Finalmente, también se añade un nuevo tipo de arco, llamado *arco de resumen*, el cual representa las dependencias transitivas de las llamadas a procedimientos, de forma que un arco de resumen existe desde un nodo *actual_in* a un *actual_out* si la variable correspondiente al argumento de entrada influye (transitivamente) en el argumento de salida.

La obtención del fragmento sería un proceso que conllevaría, en primer lugar e igual que en el PDG, localizar el nodo correspondiente al criterio de fragmentación. Acto seguido, el procedimiento es recorrer todos los arcos entrantes de dicho nodo hacia atrás de forma recursiva, exceptuando los arcos de parámetro de salida y para cada nodo visitado y, como paso adicional, realizar una segunda pasada recorriendo todos los arcos salientes del nodo recursivamente también para cada nodo visitado, exceptuando los arcos de llamada y de parámetro de entrada. De esta forma, todos los nodos que han sido recorridos en las dos pasadas forman parte del fragmento final.

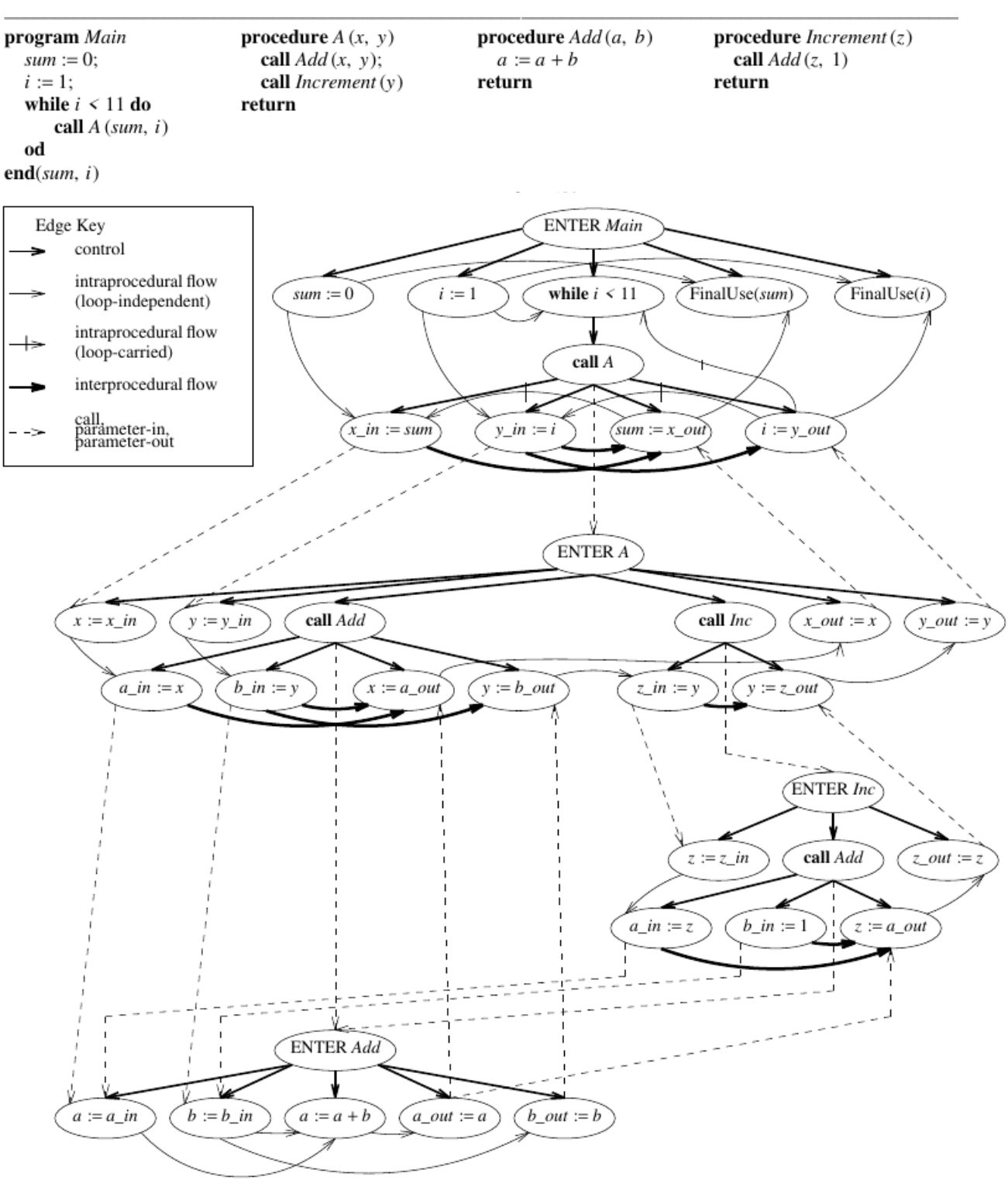

Figura 2.3: Ejemplo de un programa completo con su correspondiente SDG[6]

Dado que el SDG extiende del PDG, es necesario realizar algunos cambios para poder soportar los saltos incondicionales. En primer lugar, en lugar de usar el PDG para representar los métodos, construiremos el PPDG, que incluye la representación de dichos saltos incondicionales. Tal y como se ha descrito en la Sección 2.1.1, el algoritmo de fragmentación queda alterado al basar el SDG en el PPDG.

Por otro lado, cuando se calculan los arcos de resumen haciendo fragmentación intraprocedural, también se debe tener en cuenta la restricción introducida por el PPDG, y no se deben atravesar arcos de control que lleven a un pseudo-predicado, salvo que éste sea el criterio de fragmentación.

2.1.3. Proceso de un fragmentador

El proceso completo que debe realizar un fragmentador para obtener un fragmento puede dividirse en dos partes distintas:

1. **Análisis del código del programa y construcción de los grafos de dependencias**. Esta parte es específica para cada lenguaje, dado que es necesario conocer la idiosincrasia propia del lenguaje para poder interpretar de forma correcta todas sus instrucciones y lo que representan, dando paso a poder realizar una abstracción con la que poder trabajar. Este paso es el que impide que exista un fragmentador válido para cualquier lenguaje, puesto que requiere grandes conocimientos concretos del mismo. Durante el análisis del código se construyen las representaciones necesarias (CFG, PDG, SDG), siendo la tarea que más recursos computacionales consume (construir el grafo tiene un coste temporal $O(n^2)$, siendo n el número de instrucciones del programa).
2. **Obtener el fragmento**. Finalmente, haciendo uso de los grafos ya construidos, se aplica el algoritmo específico para obtener el fragmento (obtener el fragmento a partir del grafo tiene un coste temporal $O(n)$).

2.2. Trabajos relacionados

Como se ha mencionado anteriormente, no existe una gran variedad de herramientas de fragmentación de programas que sean completos, estén actualizados y sean públicos. A continuación, se detallan algunas herramientas que incluyen fragmentación de programas estática:

Indus [8] es un proyecto público que incluye una colección de herramientas de análisis y transformación de código para Java, entre las que incluye un fragmentador de código estático el cual es capaz de obtener fragmentos completos y ejecutables. No obstante, no está actualizado, pues no soporta la fragmentación para programas Java compilados con un JDK posterior al JDK4.

Kaveri [24] es un fragmentador estático capaz de realizar fragmentación hacia atraś y hacia delante para Java que también permite realizar comprensión de código sin usar el SDG. Está basado en Indus y se trata de un *plugin* para el entorno de desarrollo *Eclipse*, lo cual obliga a usar este entorno para poder utilizarlo. No se actualiza desde 2014.

WALA [36] es un conjunto de herramientas que realiza análisis estático sobre código y bytecode de Java. Dispone de un fragmentador basado en SSA, pero no es capaz de obtener fragmentos ejecutables.

JOANA [25] es otro fragmentador estático de que permite realizar control de la información de flujo de *bytecode* de Java. Está actualizado hasta el JDK8 y es público, aunque se ejecuta usando el framework WALA y en el entorno de desarrollo Eclipse.

JavaSMT [28] es un fragmentador que permite realizar fragmentación hacia atrás y hacia delante de código Java sin usar el SDG. Es capaz de realizar análisis de programas no compilables o ejecutables, y obtener un fragmento válido, aunque no ejecutable. Su código no es público.

Por otra parte, CodeSonar[35] es una herramienta comercial de análisis estático de código para Java que hace uso de la fragmentación de programas en sus análisis.

Adicionalmente, se puede encontrar una serie de aproximaciones a la extensión del SDG para soportar la orientación a objetos. En este sentido, Larsen et al.[30], Zhao [32] y Liang et al. [31] fueron capaces de extender el SDG para abarcar algunos conceptos de la orientación a objetos y el algoritmo específico para su fragmentación. Más adelante, surgieron aproximaciones específicas para Java, como [7] y el JSysDG [26], los cuales extienden el SDG de forma que añade soporte a clases, clases abstractas, interfaces y objetos. Así lo hace también de forma mejorada [27]. No obstante, esto queda fuera del ámbito de este trabajo, pues no es el objetivo extender el SDG hacia la orientación a objetos.

En este trabajo, se pretende que el fragmentador desarrollado sea capaz de obtener fragmentos ejecutables y completos compatible hasta el JDK11 a partir del análisis estático del código del programa tomando como base la definición inicial del SDG [6].

Capítulo 3

El lenguaje Java y el SDG

En esta sección se comentarán los detalles de las características del SDG que pueden ser aplicadas al lenguaje Java. Recordemos que el SDG se ideó para lenguajes procedurales y con un conjunto de características concretas, las cuales se muestran de nuevo a continuación:

- El sistema completo esté compuesto por un programa principal y un conjunto de procedimientos auxiliares. *Compatible con Java*
- Los procedimientos terminan con sentencias *return*, que no incluye una lista de variables (no devuelven ningún valor). *Compatible con Java*
- Los parámetros se pasan por valor-resultado. *Compatible con Java, aunque el paso de parámetros no se realice de la misma forma*

Es posible utilizar el SDG para realizar fragmentación de programas Java, pero está limitado en ciertos aspectos. El objetivo de este trabajo es la aplicación del SDG para la fragmentación de un programa escrito en Java, por lo que es necesario analizar qué características concretas del lenguaje es capaz de cubrir el SDG, y realizar las adaptaciones que sean necesarias.

A partir de este punto, cuando se habla de SDG, se habla del SDG construido con PPDGs, y con las variaciones necesarias en el algoritmo de fragmentación para soportar los saltos incondicionales.

3.1. Orientación a objetos

El SDG se ideó para lenguajes procedurales, por lo que no concibe conceptos del paradigma orientado a objetos tales como clase, herencia, interfaz, etc. Debido a esto, y dado que el objetivo principal de este trabajo consiste en la aplicación del SDG, el fragmentador final no cubrirá funcionalidades relacionadas con orientación a objetos.

3.2. Paso por valor-resultado y paso por valor

El paso por valor-resultado permite que a una variable pasada al procedimiento se le reasigne un nuevo valor. De ahí la importancia de tener nodos *formal_out* y *actual_out*.

Java es un lenguaje cuyos parámetros son pasados por valor. Para variables no primitivas (objetos), se pasa el valor de la referencia. Esto quiere decir que si una variable es reasignada dentro de un método al que se le pasa como argumento, dicha reasignación no tiene efecto fuera de él. En otras palabras, el valor (o la referencia, para variables no primitivas) de una variable x, que se pasa como argumento a un método m en el que se reasigna su valor, no cambia después de la ejecución del método m en ningún caso.

Esto tiene dos implicaciones:

- Las variables primitivas no son afectadas por lo que pase dentro del método al que se le pasan como argumento de ninguna forma.
- Las variables no primitivas no son afectadas por reasignaciones dentro del método al que se le pasan como argumento, pero el objeto sí puede sufrir cambios, pues se pasa la referencia de este.

Dado que no se va a dar soporte a la orientación a objetos del lenguaje Java, las llamadas a métodos que contengan variables no primitivas (objetos) como argumentos, serán ignoradas. Por tanto, y dado que es imposible que el valor de las variables primitivas cambie dentro de un método, en nuestro SDG no sería necesario incluir nodos *formal_out* ni *actual_out* para parámetros y argumentos. No obstante, para garantizar un SDG coherente con su definición, sí se incluirán. Esto implica que, por definición de los arcos de resumen, puesto que una variable no cambia dentro del cuerpo de un método al que se le pasa como parámetro, siempre existirá un arco de resumen del nodo *actual_in* al nodo *actual_out* de esa variable.

3.3. Procedimientos y métodos

Existe una equivalencia funcional bastante evidente entre procedimiento y método de Java (ambos reciben un conjunto de parámetros de entrada y devuelven un valor, aunque los procedimientos lo hacen a través de un argumento de salida).

Aunque el SDG no soporta lenguajes cuyos procedimientos devuelven un valor, es posible realizar una pequeña modificación para que pueda hacerlo. El valor de retorno del método actúa como una variable de salida que puede estar influenciada por un parámetro del método, por lo que actúa como un *formal_out* y *actual_out*. Por tanto, añadiendo un nodo *output* en la definición del método (equivalente al *formal_out*) y otro nodo *output* en la llamada (equivalente a un *actual_out*) es posible soportar esta característica. Adicionalmente, el nodo que contiene la llamada será dependiente por datos del nodo *output* de la llamada y se deberán calcular los arcos de resumen que vayan de los *actual_in* correspondientes al *output* (si es necesario).

3.4. Sentencias soportadas de Java

El lenguaje Java dispone de un conjunto de sentencias definidas. Dado que el SDG analiza a nivel de sentencia, se ha de decidir qué conjunto de sentencias va a soportar nuestro SDG y, por tanto, el fragmentador. El conjunto de sentencias soportadas será el siguiente:

- *Expression*
- *If*
- *Do*
- *While*
- *For*
- *Foreach*
- *Continue*
- *Break*
- *Return*
- *Switch*

Las sentencias de salto incondicional (*continue*, *break*, *return*, *switch*) están soportadas gracias a extender el SDG, construyendo PPDGs en lugar de PDGs y aplicando el algoritmo de fragmentación interprocedural actualizado con los saltos incondicionales.

El resto de sentencias no incluidas en esta lista (como, por ejemplo, *try*, *assert*, etc.) serán ignoradas por el fragmentador y no aparecerán en el fragmento final.

Capítulo 4

Diseño

En esta sección, se describe el diseño y las particularidades técnicas del fragmentador, detallando cada decisión tomada y especificando la arquitectura, así como las librerías utilizadas y la especificación de los modelos creados.

Uno de los objetivos de este trabajo es que el fragmentador implementado tenga una estructura y una arquitectura bien definidas, de forma que sea fácilmente extensible y reusable para que se puedan añadir funcionalidades nuevas sin demasiado esfuerzo. Los principios SOLID son un conjunto de reglas para lenguajes orientados a objetos que permiten seguir patrones comunes y que representan una guía que ayuda al programador a escribir un código mantenible y extensible. A continuación, se detalla cada principio:

- S - Principio de responsabilidad única. Es la noción de que un objeto debe tener una única responsabilidad.
- O - Principio de abierto/cerrado. Especifica que toda entidad software debe estar abierta para su extensión, pero cerrada para su modificación.
- L - Principio de Liskov. Es la noción de que los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.
- I - Principio de la segregación de interfaz. Es la noción de que muchas interfaces cliente específicas son mejores que una interfaz de propósito general.
- D - Principio de inversión de la dependencia. La noción de que se debe depender de abstracciones, y no de implementaciones.

Estos principios se tendrán en cuenta a la hora de tomar decisiones de diseño para favorecer un código más limpio y reusable.

El fragmentador también dispondrá de un CLI (*Command Line Interface*) mediante el cual el usuario podrá invocar el fragmentador aportando la información necesaria para realizar la fragmentación.

4.1. Librerías utilizadas

El fragmentador tiene como base el uso del SDG para poder obtener el fragmento final. Dado que esta base es fundamentada principalmente en teoría de grafos, es necesaria una forma eficiente y efectiva de trabajar en este ámbito. Por ello, la mejor opción es usar una librería que ya nos proporcione esta forma de trabajar, en lugar de implementarla desde cero. Al usar una librería, se está favoreciendo la reutilización de código y la optimización de tiempo.

Como librería para trabajar con grafos, se ha escogido JGraphT[37, 38]. Se trata de una librería para Java muy completa que permite trabajar con grafos de forma cómoda y sencilla. Algunas de las características que hacen interesante usarla en el fragmentador son:

- Provee implementaciones de distintos tipos de grafos mediante clases para cada uno de ellos (grafo dirigido, no dirigido, multigrafo, etc.), aunque en este caso únicamente se usará el multigrafo dirigido.
- Flexibilidad en la representación tanto de nodos como de arcos, permitiendo usar estructuras de datos genéricas.
- Algoritmos y estrategias de recorrido de grafos como DFS, BFS, etc. implementadas por defecto.
- Rápida y eficiente, consume pocos recursos.
- Provee distintas formas de exportar grafos, que se realiza mediante objetos llamados *exporters*. Por defecto, proporciona varios exportadores en distintos formatos (por ejemplo, uno de ellos en lenguaje *dot*), lo que resulta útil a la hora de depurar y encontrar errores.

Por otra parte, analizar el código fuente de los programas también es un paso que se puede realizar mediante el uso de una librería, ahorrando tiempo y optimizando los recursos. JavaParser [22] es una librería para Java que permite analizar, transformar y generar código de manera simple. En esencia, permite construir, a partir de un código fuente Java, una representación de objetos Java en un determinado entorno Java. Esta representación es conocida como *Abstract Syntax Tree* (AST). Adicionalmente, provee de mecanismos para navegar por el AST, de forma que el usuario no tenga que preocuparse de implementarlos.

En el fragmentador, JavaParser puede ser usado, por una parte, para analizar más fácilmente el código fuente del programa que se va a fragmentar y, por otra, para el proceso contrario. Es decir, partiendo del AST del programa fragmentado, traducirlo a código para escribirlo en un fichero.

JavaParser hace uso del patrón de diseño visitador, de forma que tanto el AST como cualquiera de sus nodos es un objeto visitable. Este patrón permite crear objetos llamados visitadores que realicen ciertas operaciones en función del tipo del objeto visitado.

Por otra parte, para implementar el CLI, se hará uso de la librería Apache Commons CLI[1], la cual permite diseñar un CLI así como trabajar con los argumentos proporcionados por el usuario de forma sencilla.

[1] https://commons.apache.org/proper/commons-cli/

4.2. Arquitectura

A continuación, se detalla el diseño y la estructura de paquetes, así como los detalles acerca de los modelos creados. En primer lugar, se proporcionará una visión general de la estructura de paquetes y, a continuación, se detallarán los contenidos de cada paquete (modelos y otras estructuras de datos).

4.2.1. Visión general

El código del fragmentador se reparte en una serie de paquetes, los cuales se muestran a continuación:

- **cli** - Es el paquete que contiene la lógica correspondiente al CLI (*Command Line Interface*), y es el punto de entrada del usuario al fragmentador
- **core** - Es el paquete que contiene toda la lógica del fragmentador
- **core.arcs** - Contiene toda la lógica relacionada con los arcos de los grafos
- **core.graphs** - Contiene toda la lógica relacionada con los grafos
- **core.nodes** - Contiene toda la lógica relacionada con los nodos de los grafos
- **core.slicing** - Contiene la lógica relacionada con la fragmentación
- **core.utils** - Contiene clases y estructuras de datos auxiliares

4.2.2. Paquete *cli*

El paquete *cli* contiene la lógica de la interfaz de línea de comandos del fragmentador. La clase principal (punto de entrada del fragmentador) es la clase *Slicer*. Es en esta clase donde se analizan los argumentos de entrada al programa.

Para realizar la fragmentación, en su forma más básica, es necesario disponer de dos datos fundamentales. Esto es, el archivo Java a analizar y el criterio de fragmentación a utilizar. Por tanto, estos datos deben poder ser especificados mediante argumentos del CLI por el usuario.

Además del archivo y el criterio de fragmentación, es necesario añadir dos opciones más al CLI. En primer lugar, es posible que el archivo a analizar tenga dependencias de otros archivos (como, por ejemplo, que en algún punto del archivo a analizar se llame a un método declarado en otro archivo distinto). Para ofrecer la posibilidad de que esos archivos adicionales se incluyan en la fragmentación, se debe añadir un comando que permite indicar un archivo o un directorio de archivos Java. Por otra parte, se puede dar la opción de indicar un directorio de salida donde se depositará el resultado de la fragmentación que, por defecto, será el mismo directorio del archivo principal a fragmentar.

Teniendo todo esto en cuenta, el resumen de las opciones disponibles en el CLI sería el siguiente:

- *-c* - Indica el archivo Java y el criterio de fragmentación. Se trata de un *string* en formato *file#line:variable*, especificando de esta forma el archivo, la línea concreta del archivo y la variable
- *-i* (Opcional) - Indica un archivo o un directorio de archivos Java que serán incluidos en la fragmentación
- *-o* (Opcional) - Indica un directorio donde se depositarán los archivos resultantes de la fragmentación. Por defecto, es el mismo directorio del archivo principal

4.2.3. Paquete *core*

Modelos base

Nodo

Un nodo es un componente de un grafo que representa algún tipo de dato. En este caso, un nodo representa una sentencia del programa, pero es necesario almacenar más información para realizar la fragmentación. Para ello, se usará un objeto *GraphNode*, que se usará indistintamente en el CFG, PDG y SDG, y que tiene los siguientes atributos:

- **type** - Se trata de un enumerado *NodeType* que indica el tipo del nodo (véase el apartado de *Estructuras de datos y clases adicionales* más adelante)
- **id** - Un número identificador. Se genera en la creación del nodo y es único, lo que quiere decir que no puede existir ningún par de nodos cuyos identificadores sean iguales. Se asigna en el momento de la creación del nodo y se obtiene mediante la clase *IdHelper* (véase el apartado de *Estructuras de datos y clases adicionales* más adelante)
- **instruction** - Una cadena de texto que representa la sentencia
- **astNode** - El nodo del AST de JavaParser que corresponde a la sentencia
- **usedVariables** - Una lista de variables usadas en la sentencia
- **definedvariables** - Una lista de variables definidas en la sentencia
- **declaredVariables** - Una lista de variables declaradas en la sentencia

El valor de estos atributos se establecerá durante la creación del nodo y no deben ser cambiados después, por lo que serán atributos privados y finales. Esto hace que un objeto *GraphNode* sea un objeto inmutable, que satisface los principios SOLID. Concretamente, el principio de abierto/cerrado.

Además, la clase *GraphNode* admitirá un parámetro genérico el cual debe ser una clase hija de la clase *Node* de JavaParser. La clase del atributo *astNode* vendrá dada por dicho parámetro genérico.

Teniendo en cuenta que se va a trabajar con diferentes estructuras de datos que implican un orden, la clase *GraphNode* debe implementar la interfaz *Comparable*. Dado el hecho de que los

identificadores de cada nodo son únicos, la comparación entre nodos se puede realizar comparando sus identificadores de forma que, por tanto, sea posible ordenarlos según su identificador. Esto se requiere para poder ordenar los objetos *GraphNode* que estén incluidos en una colección (como, por ejemplo, una lista), si fuera necesario.

La cabecera de la clase *GraphNode* se puede observar en la figura 4.1:

```
1 public class GraphNode<N extends Node> implements Comparable<GraphNode<?>>
```

Figura 4.1: Cabecera de la clase *GraphNode*

Arco

Un arco representa una conexión entre nodos en un grafo, pudiendo ser unidireccional o bidireccional. En este caso, los arcos serán unidireccionales, ya que CFG, PDG y SDG son grafos dirigidos.

JGraphT provee una clase por defecto para representar arcos, *DefaultEdge*, la cual es conveniente usar ya que implementa algunos métodos básicos de Java necesarios para tratar con distintas estructuras de datos (*equals*, *hashCode*, etc,)

En este caso, se usará una clase abstracta llamada *Arc* que extenderá de la clase *DefaultEdge* para aprovechar estas implementaciones por defecto. Esta clase proveerá algunos métodos comunes para todos los arcos también, como *toString*, para ayudar al hacer depuración de código.

Grafo

Es posible representar una clase base para cualquier grafo de los que se van a implementar mediante una clase abstracta llamada *Graph*, que extiende de la clase *DirectedPseudograph* de JGraphT (dado que CFG, PDG y SDG son multigrafos dirigidos), la cual acepta dos parámetros genéricos (el primero indica la clase de los nodos, y el segundo la de los arcos), que corresponden con *GraphNode* y *Arc*, respectivamente. La cabecera de la clase *Graph* sería, pues, la que se muestra en la figura 4.2.

```
1 public abstract class Graph extends DirectedPseudograph<GraphNode<?>, Arc>
```

Figura 4.2: Cabecera de la clase *Graph*

Esta clase ofrece las operaciones básicas comunes a todos los tipos de grafo (CFG, PDG y SDG) y, además, métodos para representarlo de una forma determinada (exportarlo como una cadena de texto y en lenguaje *dot*), como son:

- **addNode** - Es un método que dispone de tres sobrecargas. La primera, acepta como parámetro un objeto *GraphNode* que simplemente se añade al grafo. La segunda, requiere como parámetro una cadena de texto representando la instrucción, y el nodo del AST correspondiente. Esta sobrecarga crea el nodo con dicha información mediante la fábrica por defecto de nodos (véase el apartado de *Estructuras de datos y clases adicionales* más

adelante). En la tercera sobrecarga, se requieren los mismos parámetros que en la segunda y además un objeto representando una fábrica de nodos mediante el cual se creará el nodo

- **findNodeById** - Encuentra un nodo a partir de su identificador
- **findNodeByASTNode** - Encuentra un nodo a partir de su nodo AST, usando el método *equalsWithRangeInCU* de la clase *ASTUtils* (véase el apartado de *Estructuras de datos y clases adicionales* más adelante)
- **getDOTExporter** - Devuelve un objeto *DOTExporter* de JGraphT con la información adecuada para representar el grafo en lenguaje *dot*

Implementaciones de modelos base

Arcos

Los tipos de arco disponibles en todos los posibles grafos se han modelado de forma que cada uno es representado mediante una clase que hereda de *Arc*. A continuación, se muestra la lista de todas las clases de arco:

- **ControlFlowArc** - Representa un arco de control de flujo. Usado en el CFG
- **ControlDependencyArc** - Representa un arco de dependencia de control. Usado en el PDG y SDG
- **DataDependencyArc** - Representa un arco de dependencia de datos. Usado en el PDG y SDG
- **ParameterInOutArc** - Representa un arco de parámetro de entrada o salida, indistintamente. Usado en el SDG
- **CallArc** - Representa un arco de llamada a método. Usado en el SDG
- **SummaryArc** - Representa un arco de resumen. Usado en el SDG
- **ReturnArc** - Representa un arco de retorno de valor. Usado en el SDG

Grafos

- **CFG** - Representa un CFG. Extiende de *Graph* y añade el método *addControlFlowArc*, que permite añadir específicamente un arco de control de flujo. También ofrece métodos para obtener la o las últimas definiciones y declaraciones de una variable, desde un nodo concreto. Estos métodos son necesarios para calcular las dependencias de datos en el PDG
- **ACFG** - Representa un ACFG. Extiende de CFG y permite añadir arcos no ejecutables
- **PDG** - Representa un PDG. Extiende de *Graph* y añade métodos para añadir arcos de dependencia de control y de datos. Tiene asociado un CFG.
- **PPDG** - Representa un PPDG. Extiende de PDG y cambia la forma del cálculo de las dependencias de control, así como el algoritmo de fragmentación, adaptados a las sentencias de salto incondicional

- **SDG** - Representa un SDG construido con PPDGs (i.e. añadiendo soporte a sentencias de salto incondicional). Extiende de *Graph* y añade métodos para añadir arcos de dependencia de control, de datos, de llamada, de resumen y de parámetro de entrada o salida. También dispone de un método para obtener la declaración de una variable que puede haber sido declarada en un método distinto. Tiene asociada una lista de CFGs correspondiente a cada uno de los métodos que se han analizado.

Estructuras de datos y clases adicionales

NodeType

NodeType es un enumerado que representa el tipo de un nodo de una forma más exacta. Esto resulta útil para diferenciar entre nodos que representan código real y entre nodos ficticios. Por lo general, los nodos reales representan normalmente una sentencia. Los tipos son los siguientes:

- **STATEMENT** - Indica que el nodo representa una sentencia (nodos reales)
- **METHOD_ENTER** - Indica que el nodo es un nodo de entrada a un método
- **METHOD_EXIT** - Indica que el nodo es un nodo de salida de un método
- **METHOD_CALL** - Indica que el nodo representa una llamada a método
- **ACTUAL_IN** - Indica que el nodo representa un parámetro real de entrada
- **ACTUAL_OUT** - Indica que el nodo representa un parámetro real de salida
- **FORMAL_IN** - Indica que el nodo representa un parámetro formal de entrada
- **FORMAL_OUT** - Indica que el nodo representa un parámetro formal de salida
- **METHOD_CALL_RETURN** - Indica que el nodo representa un nodo de valor de salida de una llamada
- **METHOD_OUTPUT** - Indica que el nodo representa un nodo de valor de salida de un método

IdHelper

Se trata de una clase *singleton*. Es decir, que solo se crea una instancia de la misma en todo el programa. La instancia tiene un único atributo, que se corresponde con el último identificador asignado, al que, al obtener un nuevo identificador, se le suma 1. De esta forma, esta clase se encarga de generar un identificador autoincremental que permite que cada identificador sea único. Satisface el principio de responsabilidad única (S de SOLID)

Clases Builder

Cada grafo de dependencia tiene asociada una clase *Builder* que se encarga de construir dicho grafo. Esta clase construye el grafo a partir del argumento necesario específico de cada grafo (por ejemplo, para un CFG, es necesario especificar el nodo del AST que corresponde a la declaración del método del que se va a contruir el grafo).

Algunas de estas clases dependen de otras clases que realizan tareas complejas. Por ejemplo, el *Builder* asociado al PDG, hace uso de forma interna de la clase *ControlDependencyBuilder* para calcular las dependencias de control. Esto satisface el principio de responsabilidad única de los principios SOLID.

Estas clases *Builder* implementan la interfaz *Visitor* de JavaParser. Por tanto, son en realidad visitadores de nodos del AST, que visitan los nodos necesarios para poder construir el grafo correctamente.

La lista completa de *Builder*s de los grafos es la siguiente:

- **CFGBuilder** - Construye un CFG a partir de un nodo *MethodDeclaration* de JavaParser y un CFG vacío
- **ACFGBuilder** - Construye un ACFG a partir de un nodo *MethodDeclaration* de JavaParser y un ACFG vacío
- **PDGBuilder** - Construye un PDG a partir de un nodo *MethodDeclaration* de JavaParser, un PDG vacío y, opcionalmente, un CFG construido. Si no se proporciona un CFG construido, se construye. Usa las clases *ControlDependencyBuilder* y *DataDependencyBuilder* para calcular las dependencias de control y de datos, respectivamente
- **PPDGBuilder** - Extiende de *PDGBuilder*, con la única diferencia de que admite un ACFG en lugar de un CFG y calcula las dependencias de datos mediante la clase *PPControlDependencyBuilder*
- **SDGBuilder** - Construye un SDG a partir de un conjunto de nodos *CompilationUnit*s de JavaParser. Para cada método, crea su PPDG. Usa la clase *MethodCallVisitor* para añadir los nodos *actual* de cada llamada, y enlazarlos con los correspondientes nodos formales, así como el nodo de llamada con el nodo de la declaración del método al que hace referencia. También, usa la clase *SummaryArcsBuilder* para añadir los arcos de resumen

ASTUtils

Esta clase contiene una colección de métodos estáticos que proporcionan funcionalidades útiles relacionadas con el AST. Por ejemplo, proporciona un método por el cual se puede comprobar si dos nodos, además de ser iguales, también pertenecen a la misma clase y están en la misma posición en el fichero. Esto es debido a que, aunque cada nodo del AST de JavaParser implementa el método *equals* para comprobar la igualdad con otro nodo, no proporciona una forma de comprobar que, además de ser iguales, pertenecen a la misma clase y a la misma posición en el archivo (es decir, que representan exactamente la misma instrucción del mismo programa).

Fábricas de nodos. NodeFactory

La interfaz *NodeFactory* representa una fábrica genérica mediante la cual crear un nuevo *GraphNode*, ofreciendo dos métodos distintos. El primero, crea un nuevo nodo a partir de la instrucción en forma de *String*, el nodo del AST, una lista de variables usadas, otra lista de variables definidas y, por último, otra lista de variables declaradas. Este primer método no analiza el nodo del AST en busca de variables usadas, definidas y declaradas, pues la información ya viene dada. El segundo método crea también un nodo pero realizando este análisis, por lo que solo es necesaria la instrucción en forma de *String* y el nodo del AST.

Como implementación de la interfaz *NodeFactory* encontramos la clase *TypeNodeFactory*, la cual dispone de un método estático que devuelve una instancia de la clase, en función del tipo de nodo que se le especifique. Esta instancia creará siempre nodos del tipo especificado por lo que, para crear otro tipo de nodos, se deberá crear otra instancia distinta especificando el tipo deseado.

La fábrica de nodos por defecto es una instancia de *TypeNodeFactory* que crea nodos de tipo *STATEMENT*, ya que es el tipo más común.

Slice y otras estructuras relacionadas con la fragmentación

El SDG implementa la interfaz *Sliceable*, que contiene un método *slice*, que admite como parámetro un objeto *SlicingCriterion* y que devuelve un objeto *Slice*.

Por un lado, la clase *SlicingCriterion* es una clase abstracta que representa un criterio de fragmentación. Esta clase tiene un único método abstracto que sirve para encontrar el nodo en función del tipo de criterio de fragmentación utilizado (la implementación concreta) llamado *findNode*. También, tiene un campo de tipo *String* que representa el nombre de la variable especificada por el usuario. Una implementación concreta de *SlicingCriterion* es la clase *LineNumberCriterion*, la cual la extiende y añade un campo que indica la línea del archivo especificada por el usuario, a la vez que implementa el método *findNode*.

Por otro lado, la clase *Slice* representa el fragmento final, con métodos que permiten obtener un AST a partir de los nodos correspondientes al fragmento únicamente. Dispone de un campo en forma de lista que representa esta información, y un método *toAST* que construye un AST a partir de los nodos del fragmento.

Adicionalmente, puede existir distintos algoritmos de fragmentación, para lo que se define una interfaz *SlicingAlgorithm*, la cual define un único método *traverse*, que acepta un *GraphNode* (el nodo correspondiente al criterio de fragmentación) y devuelve un objeto *Slice*. La clase *ClassicSlicingAlgorithm* es una implementación del algoritmo clásico de fragmentación del SDG, la cual implementa la interfaz *SlicingAlgorithm*.

Capítulo 5

Implementación

En esta sección, se detalla la implementación del fragmentador: En primer lugar, la implementación del CLI, en segundo lugar, el proceso concreto de construcción del SDG, junto con el CFG y ACFG, PDG y PPDG y, por último, cómo se obtiene el fragmento una vez construido el SDG.

5.1. CLI

Como ya se ha comentado, la clase que contiene toda la lógica del CLI y que representa el punto de entrada es la clase *Slicer*. En esta clase se hace uso de la librería de Apache Commons CLI para implementar un CLI de forma simple.

Con Apache Commons CLI, se pueden declarar los argumentos disponibles definiendo un objeto *Options*, al cual se le pueden añadir distintas opciones. En este caso, se añaden las opciones descritas anteriormente, en el apartado 4.2.2.

El objetivo es, para cada opción del fragmentador, crear un objeto *Option* que sea añadido al objeto *Options* principal. Para crear un objeto *Option*, es posible llamar al método estático *builder*, al que se le pasa como parámetro el comando con el que se invoca la opción. También, es posible llamar al método *hasArg* para indicar que la opción requiere de un argumento. Finalmente, se llama al método *build* para construir el objeto *Option* configurado.

Como ejemplo, en la figura 5.1 se muestra el código para la opción *-c*:

```
1     Options options = new Options();
2
3     Option criterion = Option
4                 .builder("c").longOpt("criterion")
5                 .hasArg().argName("file#line:var")
6                 .build();
7
8     options.addOption(criterion);
```

Figura 5.1: Código que describe la opción *-c* y la añade al CLI

A continuación, se describen las acciones que se realizan con cada argumento especificado:

- *-c* - Se aplica una expresión regular para obtener el archivo, la línea y la variable de forma separada a partir de la cadena de texto introducida (recordemos, de la forma *file#line:variable*). Si falta alguno de estos tres datos, se devuelve un error
- *-i* - Se obtienen las *CompilationUnit*s de las clases incluidas mediante JavaParser, y se almacenan en una lista para tenerlas en cuenta a la hora de realizar la fragmentación
- *-o* - Se configura una variable que indica el nuevo directorio de salida especificado por el argumento

5.2. Construcción del SDG

En este apartado se describen con detalle las configuraciones iniciales realizadas (en concreto, de JavaParser) así como los pasos seguidos a la hora de crear un CFG, ACFG, PDG, PPDG y un SDG.

5.2.1. Configurando JavaParser

Antes de comenzar a construir el SDG, se han de realizar unas configuraciones previas relacionadas con JavaParser. En concreto, una de estas configuraciones está relacionada con el *symbol solver*, el cual permite que algunas expresiones y las llamadas a método pueden ser resueltas y obtener el objeto al que se refieren (por ejemplo, en caso de llamada a método, obtener la declaración del método al que se llama). Este *symbol solver* se puede configurar de forma estática, de forma que se le puede indicar librerías adicionales de Java que ha de tener en cuenta a la hora de resolver las referencias.

Por ejemplo, opcionalmente, se le puede indicar que resuelva referencias a los métodos de la JRE. Si se le indica esto, por ejemplo, la llamada *System.out.print()* podría ser resuelta, y se obtendría la declaración del método *print* de la JRE. En caso contrario, no se obtendría y la resolución de la referencia fallaría.

En el caso del fragmentador, se incluye la opción de resolver llamadas a la JRE para realizar un análisis más completo. Además, debe tener en cuenta las clases adicionales indicadas mediante

el comando *-i* en el CLI, por lo que también se añaden dichas clases, en caso de que el usuario las haya especificado, en la configuración.

Adicionalmente, JavaParser permite analizar también los comentarios que haya en el código. Esta opción está desactivada en el fragmentador ya que no ayudan de ninguna forma a realizar la fragmentación, por lo que simplemente se ignorarán.

5.2.2. Construyendo el CFG y el ACFG

El CFG se construye mediante la clase *CFGBuilder*. Esta clase implementa la clase *VoidVisitor* de JavaParser, e implementa un método por cada *statement* que se va a cubrir en el fragmentador (ver sección 3).

El objeto *CFGBuilder* está pensado para admitir como punto de entrada un nodo del tipo *MethodDeclaration*. La tarea de este objeto será recorrer los sentencias del cuerpo del método, creando los *GraphNode*s y los *ControlFlowArc*s correspondientes.

Para sentencias no condicionales, el procedimiento para enlazar un nodo con otro es muy simple. Es suficiente con mantener una referencia al nodo creado anteriormente, de forma que al crear un nuevo nodo, si existe esa referencia, se crea un nuevo arco del nodo anterior al que se crea en el momento.

No obstante, para sentencias condicionales, el funcionamiento no es tan simple. Puesto que se deben cubrir todas las posibles ejecuciones, se deben cubrir todas las posibles bifurcaciones de ejecución del programa, y por tanto, para cada bifurcación hay que almacenar el o los últimos nodos creados para enlazarlo con el siguiente.

Para generalizar estos dos casos anteriores, en lugar de mantener una referencia a un único nodo, se utiliza una lista donde se almacenan los nodos que hay que enlazar al siguiente. De este modo, al crear un nuevo nodo, se extraen de la lista los nodos anteriores y se crean los arcos desde cada uno de esos nodos al nuevo.

Además, existen otro tipo de sentencias, que son los de salto incondicional (*break*, *continue* y *return*). Este tipo de sentencias han de tratarse de manera especial, puesto que representan un salto a un punto del programa que no tiene por qué necesariamente ser el o los nodos inmediatamente anteriores. En caso de *break* y *continue*, no se puede saber, en el momento de analizarlos, cual será el siguiente nodo. Por ello, es necesario que haya dos pilas independientes, una para los *break* y otra para los *continue*, donde se vayan introduciendo los sentencias de cada tipo encontrados. Esto también obliga a que, al añadir un nuevo nodo, además de recorrer la lista de últimos nodos, haya que obtener el primer elemento de cada una de las pilas para enlazarlo con el nuevo. En caso de *return*, dado que representa un salto incondicional al final del método, únicamente almacenamos cada ocurrencia en forma de nodo en una lista, para enlazarlos con el nodo STOP al final.

El proceso de construcción general, pues, sería el siguiente: En primer lugar, antes de empezar a recorrer el método, se crea como nodo de partida el nodo *Start*. Este se añade a la lista de nodos colgantes, por lo que el siguiente nodo se conectará directamente a este. Una vez hecho esto, se comienza a procesar las sentencias del cuerpo del método de la forma descrita anteriormente para cada una de ellas. Por último, se crea el nodo *Stop* y se enlazan los nodos colgantes y, además,

los nodos *return* que se hayan encontrado durante el procesamiento del cuerpo del método.

Adicionalmente, en el CFG se analizan los parámetros del método analizado y se incluyen los nodos formales y el nodo *output* del método, los cuales facilitarán en gran medida la construcción del SDG, ya que estarán disponibles a la hora de realizar el cálculo de las dependencias de datos, incluyendo los parámetros por defecto. Los nodos *formal_in* se situarán entre el nodo START y la primera sentencia del cuerpo, mientras que los nodos *formal_out* y *output* se encontrarán entre la última sentencia del cuerpo y el nodo STOP, en ese orden. Estos nodos no influyen en la construcción del CFG, por lo que no hay problema en añadirlos en este paso.

```
1  public class CFGExample {
2
3      public static int min(int x, int y) {
4          int res = 0;
5
6          if (x > y) {
7              res = y;
8          } else {
9              res = x;
10         }
11
12         return res;
13     }
14
15 }
```

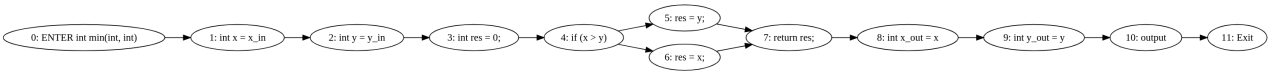

Figura 5.2: Método de Java y el CFG resultante

En cuanto al ACFG, el método de construcción es exactamente el mismo al del CFG, realizado mediante la clase *ACFGBuilder* que extiende de *CFGBuilder*. La única diferencia es que esta clase añade los arcos no ejecutables hacia los nodos que representan sentencias de salto incondicional.

5.2.3. Construyendo el PDG y el PPDG

El PDG es construido mediante la clase *PDGBuilder*. Esta clase implementa la clase *VoidVisitor* de JavaParser, y la entrada esperada es un nodo *MethodDeclaration* del AST.

Antes de poder costruir el PDG, es necesario disponer del CFG del método que se va a analizar. Para ello, el constructor de *PDGBuilder*, admite como parámetro opcional un CFG ya construido. En caso de que no se proporcione, se añade el paso extra de construcción del CFG del método.

Una vez se tiene el CFG del método construido, en el PDG se incluyen todos los nodos existentes del CFG, exceptuando el nodo STOP. Como un nodo es un *GraphNode*, es un objeto inmutable, independiente del grafo en el que esté, y contiene información útil para CFG, PDG y SDG, se puede usar la misma instancia directamente sin crear nuevos nodos.

Una vez incluidos los nodos, se procede a calcular las dependencias de control y las dependencias de datos de cada uno de ellos.

Dependencias de control

La tarea de calcular las dependencias de control se delega en la clase *ControlDependencyBuilder*, la cual mantiene una referencia al CFG ya construido y al PDG semi-construido (sin dependencias de control calculadas). El CFG se usa para realizar el cálculo, y al PDG se le añaden los arcos de dependencia de control donde corresponda.

Las dependencias de control se calculan de una forma simple. En concreto, se recorre cada par de nodos del PDG y, para cada par, se comprueba si uno es dependiente por control del otro. Esta comprobación se realiza mediante las propiedades de la definición 3 (dependencia de control) y teniendo en cuenta la Definición 2 (postdominancia).

De forma resumida, el proceso lógico que se sigue es el siguiente: Para comprobar si un nodo a es dependiente por control de otro nodo b, se comprueba el grado de salida de a. En caso de ser menor de 2, se descarta por definición. En caso de ser 2, se comprueba si a es postdominado por b. Si existe un camino desde a en el que sea postdominado por b y otro en que no, entonces b es dependiente por control de a. En la figura 5.3, se puede observar de forma gráfica el escenario descrito.

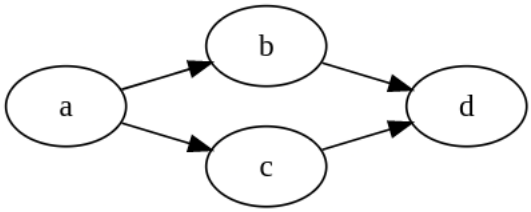

Figura 5.3: Ejemplo en el que b (y también c) es dependiente por control de a

Dependencias de datos

Las dependencias de datos se calculan mediante la clase *DataDependencyBuilder*. Esta clase recorre todos los nodos del PDG y, para cada variable usada en cada uno de ellos, partiendo del mismo nodo en el CFG, obtiene el nodo en el que la variable fue definida por última vez (o los nodos si existe más de un camino y más de una definición de la variable en distintos caminos), en base a la Definición 3 (dependencia de datos). A continuación, se crea un arco de dependencia de datos desde cada nodo de definición encontrado hasta el nodo que se está analizando en ese momento.

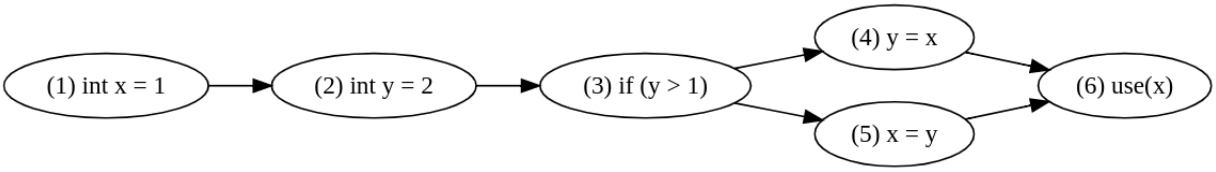

Figura 5.4: Ejemplo en el que existe dependencia de datos de la variable x de (1) a (4) y (6), y de (5) a (6), y de la variable y de (2) a (3) y (5)

Para construir el PPDG, se realiza el mismo procedimiento descrito en la sección 2. Es decir, el PPDG se construye mediante una clase *PPDGBuilder*, la cual extiende directamente de *PDG-Builder*. La diferencia radica en la forma de calcular las dependencias de control, dado que se realizan usando la clase *PPControlDependencyBuilder*, la cual extiende de *ControlDependency-Builder*. Esta clase únicamente sobreescribe y modifica el método de comprobación de si un nodo postdomina a otro nodo, de forma que lo realiza según se describe en la definición 5.

5.2.4. Construyendo el SDG

Para construir el SDG, se necesita obtener los ACFGs y PPDGs de cada uno de los métodos a analizar. Una vez obtenidos, se han de realizar las siguientes tareas:

1. Crear y enlazar nodos de llamada con los nodos de declaración de método correspondiente
2. Añadir los nodos *actual* y el nodo *output* al nodo de llamada. Los nodos *actual* se corresponden con los argumentos de la llamada y el nodo *output* al valor devuelto tras la ejecución del método
3. Enlazar los nodos *actual* y el nodo *output* con los nodos formales y el nodo *output* de la declaración del método, respectivamente
4. Añadir los arcos de resumen

Para los puntos 1, 2 y 3, se hace uso de la clase *MethodCallVisitor*. Para el punto 4, se hace uso de la clase *SummaryArcsBuilder*.

Analizando las llamadas a métodos

Para analizar las llamadas a métodos y conectarlas con sus declaraciones, se visita el AST para encontrar expresiones de llamada de método (*MethodCallExpr* en JavaParser). Cuando se encuentra un nodo que contiene una llamada, lo primero que se realiza es la búsqueda de la declaración de la misma. Esto se hace mediante el método *resolve* del objeto *MethodCallExpr*, el cual permite obtener el nodo del AST que corresponde a la declaración del método mediante el *symbol solver* de JavaParser. Si el método no se encuentra, esta llamada se ignora, pues no es posible analizar nada más.

A continuación, una vez encontrado el nodo de declaración del método, se añade al SDG un nuevo nodo ficticio que contiene únicamente la llamada de la función, el cual es dependiente por

control del nodo que contenía la llamada. Una vez creado, se enlaza este nuevo nodo ficticio con el nodo de la declaración del método mediante un arco de llamada.

Después de crear el nodo de llamada, se procede a analizar los argumentos para crear los nodos *actual*. En concreto, se utilizan los parámetros de la declaración del método, que aportan más información acerca del argumento. Para cada parámetro, se realizan las siguientes acciones:

1. Se crea un nodo *actual_in*, se añade al SDG y se añade un arco de dependencia de control del nodo de la llamada a este
2. Se corrigen las dependencias de datos del nodo original de la llamada: Se mueven los arcos que corresponda de forma que el nuevo nodo *actual_in* sea el dependiente por datos de la variable correspondiente, en lugar del nodo original de la llamada
3. Se crea un arco de parámetro de entrada/salida desde el nodo *actual_in* al *formal_in*. Para comprobar qué *formal_in* le corresponde, se comprueba que el nombre de la variable de entrada (terminada en *_in*) corresponda tanto en el nodo *actual_in* como en el *formal_in*
4. Se crea un nodo *actual_out*, se añade al SDG y se añade un arco de dependencia de control del nodo de llamada a este
5. Se corrigen las dependencias de datos: Se mueven los arcos salientes del nodo original de llamada que correspondan con la misma variable de forma que ahora salgan del nodo *actual_out* al nodo dependiente
6. Se crea un arco de parámetro de entrada/salida desde el nodo *formal_out* al *actual_out*, identificando el nodo *formal_out* de la misma forma que en el paso 3

Por último, se comprueba si el método devuelve un valor (no es *void*). En caso negativo, no se hace nada más. En caso de que devuelva algún valor, se crea el nodo *output* de la llamada y se añade al SDG. Se añade un arco de dependencia de control desde el nodo ficticio de la llamada hasta el nodo *output*, y se añade otro arco de dependencia de datos desde el nodo *output* hasta el nodo original de la llamada. Después, se añade un arco de retorno desde el nodo *output* del método al nodo *output* de la llamada.

Calculando arcos de resumen

En último lugar, para completar el SDG, debemos calcular los arcos de resumen. Como se ha descrito anteriormente, un arco de resumen sirve para representar las dependencias transitivas de las llamadas a procedimientos (en este caso, métodos).

Dado que un arco de resumen existe de un *actual_in* a un *actual_out* o *output* cuando la variable pasada como parámetro no es redefinida en el cuerpo del método al que se le pasa, y teniendo en cuenta que en el alcance que vamos a tener de Java las variables no pueden ser redefinidas de ninguna forma (recordemos que solo soportamos variables primitivas pasadas como parámetro), siempre existirá un arco de resumen de *actual_in* al *actual_out* correspondiente. Sin embargo, sí hay que calcular los arcos de resumen que van de *actual_in* a *output*, dado que el resultado final del método puede estar influido por algún parámetro de entrada.

El cálculo de los arcos de resumen requiere de un grafo adicional, llamado arco de llamadas (*Call graph*). Este grafo representa las llamadas realizadas entre métodos. Con este grafo, es

posible calcular fácilmente los arcos de resumen para casos en los que existen métodos que se llamen entre sí (recíprocos) o métodos que se llamen a sí mismos (recursivos).

El grafo de llamada es un multigrafo dirigido que se construye mediante visitadores de JavaParser donde cada declaración de método es un nodo y cada arco representa una llamada. En primer lugar, se visitan todas las declaraciones de método (*MethodDeclaration*) y se añaden como nodos al grafo. En segundo lugar, se visitan de nuevo las declaraciones de método, guardando la referencia al método actual, y se visitan también las expresiones de llamada a método (*MethodCallExpr*), de forma que se intentan resolver y, si se resuelven satisfactoriamente, se añade el arco desde la declaración de método almacenado (el método actual) hasta la declaración del método resuelto. Esta forma de calcular los arcos de resumen tiene en cuenta la alteración del SDG para poder soportar sentencias de saltos incondicionales, por lo que se evita recorrer arcos de dependencia de control que conduzcan a un pseudo-predicado.

```
1  public class CallGraphExample {
2
3      public static void main() {
4          int x = 1;
5          int y = 2;
6
7          int res = fact(x);
8
9          System.out.println(res);
10     }
11
12     public static int fact(int n) {
13         if (n == 0) {
14             return 1;
15         }
16
17         int nextN = n - 1;
18         int b = fact(nextN);
19
20         return mult(n, b);
21     }
22
23     public static int mult(int n, int m) {
24         if (m == 0) {
25             return 0;
26         }
27
28         int multRes = mult(n, m - 1);
29         int sum = n + multRes;
30
31         return sum;
32     }
33
34 }
```

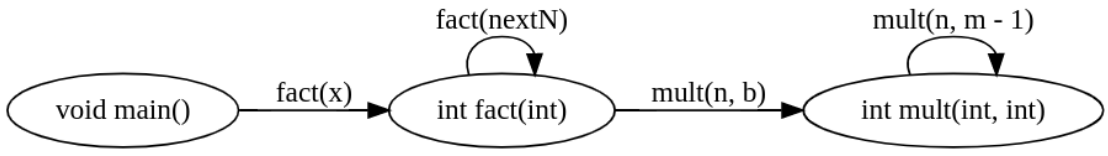

Figura 5.5: Ejemplo de un programa y su respectivo *call graph*

Una vez construido el grafo de llamada, se procede a realizar un análisis de punto fijo sobre el mismo, de forma que se calculan los arcos de resumen para cada nodo del grafo de llamada (declaración de método) repetidas veces, de forma que, con cada nuevo cálculo, se añaden nuevos arcos. El análisis termina cuando se haga un nuevo cálculo y no se añada ningún arco nuevo. En ese momento, el cálculo de los arcos de resumen habrá terminado y el SDG estará completo.

```
1  public class SDGExample {
2
3      public static void main() {
4          int x = 1;
5          int y = 2;
6
7          int res = sum(x, y);
8
9          return res;
10     }
11
12     public static int sum(int a, int b) {
13         int result = a + b;
14         return result;
15     }
16
17 }
```

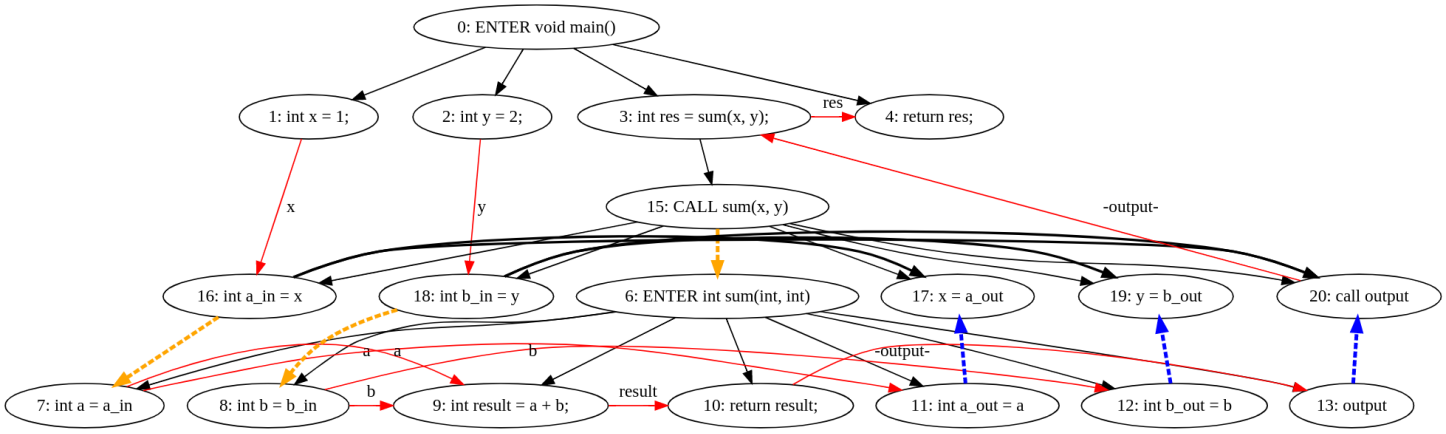

Figura 5.6: Ejemplo de programa en Java y su respectivo SDG construido. Las flechas negras delgadas representan dependencias de control, las flechas rojas dependencias de datos, las flechas amarillas representan o bien llamada a método o parámetro de entrada, las flechas azules representan o bien parámetro de salida o retorno de valor, y las flechas negras en negrita representan los arcos de resumen

5.3. Obteniendo el fragmento

A partir de la información especificada por el usuario, se crea un objeto de tipo *FileLineSlicingCriterion*, el cual contiene la información del criterio de fragmentación para el fichero. Este objeto se le pasa como parámetro al método *slice* del SDG, el cual, en primer lugar, y haciendo uso del método *findNode* del objeto *SlicingCriterion* busca el nodo correspondiente al criterio de

fragmentación. Si no se encuentra, se devuelve un error. Si se encuentra, se aplica el algoritmo clásico de fragmentación para el SDG, por lo que se crea un nuevo objeto *ClassicSlicingAlgorithm*, al que se le pasa el SDG y, además, se ejecuta su método *traverse*, al cual se le proporciona el nodo correspondiente al criterio de fragmentación, y devuelve un objeto *Slice*.

El método *traverse* de la clase *ClassicSlicingAlgorithm* aplica el algoritmo clásico del SDG. Este consiste, recordemos, en recorrer los nodos del grafo en dos pasadas desde el nodo correspondiente al criterio de fragmentación, de la siguiente manera:

En la primera pasada, la forma de recorrer los nodos es muy parecida al método de fragmentación del PDG. Es decir, consiste en recorrer los arcos de entrada de los nodos visitados recursivamente. En el SDG hay una restricción en cuanto a los arcos que se han de recorrer. En concreto, se han de recorrer todos los arcos exceptuando los arcos de tipo de parámetro de salida (*ParameterOutArc*) y de retorno (*ReturnArc*).

En la segunda pasada, partiendo también del nodo correspondiente al criterio de fragmentación, se recorren todos los arcos de salida de los nodos visitados recursivamente, exceptuando los arcos de llamada (*CallArc*) y los arcos de parámetro de entrada (*ParameterIn*).

Es necesario recordar que durante el recorrido de estas dos pasadas, se evita recorrer arcos de dependencia de control que conduzcan a pseudo-predicados, a no ser que se trate del criterio de fragmentación.

Como resultado de la ejecución del método *traverse*, se devuelve un objeto *Slice*, a partir del cual se puede obtener el AST resultante mediante el método *toAST*, el cual crea un nuevo AST clonando los nodos correspondientes al fragmento, y elimina los nodos ficticios, ya que no representan sentencias reales. El AST resultante, finalmente, se escribe en un archivo o en varios, dependiendo si el fragmento abarca más de un solo archivo.

Capítulo 6

Ejemplo de caso de uso

A continuación, se expondrá un ejemplo completo de ejecución, mostrando un programa Java a fragmentar con un determinado criterio de fragmentación, y se obtendrá su SDG y su fragmento final.

El programa a fragmentar se muestra en la figura 6.1.

```
1  public class Program {
2
3      public static void main(String[] arcs) {
4          int x = 100;
5          int y = 5;
6          int z = 50;
7
8          int factY = fact(y);
9
10         if (z < x) {
11             z = fact(x);
12         }
13
14         x = sum(x, factY);
15
16         System.out.println(x);
17         System.out.println(z);
18     }
19
20     public static int sum(int a, int b) {
21         int result = a + b;
22         return result;
23     }
24
25     public static int fact(int n) {
26         if (n == 0) {
27             return 1;
28         }
29
30         int nextArg = n - 1;
31         int next = fact(nextArg);
32
33         return n * next;
34     }
35 }
```

Figura 6.1: Programa de ejemplo a fragmentar

El criterio de fragmentación que se usará es el $\langle 17, z \rangle$. De esta forma, obtendremos las partes del código que han afectado a la variable z en el momento en el que es imprimida por pantalla.

El comando para ejecutar el fragmentador es el siguiente:

java -jar slicer.jar -c ./example/Program.java#17:z -o ./slice

Después de ejecutar el comando, en la carpeta *slice*, nos encontramos el archivo *Program.java*, que contiene el siguiente código:

```
1  public class Program {
2
3      public static void main(String[] arcs) {
4          int x = 100;
5          int z = 50;
6          if (z < x) {
7              z = fact(x);
8          }
9          System.out.println(z);
10     }
11
12     public static int fact(int n) {
13         if (n == 0) {
14             return 1;
15         }
16         int nextArg = n - 1;
17         int next = fact(nextArg);
18         return n * next;
19     }
20 }
```

Figura 6.2: Fragmento resultante de la fragmentación del programa de la figura 6.1

Como se puede observar, ni la variable y ni el método *sum* son relevantes para el valor de z en el punto indicado, por lo que no se incluyen en el fragmento final.

Finalmente, el SDG generado para el programa y sobre el cual se ha realizado la fragmentación se muestra en la figura 6.3:

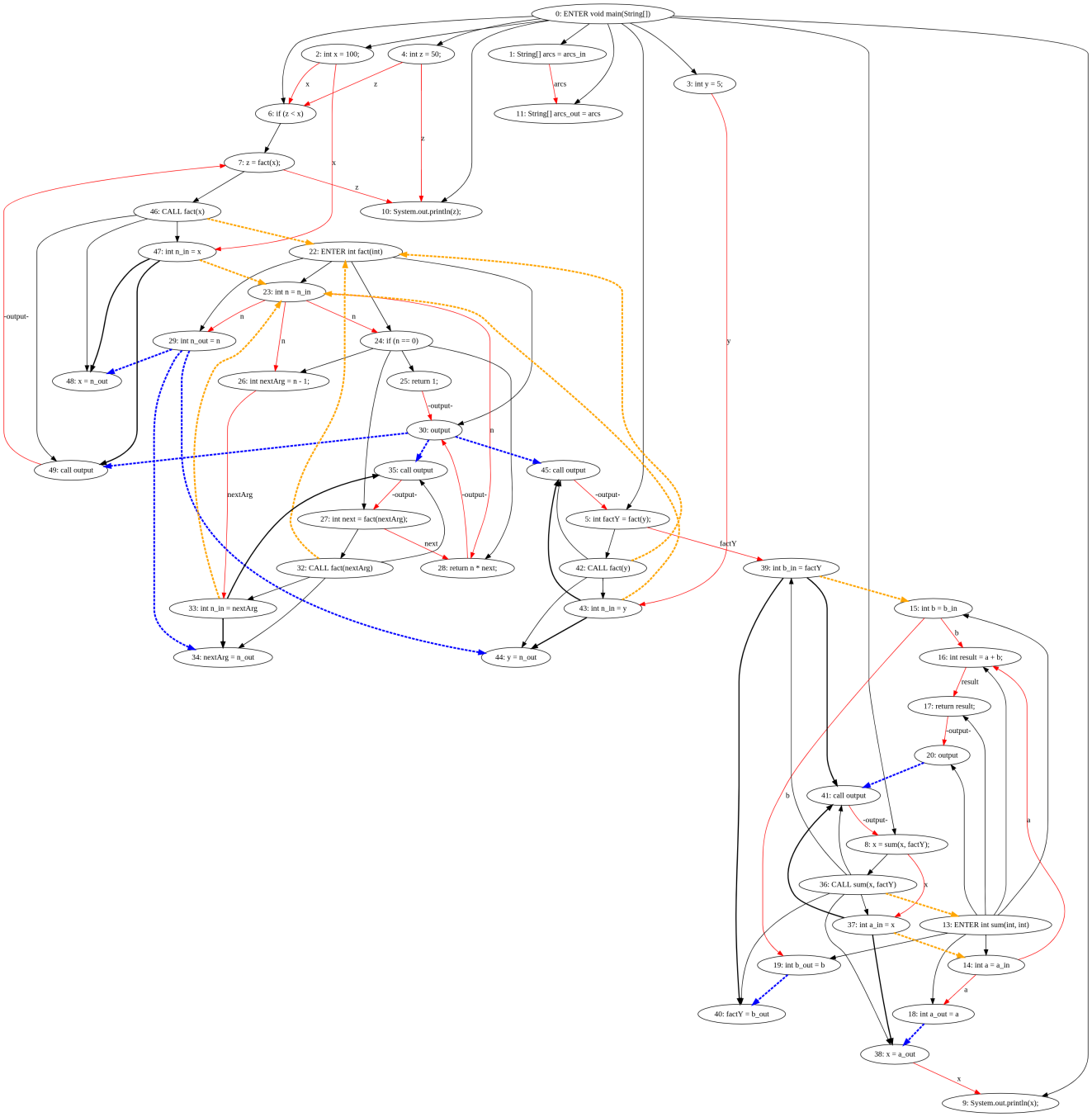

Figura 6.3: SDG correspondiente al programa de la figura 6.1

Capítulo 7

Evaluación

A la hora de evaluar un fragmento, existen un conjunto de métricas que podemos usar para comparar unos fragmentos con otros. La más importante ha sido referenciada a lo largo de este documento y se ha establecido como objetivo: la completitud.

Un fragmento es **completo** cuando incluye todas las sentencias que afectan al criterio de fragmentación. Este es el principal objetivo que se busca a la hora de desarrollar un fragmentador. Por ejemplo, un fragmento que incluya todas las sentencias del programa original es completo, pues incluye todas las sentencias que afectan al criterio de fragmentación (aunque incluye otras que no afectan), sin embargo, este sería un fragmento de poca utilidad.

Por otro lado, existe otra métrica: la **corrección** del fragmento. Un fragmento es correcto cuando excluye todas las sentencias que no afectan al criterio de fragmentación. Dicho de otra forma, solo incluye las sentencias que, exclusivamente, afectan al criterio de fragmentación. Esto hace que los fragmentos sean más pequeños, lo que ayuda en gran medida al programador y optimiza el tiempo y los recursos en tareas como la depuración y la comprensión de código, como se ha comentado anteriormente. No obstante, es una propiedad que es en general imposible de comprobar formalmente, dado que encontrar el fragmento mínimo es un problema indecidible [1].

Aunque el objetivo de este trabajo sea el de obtener fragmentos completos, no es posible evaluar formalmente las mencionadas propiedades de los fragmentos generados, dado que su validación constituye en sí misma un ámbito de estudio donde aún existen problemas no resueltos. Por lo tanto, y dado que la validación de los fragmentos no forma parte del alcance del presente trabajo, no se evaluará ninguna de estas propiedades.

Sin embargo, sí es posible realizar evaluaciones manuales con ejemplos prácticos, demostrando que se realiza la cobertura descrita para el lenguaje Java. En concreto, el conjunto de sentencias especificado en el capítulo 3. Estos ejemplos consisten en una batería de programas para los cuales se obtiene un fragmento, el cual podremos comprobar que es completo y correcto. A continuación, se muestra una lista de algunos programas usados durante la validación. En todos los casos, los fragmentos obtenidos son correctos y completos (el fragmento mínimo):

Sentencia *Expression*

```
1  public class Expression {
2
3      public static void main(String[] args) {
4          int x = 1;
5
6          x = 2;
7          x = 3;
8          x = 4;
9
10         System.out.println(x);
11     }
12 }
```

```
1  public class Expression {
2
3      public static void main(String[] args) {
4          int x = 1;
5          x = 4;
6          System.out.println(x);
7      }
8  }
```

Figura 7.1: Programa Java que contiene sentencias de expresión (arriba) y el resultado de ejecutar el fragmentador con respecto a $\langle 10, x \rangle$

En la figura 7.1, se demuestra la cobertura que el fragmentador realiza sobre las sentencias de expresión, dado que, respecto al criterio de fragmentación $\langle 10, x \rangle$, las líneas 6 y 7 no deben incluirse en el fragmento final. El fragmentador obtiene el fragmento completo y correcto, pues escoge las instrucciones que afectan al criterio de fragmentación (4, 8) y las incluye en el fragmento final, excluyendo el resto.

Sentencia *If*

```
1 public class If {
2   public static void main(String[] args) {
3     int testscore = 76;
4     char grade;
5
6     if (testscore >= 90) {
7       grade = 'A';
8     } else if (testscore >= 80) {
9       grade = 'B';
10    } else if (testscore >= 70) {
11      grade = 'C';
12    } else if (testscore >= 60) {
13      grade = 'D';
14    } else {
15      grade = 'F';
16    }
17    System.out.println("Grade␣=␣" + grade);
18    System.out.println("Score␣=␣" + testscore);
19  }
20 }
```

```
1 public class If {
2
3     public static void main(String[] args) {
4         int testscore = 76;
5         System.out.println("Score␣=␣" + testscore);
6     }
7 }
```

Figura 7.2: Programa Java que contiene una sentencia if (arriba) y el resultado de ejecutar el fragmentador con respecto a $\langle 18, testscore \rangle$

En la figura 7.2, se demuestra la cobertura que el fragmentador realiza sobre las sentencias if. Respecto al criterio de fragmentación $\langle 18, testscore \rangle$, toda la sentencia if debería excluirse, dado que no es relevante para la variable *testscore*. Así ocurre también con la variable *grade*. El fragmentador, de nuevo, obtiene el fragmento completo y correcto escogiendo las instrucciones estrictamente necesarias.

Sentencia *Do*

```
1  public class Do {
2
3      public static void main(String[] args) {
4          int x = 1;
5          int y = 2;
6
7          do {
8              x++;
9              y++;
10         } while (x > 1);
11
12         System.out.println(x);
13         System.out.println(y);
14     }
15 }
```

```
1  public class Do {
2
3      public static void main(String[] args) {
4          int x = 1;
5          do {
6              x++;
7          } while (x > 1);
8          System.out.println(x);
9      }
10 }
```

Figura 7.3: Programa Java que contiene una sentencia do (arriba) y el resultado de ejecutar el fragmentador con respecto a $\langle 12, x \rangle$

En la figura 7.3, se demuestra la cobertura que el fragmentador realiza sobre las sentencias *do*. Respecto al criterio de fragmentación $\langle 12, x \rangle$, se excluyen todas las instrucciones relacionadas con la variable y antes de la línea 12.

Sentencia *While*

```
1 public class While {
2
3   public static void main(String[] args) {
4     int x = 1;
5     char y = 'a';
6     while (x <= 10) {
7       System.out.print("␣" + x);
8       y = 'a';
9       while (y <= 'c') {
10        System.out.print("␣" + y);
11        y++;
12      }
13      x++;
14    }
15    System.out.println();
16  }
17 }
```

```
1 public class While {
2
3     public static void main(String[] args) {
4         int x = 1;
5         while (x <= 10) {
6             x++;
7         }
8     }
9 }
```

Figura 7.4: Programa Java que contiene una sentencia *while* (arriba) y el resultado de ejecutar el fragmentador con respecto a $\langle 13, x \rangle$

En la figura 7.4, se demuestra la cobertura que el fragmentador realiza sobre las sentencias *while*. Respecto al criterio de fragmentación $\langle 13, x \rangle$, se excluyen todas las instrucciones relacionadas con la variable y antes de la línea 13 (líneas 5, 7, 8, 9, 10, 11, 12). El fragmento que se obtiene es completo y correcto, al componerse únicamente de las instrucciones que afectan a x en la línea 13.

Sentencia *For*

```
1 public class For {
2
3   public static void main(String[] args) {
4     int x = 1;
5
6     //Bucle 1: Contador
7     while (x < 10) {
8       System.out.println(x);
9       x++;
10    }
11
12    //Bucle 2: Sumatorio
13    int suma = 0;
14    int y = 1;
15    while (y < 10) {
16      suma += y;
17      y++;
18    }
19    System.out.println(suma);
20
21    //Bucle 3: Sumatorio
22    int sumatorio = 0;
23    int min = 10;
24    int max = 100;
25    for (int num = min; num <= max; num++) {
26      sumatorio += num;
27    }
28    System.out.println(sumatorio);
29
30    int count = 0;
31    while (count < 10)
32      count++;
33    System.out.println(count);
34  }
35 }
```

```
1 public class For {
2
3     public static void main(String[] args) {
4         int sumatorio = 0;
5         int min = 10;
6         int max = 100;
7         for (int num = min; num <= max; num++) {
8             sumatorio += num;
9         }
10        System.out.println(sumatorio);
11    }
12 }
```

Figura 7.5: Programa Java que contiene una sentencia for (arriba) y el resultado de ejecutar el fragmentador con respecto a $\langle 28, sumatorio \rangle$

En la figura 7.5, se demuestra la cobertura que el fragmentador realiza sobre las sentencias for. Respecto al criterio de fragmentación $\langle 28, sumatorio \rangle$, se incluyen únicamente las instrucciones relacionadas con la variable *sumatorio* antes de la línea 28 (líneas 22, 23, 24, 25, 26, 27). El

fragmento es correcto y completo por esta misma razón.

Sentencia *Foreach*

```
1  public class Foreach {
2
3      public static void main(String[] args) {
4          int x = 1;
5          int y = 2;
6
7          int values[] = new int[]{ 1, 2, 3, 4 };
8
9          for (int value : values) {
10             x += value;
11             y++;
12         }
13
14         System.out.println(x);
15         System.out.println(y);
16     }
17 }
```

```
1  public class Foreach {
2
3      public static void main(String[] args) {
4          int x = 1;
5          int[] values = new int[] { 1, 2, 3, 4 };
6          for (int value : values) {
7              x += value;
8          }
9          System.out.println(x);
10     }
11 }
```

Figura 7.6: Programa Java que contiene una sentencia $foreach$ (arriba) y el resultado de ejecutar el fragmentador con respecto a $\langle 14,x \rangle$

En la figura 7.6, se demuestra la cobertura que el fragmentador realiza sobre las sentencias $foreach$. Respecto al criterio de fragmentación $\langle 14,x \rangle$, se incluyen las instrucciones que afectan a dicha variable en este punto (líneas 4, 7, 9, 10, 12). El fragmento es completo y correcto por dicha razón, pues solo se incluyen las sentencias necesarias.

Sentencia *Switch* y saltos incondicionales

```
public class Switch {
  public static void main(String[] args) {
    int x = Integer.valueOf(args[0]);
    int y = -1;
    switch (x) {
      case 1:
        y = 10;
        break;
      case 2:
        y = 20;
        break;
      case 3:
        y = 30;
        break;
      case 4:
      case 5:
        y = 100;
        break;
      case 6:
        System.err.println("Error");
      case 10:
        y = 0;
    }
    System.out.println(y);
  }
}
```

```
public class Switch {

    public static void main(String[] args) {
        int x = Integer.valueOf(args[0]);
        int y = -1;
        switch(x) {
            case 1:
                y = 10;
                break;
            case 2:
                y = 20;
                break;
            case 3:
                y = 30;
                break;
            case 4:
            case 5:
                y = 100;
                break;
            case 6:
            case 10:
                y = 0;
        }
        System.out.println(y);
    }
}
```

Figura 7.7: Programa Java que contiene una sentencia *switch* (arriba) y varias sentencias de salto incondicional (de tipo *break*) y el resultado de ejecutar el fragmentador con respecto a $\langle 24, y \rangle$

En la figura 7.7, se demuestra la cobertura que el fragmentador realiza sobre las sentencias *switch* y de salto incondicional, como en este caso *break*. Respecto al criterio de fragmentación $\langle 24,y \rangle$, se excluye la única instrucción que no afecta a y (línea 20). El fragmento es completo y correcto por dicha razón, pues solo se incluyen las sentencias necesarias.

Como se ha podido comprobar, el conjunto de sentencias a las que el fragmentador da soporte se ha demostrado mediante ejemplos prácticos, de manera que se ha validado su funcionalidad. No obstante, no se puede decir que no existan fallos en algunos casos concretos, dado que las pruebas no pueden garantizar al cien por cien la ausencia de errores.

Capítulo 8

Trabajo futuro

En este trabajo, se ha implementado un fragmentador basado en el SDG para el lenguaje Java. No obstante, como ya se ha comentado en la sección 3, la parte relacionada con la orientación a objetos ha quedado fuera de la cobertura del lenguaje. En general, los puntos a mejorar son los que se comentan en dicha sección, pero también hay otras mejoras.

Por una parte, es necesario añadir soporte para las funcionalidades del paradigma orientado a objetos. Esto implica aumentar el SDG, formando otro tipo de grafo de dependencias que incluyera soporte para representar clases, interfaces y objetos, además de otros conceptos adicionales que son más complejos como, por ejemplo, la herencia múltiple. Esto también implica dar soporte a constructores de método, los objetos *this* y *super*, a variables no primitivas pasadas como argumento a métodos (de forma que estas sí que son mutables dentro del método), etc.

Para abordar la orientación a objetos se deberán tener en cuenta los trabajos de Malloy, B. A et al.[29], Larsen et al.[30], Zhao J.[32], Liang et al.[31], Kovacs et al.[7], Walkinshaw et al.[26] y Hammer et al.[27].

Por otra parte, ampliar el soporte a características adicionales propias de Java, relativas especialmente a sintaxis, como pueden ser las funciones lambda, referencias a métodos y otra serie de características de ese estilo.

También cabe destacar que en este trabajo las excepciones no se han tenido en cuenta. Las excepciones forman parte de un campo propio de estudio, siendo una parte muy compleja del lenguaje y que afecta en gran medida a la fragmentación. Por esta razón, se ha decidido no incluirlas. Como trabajo futuro sería posible incluir un tratamiento de excepciones para Java.

Para abordar el tratamiento de excepciones se deberán tener en cuenta los trabajos de Allen, M. and Horwitz, S.[39], Galindo Jiménez, C. S.[40] y Galindo Jiménez, C. S. et al.[41].

Adicionalmente, para mejorar el apartado de interacción con el usuario, se podría crear una interfaz gráfica que mostrara el programa original y el fragmento obtenido, con diversos elementos visuales para cambiar opciones, y la posibilidad de visualizar los grafos generados, para un mayor nivel de depuración.

Capítulo 9

Conclusiones

Como conclusión, se puede decir que se han cumplido en gran medida los objetivos establecidos del trabajo. En concreto, se ha conseguido desarrollar un fragmentador estático, describiendo de manera clara cómo se realiza el análisis del código del programa, cómo se construyen las diversas representaciones del código necesarias para realizar la fragmentación (CFG, ACFG, PDG, PPDG, SDG, *Call Graph*, etc.) y finalmente, cómo se obtiene el fragmento final, así como su exportación a fichero, cubriendo así el ciclo completo de una herramienta de estas características.

Por otro lado, el fragmentador ha sido desarrollado siguiendo los principios SOLID de desarrollo de programas orientados a objetos, los cuales ayudan en gran medida a que el código sea mantenible y extensible. De forma, resultará más fácil ampliar la funcionalidad de la herramienta en el futuro, como ampliar la cobertura de características de Java, o incluso el soporte a otro tipo de lenguajes.

Adicionalmente, se ha mostrado un caso de uso de ejemplo de un programa, describiendo el comando concreto mediante el cual se utiliza el fragmentador, mostrando el SDG generado, al cual se le ha aplicado el algoritmo correspondiente de fragmentación para obtener el fragmento deseado, y, finalmente, el fragmento final completo producido por la herramienta desarrollada con éxito.

Además, se han realizado pruebas de cobertura de las características que se habían establecido como objetivo para cubrir del lenguaje Java (en concreto, las sentencias descritas en la sección 3), demostrando que se ha desarrollado una herramienta que es capaz de obtener fragmentos completos y correctos (fragmentos mínimos).

El fragmentador desarrollado es accesible públicamente en el GitHub del grupo MIST de la Universidad Politécnica de Valencia mediante la siguiente url: `https://github.com/mistupv/JavaSDGSlicer`, donde se puede consultar su código y proponer mejoras.

Bibliografía

[1] Weiser, M. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*. 439–449.

[2] Weiser, M. 1984. Program Slicing. *IEEE Transactions on Software Engineering*, 10, 4, 352-357

[3] Bergeretti, J. and Carré, B. 1985. Information-flow and data-flow analysis of While-programs. *ACM Transactions on Programming Languages and Systems 7*, 1, 37–61.

[4] Korel, B. and Laski, J. 1988. Dynamic program slicing. *Information Processing Letters 29*, 3, 155–163

[5] Kumar, S. and Horwitz, S., 2002, April. Better slicing of programs with jumps and switches. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 96-112). Springer, Berlin, Heidelberg.

[6] Horwitz, S., Reps, T. and Binkley, D. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26–60

[7] Kovacs, G., Magyar, F. and Gyimothy T. 1996. Static slicing of Java programs, Tech. Rep. TR-96-108, Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Joesf Attila University

[8] Ranganath, V. P. and Hatcliff, J. 2007. Slicing concurrent java programs using indus and kaveri. Int. J. Softw. Tools Technol. Transf., 9(5):489–504, 2007.

[9] Agrawal, H., Demillo, R. A. and Spafford, E. H. 1993. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616.

[10] Choi, J.-D., Miller, B. P. and Netzer, R. H. B.. 1991. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491– 530.

[11] DeMillo, R. A., Pan, H. and Spafford, E. H. 1996. Critical slicing for software fault localization. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 121–134, New York, NY, USA. ACM.

[12] Fritzson, P., Shahmehri, N., Kamkar, M. and Gyimothy, T. 1992. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322.

[13] Weiser, M. 1983. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters*, 17(3):129–135

[14] Gallagher, K. B. 1990. Using program slicing in software maintenance. PhD thesis, University of Maryland at Baltimore County, Catonsville, MD, USA

[15] Horwitz, S. 1990. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA

[16] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. ACM Trans. Program. Lang. Syst., 11(3):345–387.

[17] Bates, S. and Horwitz, S. 1993. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT sym- posium on Principles of programming languages*, pages 384–396, New York, NY, USA

[18] Gupta, R., Harrold, M. J. and Soffa, M. L. 1992. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenanc*e, pages 299–308, Orlando, FL, USA.

[19] Beck, J. and Eichmann, D. 1993. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engi- neering*, pages 509–518, Los Alamitos, CA, USA.

[20] Daoudi, M., Ouarbya, L., Howroyd, J., Danicic, S., Harman, M., Fox, C. and Ward, M. P. 2002. Consus: A scalable approach to conditioned slicing. IEEE Computer Society. In *WCRE '02: Pro- ceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 109, Washington, DC, USA.

[21] Ferrante, J., Ottenstein, K., and Warren, J. 1984. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3, 319–349.

[22] Smith, N., Van Bruggen, D. and Tomassetti, F. 2018. *JavaParser: Visited. Analyse, transform and generate your Java code base*, Leanpub

[23] Silva, J. 2012. A vocabulary of program slicing-based techniques. ACM Comput. Surv. 44, 3, Article 12 (June 2012), 41 pages.

[24] Jayaraman, G., Ranganath, V. P and Hatcliff, J. 2005. Kaveri: Delivering the Indus Java Program Slicer to Eclipse. In *Proceedings of the 8th international conference*, 3442, 269-272.

[25] Giffhorn, D. and Hammer, C. 2008. Precise Analysis of Java Programs using JOANA. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 267-268

[26] Walkinshaw, N., Roper, M. and Wood, M. 2003. The Java System Dependence Graph. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*. 0-7695-2005-7/03.

[27] Hammer, C. and Snelting, G. 2004. An Improved Slicer for Java. *ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* 17-22

[28] Alokush, B., Abdallah, M., Alrifaee, M. and Salah, M. 2018. A proposed Java Static Slicing Approach. In *Indonesian Journal of Electrical Engineering and Computer Science*, 11, 308-317.

[29] Malloy, B. A., McGregor, J. D., Krishnaswamy, A., and Medlkonda, M. 1994. An extensible program representation for object-oriented software. ACM SIGPLAN Notices, 29(12):38–47.

[30] L. Larsent and M. J. Harrold. 1996. "Slicing object-oriented software" Proceedings of the 18th International Conference on Software Engineering, pp. 495-505.

[31] D. Liang, and M.J. Harrold. 1998. "Slicing objects using system dependence graphs", Proceedings., International Conference on Software Maintenance , pp. 358-367.

[32] Zhao, J. 2000. Dependence analysis of java bytecode. In *Proceedings of the 24th IEEE Annual International Computer Software and Applications Conference*, pages 486–491.

[33] Schiffel, U., Süßkraut, M, Fetzer, C and Schmitt, A. 2010. Slice Your Bug: Debugging Error Detection Mechanisms Using Error Injection Slicing. In *2010 European Dependable Computing Conference (EDCC)*, Valencia, 13-22.

[34] Reps, T. and Turnidge, T. 1996. Program specialization via program slicing. *Springer Berlin Heidelberg*, Berlin, 409-429.

[35] CodeSonar SAST for Java. `https://www.grammatech.com/codesonar-sast-java` [accedido en septiembre 2020]

[36] T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php/Main_Page` [accedido en septiembre 2020]

[37] Michail, D., Kinable, J., Naveh, B. and Sichi, J.V., 2019. JGraphT–A Java library for graph data structures and algorithms.

[38] JGraphT. `https://jgrapht.org/` [accedido en febrero 2020]

[39] Allen, M. and Horwitz, S., 2003. Slicing java programs that throw and catch exceptions. *ACM SIGPLAN Notices, 38(10)*, pp.44-54.

[40] Galindo Jiménez, C. S. 2019. Fragmentación de programas con excepciones. `http://hdl.handle.net/10251/136752`

[41] Galindo Jiménez, C. S., Pérez Rubio, S. and Silva, J. 2020. Slicing Unconditional Jumps with Unnecessary Control Dependencies. In *30th International Symposium on Logic-Based Program Synthesis and Transformation.*