CALLCLUSTER: EXTRACCIÓN, ANÁLISIS Y VISUALIZACIÓN DE CALLGRAPHS

TRABAJO PROFESIONAL

Autor:
SBRUZZI, JOSÉ IGNACIO - Padrón 97452

jose.sbru@gmail.com

Tutor:
PROF. DR. MARIANO MÉNDEZ

Facultad de Ingeniería - Universidad de Buenos Aires

Mayo 2021

Índice

1. Introducción **4**
1.1. Motivación . . . 4
1.2. Estado del arte . . . 4
1.2.1. Moose . . . 5
1.2.2. Sourcetrail . . . 6
1.3. Ideas iniciales y antecedentes . . . 6

2. Introducción teórica **8**
2.1. Grafo . . . 8
2.2. Función . . . 8
2.3. Callgraph . . . 10
2.4. Estructuras de control . . . 10
2.5. Control Flow Graph . . . 11
2.6. Complejidad ciclomática McCabe . . . 13
2.7. AST . . . 15

3. Diseño e implementación **17**
3.1. Primer Prototipo . . . 17
3.1.1. Caso de estudio elegido . . . 20
3.1.2. Aprendizajes . . . 22
3.2. Interfaz de usuario . . . 22
3.3. Aproximación del callgraph en C . . . 26
3.3.1. El preprocesador . . . 26
3.3.2. Apuntadores a funciones . . . 27
3.4. Aproximación del callgraph en C# . . . 28
3.4.1. Herencia . . . 28
3.4.2. Características funcionales . . . 29
3.5. Descripción de las métricas recolectadas por los extractores . 30
3.5.1. Complejidad ciclomática Basili . . . 31
3.5.2. Adaptación del cálculo de la complejidad ciclomática Basili para C . . . 31
3.5.3. Adaptación del cálculo de la comlejidad ciclomática Basili para C# . . . 32
3.5.4. Implementación del cálculo de la complejidad ciclomática Basili utilizando `libclang` . . . 32
3.5.5. Cantidad de líneas de código . . . 33
3.5.6. Cantidad de instrucciones contenidas en la función . . 33
3.5.7. Complejidad ciclomática McCabe . . . 34

3.6. Herramientas de análisis evaluadas 34
3.6.1. ANTLR 34
3.6.2. Roslyn 34
3.6.3. GCC 35
3.6.4. Desarrollar un plugin de GCC 35
3.6.5. LLVM 35
3.7. Análisis de librerías de visualización de grafos 35
3.7.1. Relevamiento de librerías de visualización de grafos (primera etapa) 35
3.7.2. Resultados del relevamiento de librerías de visualización de grafos 38
3.8. Arquitectura del sistema 40
3.8.1. Principio guía 40
3.8.2. Comunicación entre componentes: `analysis.json` . . 41
3.8.3. Arquitectura de los extractores `callcluster-dotnet` y `callcluster-clang` 41
3.8.4. Arquitectura de `callcluster-visu` 42

4. Casos de Estudio 44
4.1. LibUV 44
4.1.1. Treemaps 44
4.1.2. Callgraphs 47
4.1.3. Conclusiones 50
4.2. Namespace System.Net.Http del runtime de .Net 51
4.2.1. Treemaps 51
4.2.2. Callgraphs 53
4.2.3. Análisis del namespace `HPack`, incluido en `System.Net.Http` 56
4.2.4. Conclusión 58

5. Manual de usuario 59
5.1. Extracción del callgraph de programas C 59
5.1.1. Requisitos 59
5.1.2. Compilación 59
5.1.3. Extracción del callgraph 59
5.1.4. Opciones recibidas por callclusterClang 60
5.1.5. Ejemplo (php) 60
5.1.6. Ejemplo (redis) 60
5.2. Extracción del callgraph de programas C 61
5.2.1. Instalación y ejecución 61
5.2.2. Ejemplo (DNN.Platform) 61

5.2.3. Ejemplo (bc-csharp) 61
5.3. Uso del visualizador callcluster-visu 61
5.3.1. Requisitos 61
5.3.2. Instalación y ejecución 62
5.3.3. Conceptos básicos (modelo mental) 62
5.3.4. Descripción general de la interfaz 65
5.3.5. Uso del visualizador 66
5.4. Especificación del formato de callgraphs para callcluster `analysis.json` 70
5.4.1. Archivo emitido por los extractores 70
5.4.2. diccionario Comunidad 71
5.4.3. Extensibilidad 71

6. Conclusión **72**
6.1. Trabajos futuros 73

1. Introducción

Se desarrolló una herramienta útil para juzgar y entender la estructura de proyectos de software en diversos niveles de abstracción, a partir de la visualización de los mismos como un grafo, y la aplicación de técnicas de Ciencia de Datos.

1.1. Motivación

Existen muy pocas herramientas de análisis de código fuente, y aún menos que sean de código abierto. Las pocas que existen (ver sección Estado del arte) consisten en desarrollos pequeños y son poco versátiles. La mayoría están atadas a algún IDE. Ninguna permite el análisis del callgraph de proyectos grandes. Las herramientas de análisis de código existentes actualmente que se aplican en la industria son cerradas y pagas. Entre las herramientas que existen que permiten visualizar grafos de dependencias o callgraphs, ninguna permite generar ni visualizar una modularización automática.

La comprensión de la estructura del programa es un conocimiento importante para los programadores. Una interfaz basada en texto no revela fácilmente la estructura de alto nivel del mismo, con lo cual una visualización como la que se propone es muy requerida. Poder ver y mejorar la arquitectura de un sistema de software puede mejorar de forma muy significativa su legibilidad, modificabilidad y mantenibilidad.

1.2. Estado del arte

La mayoría de las herramientas de análisis estático de código son pagas, con licencias pensadas para empresas grandes. Las herramientas de código abierto que se encuentran disponibles se centran principalmente en la detección automática de bugs o problemas de seguridad (DevSkim, RIPS, PMD, facebook infer, coccinelle, CPAchecker, Cppcheck, Frama-C, Sparse, Clang Static Analyzer), o en análisis que verifican el estilo visual del código. Se listan a continuación herramientas de código abierto que permiten análisis y visualización del código:

- Moose
- Proyecto Bauhaus
- ConQAT
- SoftVis3D

- Sourcetrail

Proyecto Bauhaus y ConQAT son herramientas que inicialmente eran opensource pero gradualmente se dejaron de mantener. SoftVis3D es un plugin para SonarQube que permite visualizar el sistema como una ciudad, no exhibe el callgraph ni permite generar una modularización automática.

1.2.1. Moose

La plataforma Moose (ver figura 1) está basada en pharo, y es un proyecto de investigación compuesto por muchos equipos de investigación ubicados en universidades de distintos países. El proyecto inició en 1999, incluye software que tiene la función de extraer reportes que enumeran todas las entidades de software que tiene un sistema. El resto de la plataforma moose permite diseñar visualizaciones de estos reportes. El sistema se ejecuta dentro de pharo, y es el usuario el que debe diseñar las consultas y las visualizaciones de esas consultas. Este enfoque "abierto"respecto de las visualizaciones es evitado en Callcluster con el objetivo de facilitar la comunicación entre los usuarios.

Moose incluye extractores para Java y C#, estos extractores generan un archivo que incluye información estructural (es decir, la organización del código en clases, namespaces, etcétera), pero no incluye un callgraph aproximado.

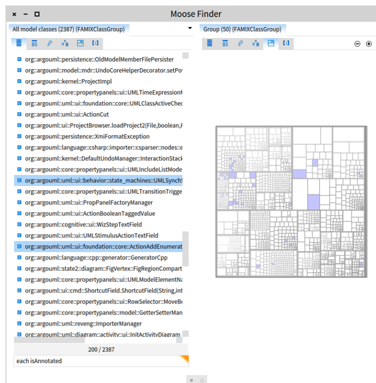

Figura 1: Captura de Moose

1.2.2. Sourcetrail

Sourcetrail (ver figura 2) es un proyecto de código abierto financiado a través de donaciones, que incluye la posibilidad de navegar el callgraph y observar el código relevante simultáneamente. Permite leer código en C, C++, Java y Python. Si bien tiene la información del callgraph, los extractores están acoplados muy fuertemente con el visor, conformando un proyecto que tiende a ser monolítico: no puede ser extendido fácilmente, ni es sencillo usar sus componentes en CallCluster.

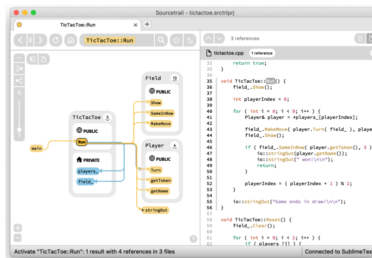

Figura 2: Captura de Sourcetrail

1.3. Ideas iniciales y antecedentes

La experiencia que inspiró callcluster fue trabajar en un proyecto monolítico grande cuyos componentes estaban muy fuertemente acoplados. En este contexto, la lectura de una función consiste sobre todo en entender sus llamadas entrantes y salientes; el nombre o la clase a la que pertenece una función no aportan significado. Así, para entender el comportamiento de un fragmento de código, el desarrollador debe construir un çallgraph mental". En este tipo de situaciones, sería muy útil poder inspeccionarlo visualmente en vez de confiar esta tarea a la imaginación del desarrollador.

Otro punto que inspiró callcluster es la dificultad de entender el funcionamiento de una entidad de software a partir de su representación textual: siempre que se documenta software, se incluyen diagramas. Sin embargo, los diagramas se tienen que mantener actualizados, lo cual implica un costo. Así, es necesario generar diagramas automáticamente para facilitar el mantenimiento del software ya que hacerlo manualmente no es óptimo (y no tener diagramas es necesariamente peor que tenerlos).

La representación textual en muchos casos no es la ideal. Es buena para describir algoritmos y además es versátil, capaz de expresar conceptos muy diversos. Sin embargo, la representación textual no expone de manera obvia

cómo se relaciona el fragmento de código que uno lee, con el resto del sistema; que uno no lee. Paradójicamente, para una persona que lee el código en vez de escribirlo, lo más importante es aquello que la representación textual esconde.

Callcluster extrae dos características de la estructura de un programa:

- Su callgraph aproximado
- Una estructura jerárquica de las funciones del callgraph

Esta es la información necesaria para visualizar claramente qué módulos conforman un programa, y cuales son las interfaces entre los mismos.

Si el sistema inspeccionado tiene una mala modularización, extraer estos datos no es suficiente para entender su estructura ya que la modularización inducida por el callgraph puede ser más informativa que la propia del programa. Detectar los módulos inducidos por un callgraph implica encontrar conjuntos de funciones que están muy relacionadas entre sí pero poco relacionadas a las demás. Este problema puede reducirse al de descubrimiento de comunidades en redes sociales: las funciones se tratan como si fueran individuos y las llamadas entre funciones, como si fueran vínculos entre ellos. Esta reducción del problema a uno más común permite aprovechar la extensa bibliografía científica disponible.

Esta "traducción"tiene otras consecuencias interesantes. El hecho de que los algoritmos para redes sociales requieren optimizar una función que cuantifica cuán buena es una cierta división en comunidades implica que Callcluster requerirá cuantificar la calidad del diseño. La elaboración de una métrica que refleje la naturaleza del problema no se incluye en el alcance inicial del Trabajo Práctico Profesional: se utilizarán las ya existentes.

2. Introducción teórica

2.1. Grafo

Según Grimaldi [1] :

> Sea V un conjunto finito no vacío, y sea $E \subseteq V \times V$. El par (V, E) es llamado grafo dirigido de V, donde V es el conjunto de vértices del grafo y E es el conjunto de aristas del grafo.

Los grafos son conceptos matemáticos que modelan una multitud de ideas y objetos. La figura 3 muestra una representación visual de un grafo.

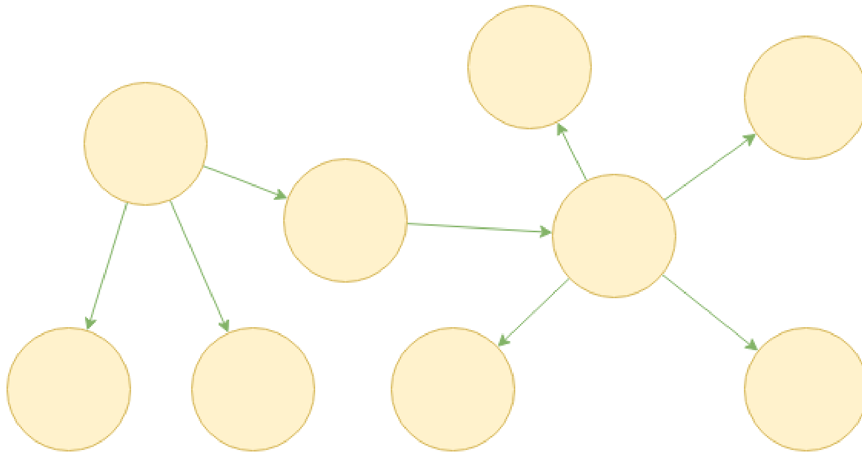

Figura 3: Un grafo. E es el conjunto de aristas verde, V es el conjunto de vértices naranjas

2.2. Función

Según Kernighan y Ritchie [2]:

> Las funciones dividen tareas de cómputo grandes en partes más pequeñas, y permiten que los desarrolladores construyan aprovechando el trabajo de otros, en vez de empezar siempre desde cero.

Entender el concepto de función como un çonjunto de tareas.^{es} propio del paradigma imperativo, donde el programador le indica a la computadora lo que tiene que hacer, es decir, le da órdenes. Existen secuencias de órdenes que

se suelen repetir en distintos lugares de un programa, pero el desarrollador, en vez de escribir las mismas sucesiones de tareas muchas veces, diseña una función, y en vez de repetir su contenido muchas veces, la invoca en distintos lugares del programa.

El concepto de función, inicialmente llamado *subrutina* o *procedimiento* es uno de los pilares de la programación estructurada.

Algoritmo 1: Poner la mesa sin funciones

```
Resultado: La mesa lista para comer
Poner plato para Aylén
Poner cubiertos para Aylén
Poner vaso para Aylén
Poner plato para Benjamín
Poner cubiertos para Benjamín
Poner vaso para Benjamín
Poner plato para Carlos
Poner cubiertos para Carlos
Poner vaso para la Carlos
Poner plato para Daiana
Poner cubiertos para Daiana
Poner vaso para Daiana
```

Algoritmo 2: Poner la mesa con funciones

```
Resultado: La mesa lista para comer
Función Poner la mesa para P:
    Poner plato para P
    Poner cubiertos para P
    Poner vaso para P
fin
Función Poner la mesa:
    Poner la mesa para Ayelén
    Poner la mesa para Benjamín
    Poner la mesa para Carlos
    Poner la mesa para Daiana
fin
```

El programa 2 es más claro y permite gestionar mejor la complejidad, al contrario del programa 1, que es más difícil de entender.

2.3. Callgraph

El concepto que en el presente Trabajo Práctico Profesional se llama *Callgraph* es descripto con precisión en [3], que lo llama *Procedure Call Graph*, o *Grafo de llamadas entre procedimientos*. Consiste en un grafo que tiene un vértice por cada función. Una arista vincula un vértice a a otro vértice b sí y sólo si la función correspondiente a a llama a la función correspondiente a b. En la figura 4 puede observarse un ejemplo.

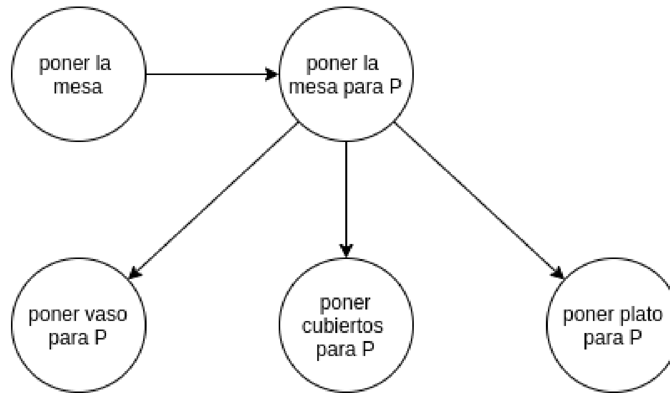

Figura 4: Callgraph correspondiente al algoritmo 2

2.4. Estructuras de control

El teorema Böhm-Jacopini[4] [5] establece que cualquier función computable puede expresarse utilizando tres estructuras de control:

- Secuencia
- Selección
- Repetición

Este teorema conforma la base teórica de la programación estructurada, en la cual el flujo de control del programa no salta aleatoriamente, sino que respeta el flujo inducido por estas estructuras. Los algoritmos 2 y 1 son ejemplos de funciones enteramente secuenciales, mientras que los programas

3 y 4 ejemplifican las estructuras de selección y repetición, respectivamente.

Algoritmo 3: Ejemplo de estructuras de selección

```
Resultado: Llegar a Facultad de Ingeniería de la UBA desde
           Avellaneda
si Hay calles cortadas por manifestaciones entonces
|  Ir a la estación de tren.
|  Tomar el tren.
|  Tomar el subte.
en otro caso
|  Ir a la parada del colectivo
|  si Hay que llegar en menos de una hora entonces
|  |  Tomar la línea 159.
|  en otro caso
|  |  si Ya pasaron las 8:00 A.M. entonces
|  |  |  Tomar la línea 22 ó la línea 159.
|  |  en otro caso
|  |  |  Tomar la línea 22.
|  |  fin
|  fin
fin
```

Algoritmo 4: Ejemplo de estructuras de repetición anidadas.

```
Resultado: Una partida a 30 puntos
mientras Ningún equipo alcanzó los 30 puntos hacer
|  para cada jugador J hacer
|  |  jugada del jugador J
|  fin
|  Anotar puntajes.
fin
```

2.5. Control Flow Graph

El concepto de *Control Flow Graph* (Grafo de control del flujo) se definió originalmente en [6]. Un *Control Flow Graph* es un grafo cuyos vértices son *bloques básicos*: tareas que siempre se ejecutan en orden secuencial. Existe un arista del vértice a al vértice b si y sólo si el flujo puede pasar del bloque básico a al bloque básico b. En las figuras 7, 5 y 6 se muestran los *Control Flow Graph* de los algoritmos 1, 4 y 3, a modo de ejemplo.

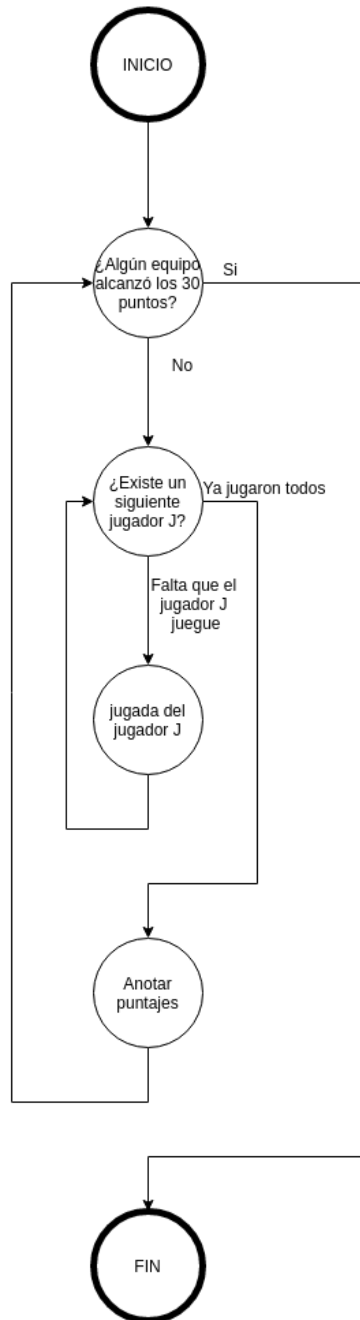

Figura 5: Control Flow Graph correspondiente al programa 4. La complejidad ciclomática de este programa es 2.

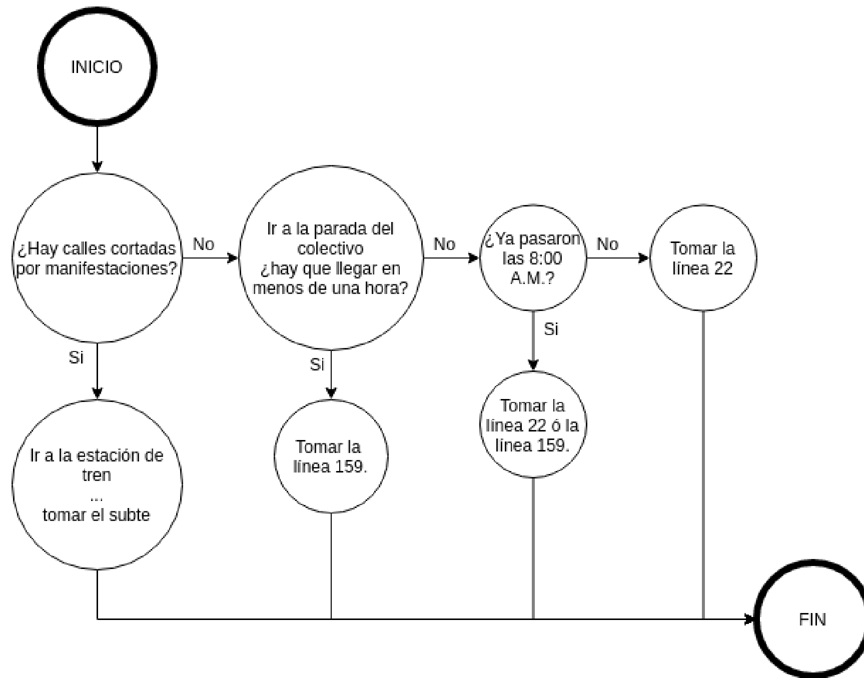

Figura 6: Control Flow Graph correspondiente al programa 3. La pregunta ¿Hay que llegar en menos de una hora? y la tarea Ir a la parada del colectivo están en un único bloque básico, el cual tiene dos salidas. La complejidad ciclomática de este programa es 3.

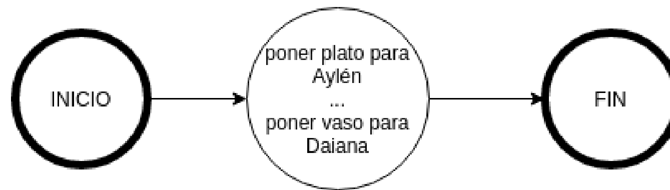

Figura 7: Control Flow Graph para el programa 1. Tiene un único bloque básico porque el programa consiste en una lista secuencial de tareas. La complejidad ciclomática de este programa es 1.

2.6. Complejidad ciclomática McCabe

La idea de evaluar funciones utilizando la complejidad ciclomática de su Control Flow Graph fue introducida por McCabe en [7]. La complejidad ciclomática $C(G)$ de un grafo $G = (V, E)$ se define como:

$$C(G) = |E| - |V| + p(G)$$

Donde $|E|$ es la cantidad de aristas del grafo G, $|V|$ es la cantidad de vértices del grafo G y $p(G)$ es la cantidad de componentes conexos del grafo G. Es importante tener en cuenta que el Control Flow Graph debe añadirse un arista que va de FIN a $INICIO$ para calcular su complejidad.

Aplicar esta métrica a control flow graphs implica contar la cantidad de estructuras de control que el mismo contiene (es decir, la cantidad de bloques básicos que tienen dos aristas salientes, o la cantidad de decisiones que toma un fragmento de código).

La figura 8 ilustra el cálculo de la complejidad ciclomática.

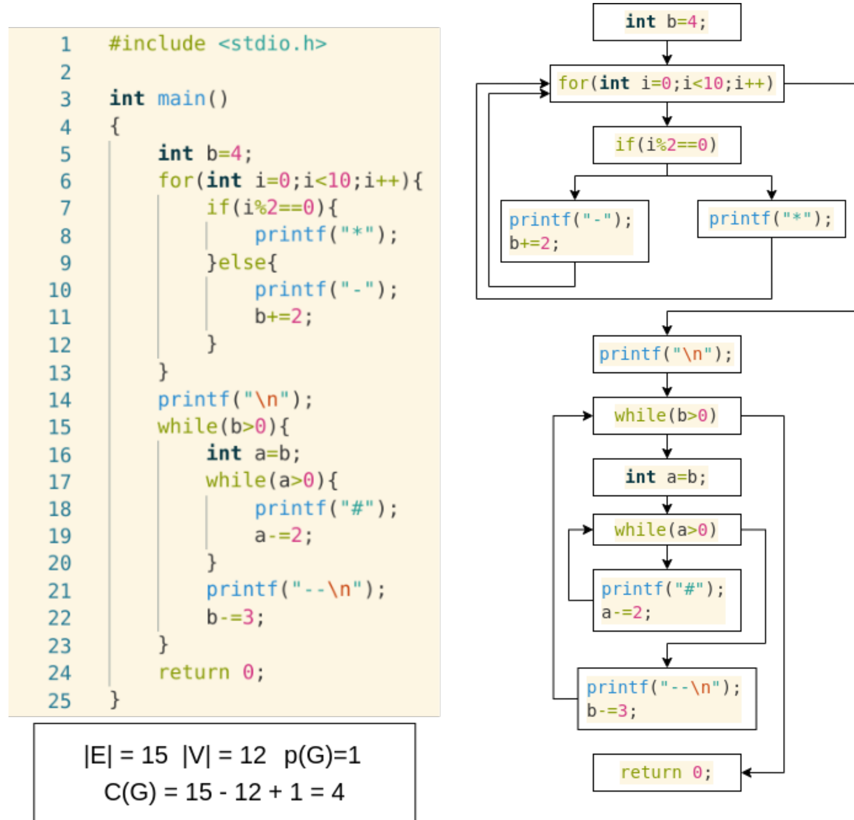

Figura 8: Código fuente `C`, *callgraph* correspondiente y análisis del mismo para encontrar la complejidad ciclomática del fragmento de código.

Cuadro 1: Descripción de la Complejidad de Código Fuente Según su Complejidad Ciclomática. Extraído de [8].

Complejidad ciclomática	Riesgo
1-10	Módulo simple con muy poco riesgo
11-20	Módulo medianamente complejo con riesgo moderado
21-50	Módulo complejo con alto riesgo
51 o mayor	Programa de alto riesgo y altamente inestable

La complejidad ciclomática mide una característica negativa de un fragmento de código: cuán grande es y cuán posible es que contenga problemas, o cambie. Así, un módulo con mayor complejidad ciclomática es más riesgoso. La tabla 1 detalla el significado de distintos rangos de Complejidad Ciclomática.

2.7. AST

El concepto de *Abtrsact Syntax Tree* (abreviado como AST y traducido textualmente como *árbol de sintaxis abstracta*) es utilizado para describir las estructuras concretas de un lenguaje formal. Los lenguajes formales son descriptos por reglas que indican cómo deben ordenarse los términos que las componen. Estas reglas pueden anidarse y pensarse como un árbol. Se incluyen ejemplos del concepto sobre un fragmento de código C (figura 10) y un fragmento de una "formalización"del español (figura 11).

```
{
  int i=0;
  if(i>0) {
    i=3;
    i++;
  }
}
```

Figura 9: Código C cuyo AST se muestra en 10

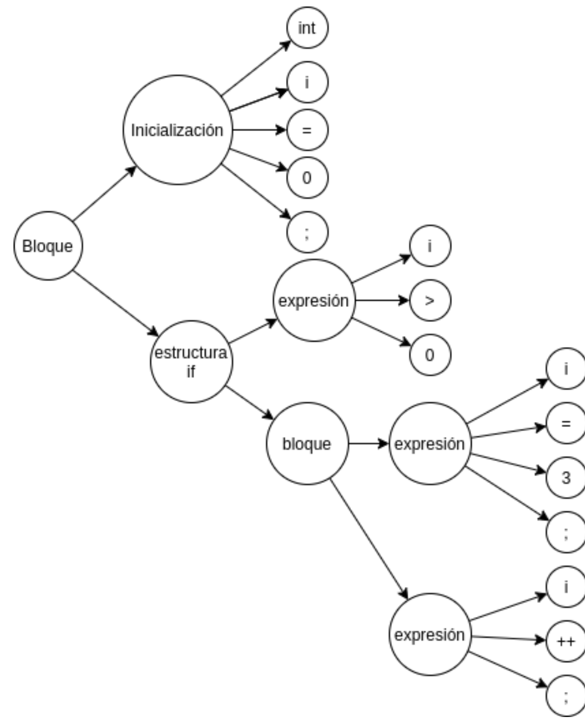

Figura 10: Árbol de sintaxis abstracta correspondiente al fragmento de código C 9

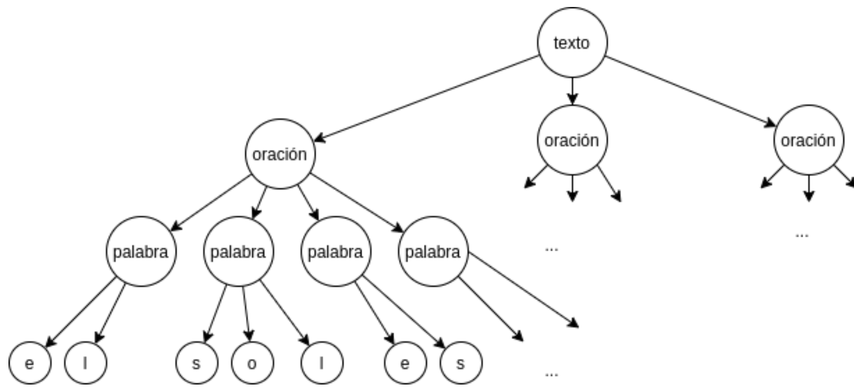

Figura 11: Árbol de sintaxis abstracta correspondiente al fragmento de español 'El sol es amarillo. El pasto es verde. El cielo es azul.'

3. Diseño e implementación

3.1. Primer Prototipo

Para verificar la feasibilidad y entender los riesgos del proyecto, se elaboró un prototipo que extraía el callgraph de un programa C#, aprovechando Roslyn. Roslyn es un compilador de código abierto de C# y VisualBasic. Está muy bien documentado ya que Microsoft alienta su uso para el desarrollo de plugins de Visual Studio.

Se generaron visualizaciones de cuatro proyectos de código abierto que github listaba como populares, escritos en C#:

- ApplicationInspector (figura 13)
- DNN Platform (figura 15)
- BouncyCastle-csharp (figura 14)
- ScreenToGif (figura 12)

  Este primer prototipo extraía únicamente los nombres de los métodos, simplemente transformaba las características de interés del AST a un archivo json. El archivo json era entonces tomado por un front-end web, que generaba el callgraph a partir de los nombres de los métodos y lo dibujaba en pantalla.

Figura 12: Callgraph del proyecto ScreenToGif

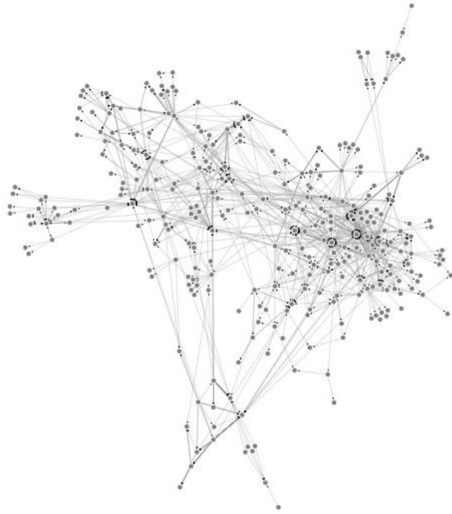

Figura 13: Callgraph del proyecto ApplicationInspector

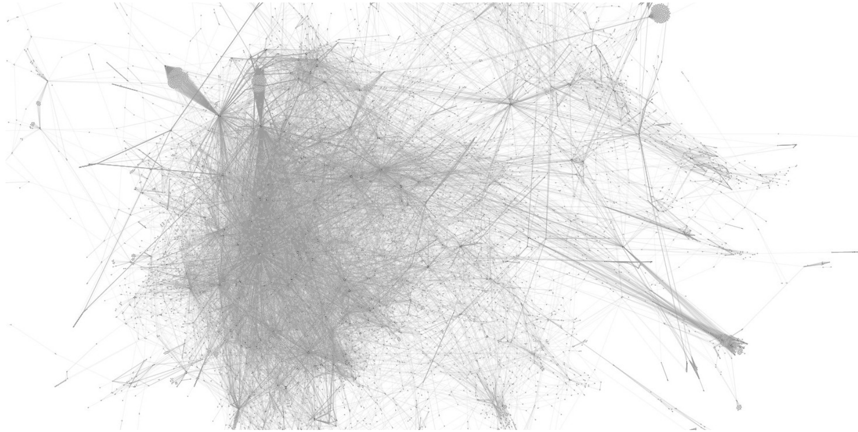

Figura 14: Callgraph del proyecto BouncyCastle-csharp

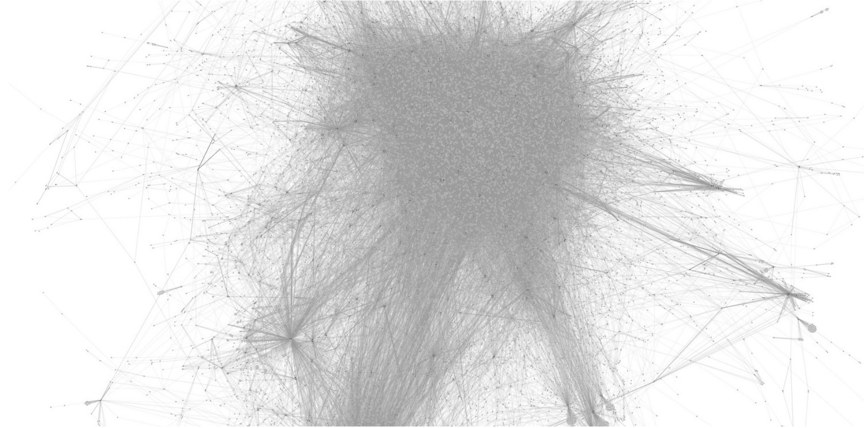

Figura 15: Callgraph del proyecto DNN Platform

3.1.1. Caso de estudio elegido

Debido a su menor tamaño, se investigó con mayor profundidad el callgraph de Application Inspector, y se generó una visualización coloreada por namespace (figura 16).

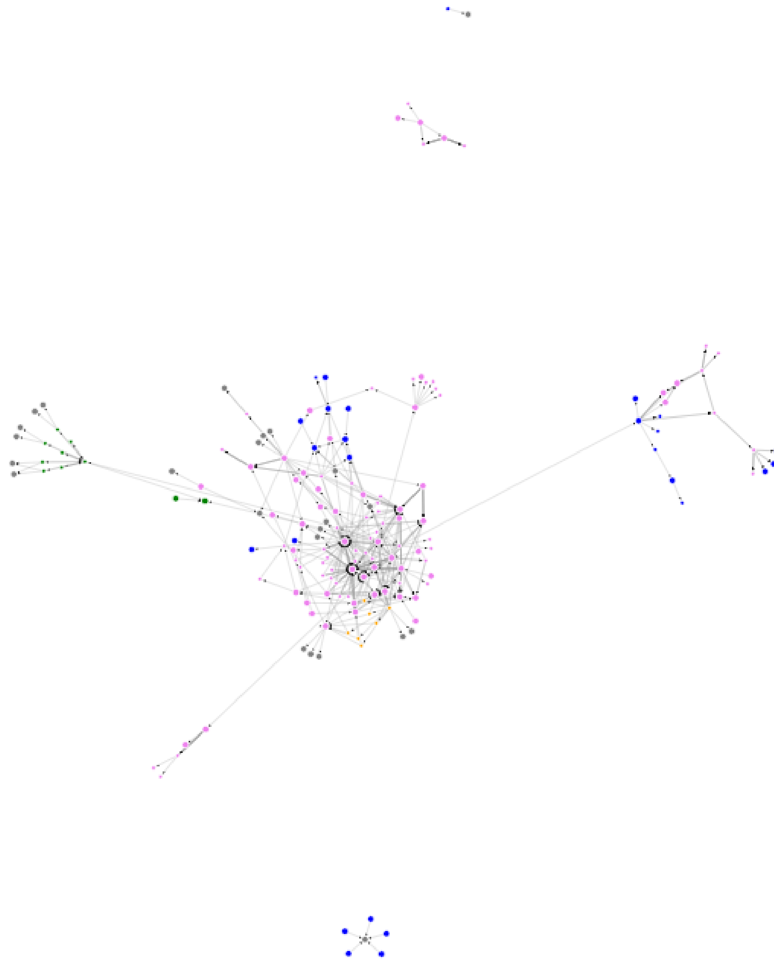

Figura 16: Callgraph de ApplicationInspector coloreado por namespace.

Esta visualización permite juzgar inmediatamente cuán buena es la modularización del módulo verde, y además expone la relación entre el módulo azul y el rosa, y las distintas partes del módulo azul. El grafo es interactivo y dinámico, permitiendo leves modificaciones para visualizar mejor los puntos de entrada de cada módulo. Además, al colocar el mouse sobre los vértices muestra el nombre y la ubicación de la función correspondiente

3.1.2. Aprendizajes

A partir de este prototipo se descubrió lo siguiente:

- No es trivial detectar la función a la cual hace referencia un nombre. Este problema se le debe delegar a la herramienta de extracción, que debe ser responsable de realizar un análisis semántico del código, no sólo sintáctico.
- Tampoco es trivial la visualización. Para proyectos grandes, ver el callgraph entero brinda poca información: por ejemplo, en el caso de las figuras 14 y 15, no es posible reconocer estructuras, pero en las figuras 12 y 13, si. La visualización de grafos grandes implica, además, una dificultad técnica: los algoritmos que disponen el grafo en pantalla suelen tener complejidad superlineal, y los cómputos para "dibujar.esos grafos en la pantalla son de complejidad lineal. Deben evitarse los grafos grandes.
- Es necesario algún análisis sobre el callgraph para poder extraer información valiosa (todavía no se había definido que se iba a utilizar detección de comunidades).

3.2. Interfaz de usuario

Para determinar qué conceptos iban a exponerse al usuario y de qué manera, se creó un modelo de dominio propio de una herramienta que permite generar visualizaciones totalmente personalizadas (figura 17). El modelo resultante resultó ser demasiado grande y, como consecuencia, demasiado difícil de comunicar. Dado que uno de los objetivos de CallCluster es brindar una vista más clara de un sistema analizado, se decidió exponer al usuario un modelo más simple y más fácil de entender pero menos poderoso, y menos personalizable.

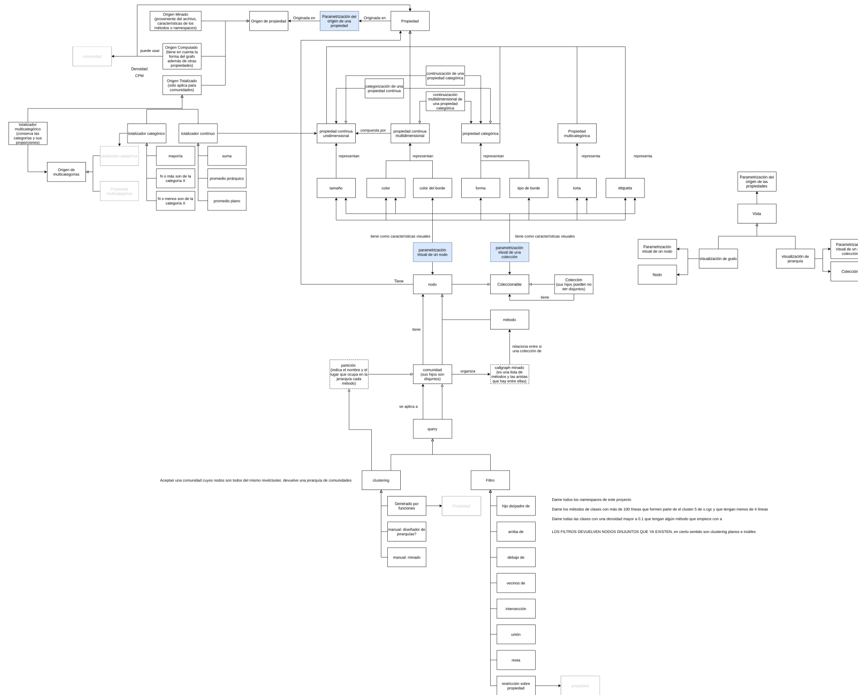

Figura 17: Modelo de dominio

El modelo expuesto al usuario está compuesto por los siguientes conceptos:

- Función
- Callgraph
- Comunidad: es un conjunto de comunidades y/o funciones disjunto, que puede ser:
  - La estructura jerárquica en que los programadores organizaron las funciones
  - El resultado de la ejecución de un algoritmo de clustering jerárquico
  - Otra estructura jerárquica que el usuario genere al usar callcluster.
- Visualización: una visualización creada por el usuario.

  A partir de un archivo generado por los extractores, CallCluster generará un archivo de análisis, que estará compuesto por:

- Una colección de funciones (extraída)
- Un callgraph completo (extraído)
- Una comunidad extraída
- Una colección de comunidades
- Una colección de visualizaciones

  A partir de este archivo el usuario puede modificar, exportar o importar visualizaciones o comunidades. El algoritmo de detección de comunidades agrega la organización jerárquica descubierta a la colección de comunidades, y las visualizaciones permiten çruzarïnformación de hasta dos de ellas.

  La decisión de diseño de interfaces más importante realizada fue la simplificación de la construcción de las visualizaciones: En vez de brindarle al usuario muchas opciones respecto de qué información representar y cómo, el usuario deberá escoger de entre una puñado de visualizaciones disponibles (ver prototipos 18 y 19), y podrá realizar pocas parametrizaciones sobre los mismos. De esta manera no solamente se ocultan detalles poco importantes del modelo sino que además se establece un "lenguaje visual"que facilita la comunicación entre los usuarios de CallCluster.

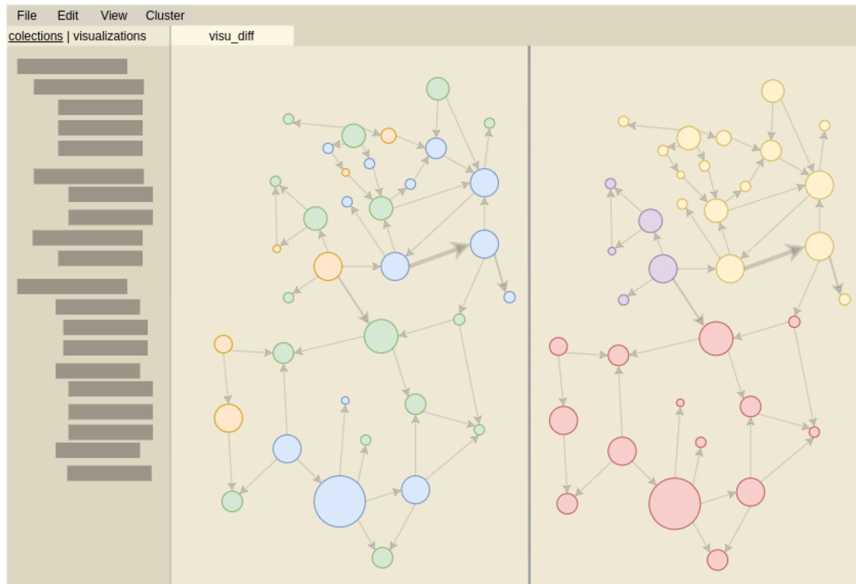

Figura 18: Interfaz de usuario

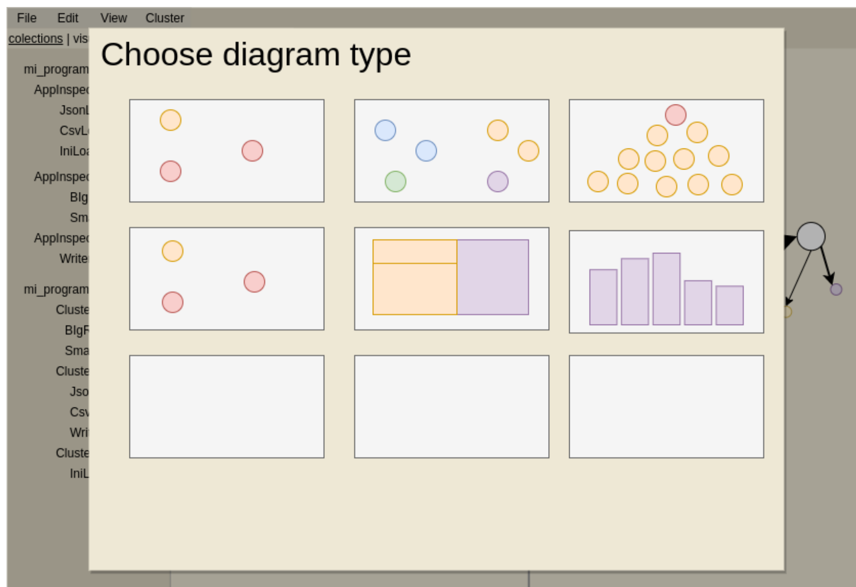

Figura 19: Selección de visualización

3.3. Aproximación del callgraph en C

3.3.1. El preprocesador

El preprocesador agrega la dificultad de que no es posible definir únicamente inspeccionando el programa cuál será el código que va a ser compilado. El uso de preprocesador incluso introduce lo que podría interpretarse como errores de sintaxis. La solución a este problema fue utilizar una herramienta que inspecciona el proceso de compilación y extrae la configuración del preprocesador, del compilador y del enlazador (`linker`).

No es posible inspeccionar todos los programas posibles sino únicamente aquellos que son resultado de una configuración del precompilador. En el caso de un programa con diversas configuraciones posibles (por ejemplo, uno que se compila distinto para distintas plataformas), sólo se puede analizar una de ellas a la vez. Esta problemática se ejemplifica en la figura 21.

```
void foo(){
    bar();
    baz();
#if defined(CREDIT)
    credit();
#else
    debit();
#endif
}
```

Figura 20: Código `C` con directivas de precompilación.

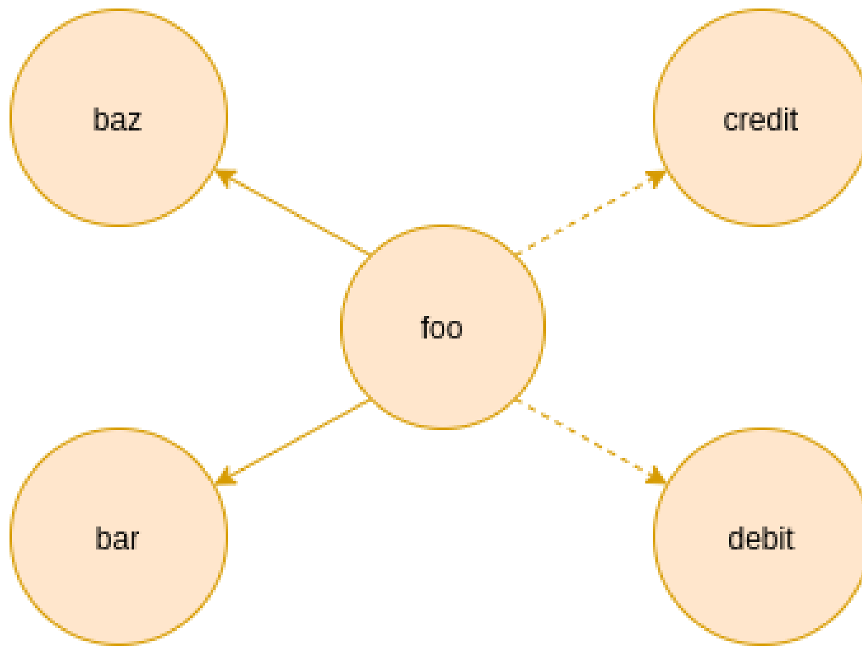

Figura 21: Callgraph del ejemplo 20. No es posible saber cuál de las dos aristas punteadas pertenece al programa a partir del código: es saber si el flag `CREDIT` está definido al compilar.

3.3.2. Apuntadores a funciones

En el lenguaje C, es posible pasar una función como argumento a otra. Calcular a qué función corresponde realmente una que fue pasada como argumento es complejo e implica aproximar la ejecución del programa. Callcluster, en este tipo de situaciones, toma cualquier referencia a una función dentro de otra como una invocación. Así, para el caso de C, se aproxima el callgraph por medio del grafo de dependencias entre funciones. La figura 23 expone un ejemplo de esta problemática.

```
void bar(){}
void baz(){}

void foo(){
   void (*fun_ptr)(void) = &bar;
   if(calcular()){
      fun_ptr = &baz;
   }
```

```
    execute(fun_ptr);
}
```

Figura 22: Código `C` con apuntadores a funciones.

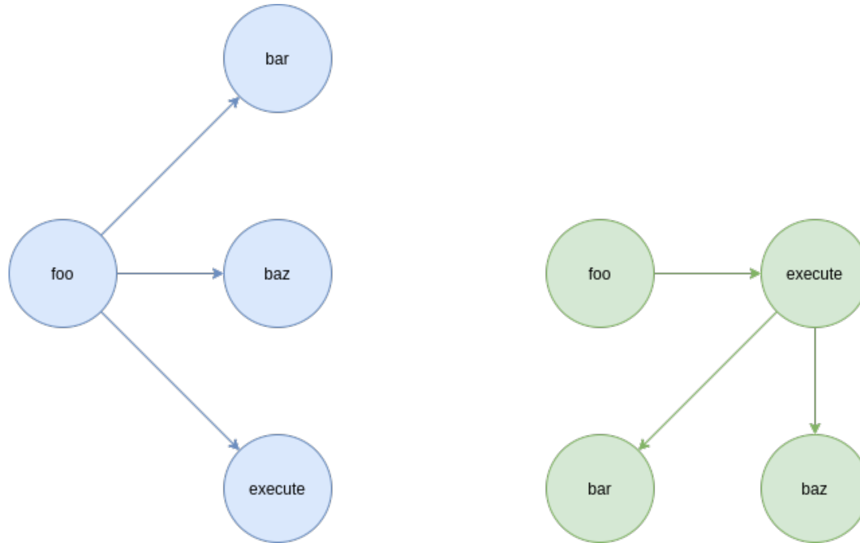

Figura 23: Aproximación del callgraph para el fragmento de código 22. Callcluster calcula el grafo de la izquierda. El grafo de la derecha es una aproximación más fiel al código y al concepto de callgraph.

3.4. Aproximación del callgraph en C#

3.4.1. Herencia

Una dificultad al extraer el callgraph de métodos C# es que, todas las invocaciones de métodos no estáticos son polimórficas. Lo que hace CallCluster es conectar todos los métodos conocidos que podrían ser llamados. Esto brinda una interpretación poco común del diseño, ya que ignora el ocultamiento que introduce el uso de interfaces o la herencia: el criterio es graficar la información que el programador puede averiguar (porción izquierda de la figura 25), no lo que está escrito en el código (porción derecha de la misma figura).

```
class Vehiculo { ... }
class Rodado extends Vehiculo { ... }
```

```
class Auto extends Rodado { ... }
class Motocicleta extends Rodado { ... }

void acelerarRodado(){
    Rodado rodado = new Rodado();
    rodado.Acelerar();
}
```

Figura 24: Código `C#` con herencia.

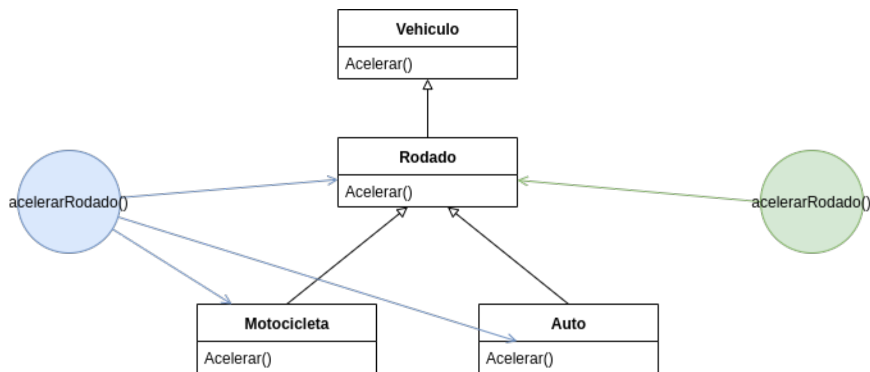

Figura 25: Esquema representativo de la información que Callcluster puede extraer del ejemplo 24. A la izquierda se representa la información que callcluster extrae, a la derecha la que podría extraer si respetara los principios de ocultamiento propios de la programación orientada a objetos. El mismo principio se utiliza para las llamadas a interfaces.

3.4.2. Características funcionales

En las versiones recientes de C# es posible crear funciones literales (también llamadas *lambdas*), pasar funciones como parámetro a otras, y asignarlas a variables. En estos casos, callcluster toma la referencia a la función como si fuera una llamada. Modelar más precisamente estos casos (que deberían ser pocos en un lenguaje orientado a objetos como C#) implica realizar un análisis estático más avanzado. El comportamiento de callcluster en este caso se describe en la figura 27.

```
class EjemploFuncional{
    void Foo(){
```

```
        if(this.Condition()){
            this.Bar(this.BazOne)
        } else {
            this.Bar(this.BazTwo)
        }
    }
    void Bar(Action<void> fn){
        fn()
    }
    void BazOne(){}
    void BazTwo(){}
}
```

Figura 26: Código `C#` con pasaje de funciones como argumento

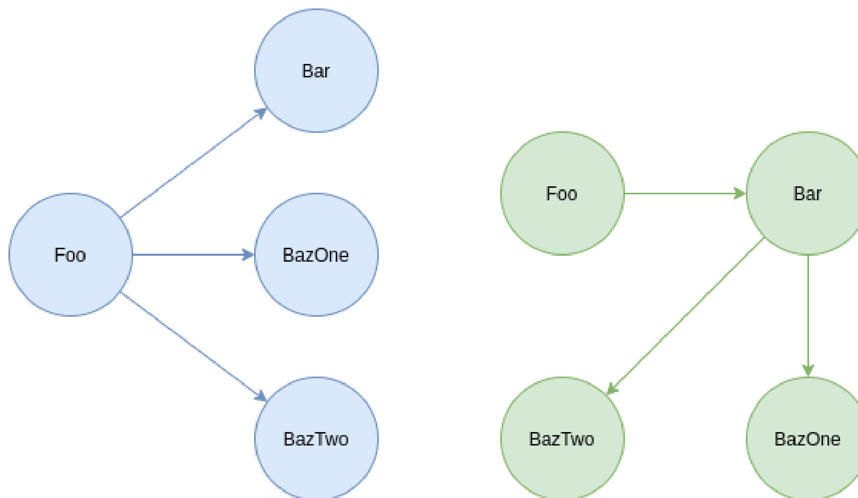

Figura 27: A la izquierda se muestra la aproximación que hace callcluster para el ejemplo 26. A la derecha, una aproximación más fiel al concepto de callgraph.

3.5. Descripción de las métricas recolectadas por los extractores

Los analizadores desarrollados extraen algunas métricas que se pueden relacionar a una noción intuitiva del tamaño de cada función. Se extrajeron las siguientes:

- Complejidad ciclomática McCabe (denominada `Cyclomatic complexity` en la interfaz visual)
- Complejidad ciclomática Basili (denominada `Basili complexity` en la interfaz visual)
- Cantidad de líneas de código (denominada `Number of lines` ó `Lines of code` en la interfaz visual)
- Cantidad de instrucciones contenidas en la función (denominada `Number of statements` en la interfaz)

3.5.1. Complejidad ciclomática Basili

La complejidad ciclomática Basili consiste en una adaptación de la implementación realizada por Basili et. al. en [9] para el análisis de programas en Fortran. Fortran tiene estructuras de control distintas de las que se encuentran en C y C# con lo cual no fue posible mantener la definición exacta.

3.5.2. Adaptación del cálculo de la complejidad ciclomática Basili para C

Dadas las variables:

- I: La cantidad de estructuras `if` contenidas en la función
- W: La cantidad de estructuras `while` contenidas en la función.
- F: La cantidad de estructuras `for` contenidas en la función.
- A: La cantidad de operadores `&&` (*and*) contenidos en la función.
- O: La cantidad de operadores `||` (*or*) contenidos en la función.
- T: La cantidad de operadores condicionales ternarios
- L: La cantidad de etiquetas presentes en la función
- G: La cantidad de `goto` calculados presentes en la función.
- C_c: La cantidad de palabras clave `case` presentes en la función.
- C_d: La cantidad de palabras clave `default` presentes en la función.

La complejidad ciclomática C se calculó como:

$$C = 1 + I + W + F + A + O + T + G \times (L - 1) + C_c + C_d$$

3.5.3. Adaptación del cálculo de la comlejidad ciclomática Basili para C#

El cálculo es idéntico al realizado para C, con la salvedad de que además se cuentan las estructuras `foreach` y `do while`, y que $G = 0$ ya que no existen `goto` calculados en C#.

3.5.4. Implementación del cálculo de la complejidad ciclomática Basili utilizando `libclang`

La implementación de la extracción de la complejidad ciclomática Basili se resume en el fragmento de código de la figura 28. El ejemplo extraído ilustra algunos conceptos importantes de `libclang`:

- Tipo `CXCursor`: Este tipo representa un cursor, que permite recorrer el programa. En la llamada a `get_basili_complexity`, el cursor recibido representa la definición de una función.

- Función `clang_visitChildren`: dado un cursor c y una función f, esta llamada a `libclang` invocará la función sobre todos los nodos del AST descendientes de c. Se permite compartir un "estado"entre distintas invocaciones de la función f por medio del tercer argumento, cuyo valor se pasa en todas las invocaciones (en el ejemplo, es un apuntador al `struct Counts`). A partir del valor de retorno de f se pude controlar el comportamiento de `clang_visitChildren`: En el ejemplo de la figura 28, se retorna `CXChildVisit_Recurse`, que hace que se recorran los nodos descendientes del AST de forma recursiva. Si la función retorna `CXChildVisit_Continue`, la ejecución continúa con los hermanos del nodo (evitando recorrer sus descendientes); si retorna `CXChildVisit_Break`, se cancela el recorrido.

- Función `clang_getCursorKind`: devuelve el tipo de un cursor.

El código de la figura 28 recorre el AST de una función entero contando las ocurrencias de cada una de las características que requiere la fórmula de la complejidad Basili, y luego ejecuta la suma correspondiente.

```
enum CXChildVisitResult
feature_counter(CXCursor cursor, CXCursor parent, CXClientData client_data)
{

    Counts* c = (Counts*) client_data;
```

```
    switch(clang_getCursorKind(cursor)){
        case CXCursor_CaseStmt: c->cases++;
        break;
        case CXCursor_DefaultStmt: c->defaults++;
        break;
        /**
        * A partir del tipo del CXCursor se
        * realiza una suma a otros contadores
        **/
        default:
        break;
    }
    return CXChildVisit_Recurse;
}

unsigned int complexity(Counts counts)
{
    return 1 + counts.defaults + counts.cases;/* + otros contadores */
}

unsigned int
get_basili_complexity(CXCursor c)
{
    Counts counts;
    clang_visitChildren(c,feature_counter, (CXClientData) &counts);
    return complexity(counts);
}
```

Figura 28: Resúmen de la implementación de la extracción de la complejidad ciclomática Basili.

3.5.5. Cantidad de líneas de código

Se cuenta la cantidad total de líneas incluyendo comentarios y líneas vacías.

3.5.6. Cantidad de instrucciones contenidas en la función

La definición de instrucción (en inglés `statements`) varía entre lenguajes. En ambos casos se contabilizó la cantidad de nodos del AST que Roslyn o

libclang caracterizan como instrucción.

3.5.7. Complejidad ciclomática McCabe

En el caso de C#, la complejidad ciclomática McCabe se calculó a partir del Control Flow Graph extraído por Roslyn. Para programas C no se calcula esta métrica. La figura 8 ilustra el cálculo de la complejidad ciclomática.

3.6. Herramientas de análisis evaluadas

No basta con la información "textual"de la firma de un método o función para determinar el callgraph, debido a que:

1. En C#, o en cualquier otro lenguaje orientado a objetos, es necesario lidiar con la herencia.
2. Tanto en C como en C#, puede haber colisiones de nombres (es decir, la misma firma exacta puede estar duplicada en distintos puntos de un sistema bajo análisis, o hacer referencia a funciones distintas)
3. En C, es necesario tener en cuenta el preprocesador, que hace que el código que se compila sea distinto al que se lee con un analizador.

Teniendo en cuenta estos aprendizajes, se evaluaron las siguientes herramientas para la extracción de la información.

3.6.1. ANTLR

ANTLR permite extraer un AST de archivos de texto, y es configurado a partir de una gramática. Existen muchas gramáticas disponibles en internet, lo cual abriría la posibilidad de extender CallCluster a cualquier lenguaje, dada su gramática. Esta herramienta se descartó debido a que requeriría desarrollar los algoritmos correspondientes a las problemáticas enumeradas: un extractor confiable debe estar tan acoplado al compilador como sea posible.

3.6.2. Roslyn

Microsoft separó Visual Studio de las APIs de análisis de de código C# y Visual Basic. El proyecto resultante es llamado Roslyn, de código abierto. Así, el compilador y las APIs de análisis de C# oficiales son abiertas y bien documentadas. No se encontraron soluciones ya desarrolladas para C# que incluyan la recolección de métricas, y que utilicen las versiones actuales de Roslyn y C#.

3.6.3. GCC

GCC incluye opciones para generar diagnósticos de cada unidad de compilación. En las versiones más recientes se incorporó entre estos, la extracción del callgraph. Sin embargo, según la documentación de GCC, el formato de los reportes es inestable y poco confiable, y existen para depurar el compilador mismo, más que para la generación de análisis.

3.6.4. Desarrollar un plugin de GCC

GCC puede ser extendido por medio de plugins, los cuales pueden introducirse en diversas etapas de la compilación para analizar o modificar las representaciones intermedias. Este acercamiento fue descartado en favor de libclang, que está mejor documentado y ofrece una interfaz estable.

3.6.5. LLVM

El proyecto LLVM incluye diversos subproyectos, entre los cuales se encuentran el compilador clang, libclang, cmake, y clang-analyzer. LLVM es un análogo a Roslyn: incluye el compilador y herramientas para analizar tanto el código fuente como el proceso de compilación mismo. Clang ofrece distintas interfaces (referencia). La interfaz acorde para CallCluster es libclang, que es la más estable.

3.7. Análisis de librerías de visualización de grafos

3.7.1. Relevamiento de librerías de visualización de grafos (primera etapa)

Una vez definidas las tecnologías que se utilizarían en el componente graficador, se realizó un relevamiento de librerías de visualización de grafos para javascript. Se tuvieron en cuenta los siguientes criterios:

1. La documentación de la librería explicita que está construida para grafos grandes (10-100K aristas)

2. La librería incluye ejemplos con grafos grandes (10-100K aristas)

3. La documentación de la librería explicita que usa DOM o SVG

4. La documentación de la librería explicita que usa canvas

5. La documentación de la librería explicita que usa WebGL

6. La librería provee ejemplos con la posibilidad de editar el grafo
7. La librería provee ejemplos con la posibilidad de mover vértices del grafo
8. La librería provee algoritmo de layout de fuerzas
9. La librería provee algoritmo de layout circular
10. La librería provee algoritmo de layout jerárquico/árbol
11. Calidad de la documentación a primera vista (ausente/escasa/suficiente/mucha)
12. Tiempo desde el último commit (un año o más/menos de un año/menos de un mes/menos de una semana)

Se excluyeron de la evaluación todas las librerías que no sean de código abierto.

1. Datos reunidos

   Se utilizaron dos listados como fuente de datos de este relevamiento:

   - We <3 graphs - página que lista varias librerías de grafos
   - Post en Medium - The list of graph visualization libraries

Link	docG	ejG	svg	canvas	webgl	ejedi	ejm	fuerzas	circular	árbol	docu	com	puntaje
alchemy.js	0	0	1	0	0	0	1	1	1	1	1	0	1
arbor.js	0	0	1	1	1	0	1	1	0	0	1	0	3
ccNetViz	0	1	0	1	1	0	0	1	1	1	1	1	9
Cytoscape	0	0	1	0	0	1	1	1	1	1	3	3	9
D3	0	0	1	0	0	1	1	1	1	1	3	3	9
dracula	0	0	1	0	0	0	1	0	0	0	0	1	2
El grapho	1	1	0	1	1	0	0	1	1	1	1	1	10
G6	0	0	0	0	0	1	1	1	1	1	3	3	9
ggraph	1	0	1	0	0	1	1	1	0	0	0	0	1
GraphGL	1	0	0	1	1	0	0	0	0	0	0	0	3
graphosaurus	0	0	0	1	1	0	0	0	0	0	1	0	3
H3Viewer	1	0	0	1	1	0	0	0	0	0	0	0	3
InfoVis	0	0	0	1	1	1	1	1	0	1	2	0	4
jointjs	0	0	1	0	0	1	1	0	0	1	3	2	7
ijs-graph.it	0	0	0	0	0	1	1	0	0	0	1	0	1
mxgraph	0	0	1	0	0	1	1	0	0	0	3	3	9
Networkcube	0	0	0	1	1	0	0	1	1	1	0	1	4
sigma.js	0	0	0	1	1	0	0	1	0	0	2	0	4
soba	0	0	1	0	0	0	0	0	0	0	1	0	1
ugraph	1	0	1	0	0	0	0	0	0	0	0	0	1
vis.js	0	1	0	1	1	1	1	1	0	1	2	3	14
ngraph	1	1	1	1	1	0	0	1	1	1	1	1	10
JsNetworkX	0	0	1	1	0	0	0	1	1	1	1	0	2

2. Exclusiones:

- Cola.js (no es para dibujar grafos)
- go.js (paga)
- neovis.js (es para leer archivos de neo4j y mostrarlos con vis.js)
- NetJSON (es para levantar NetJSON con D3.js)
- Rappid (paga)
- Springy (es sólo el force layout)
- dagre-d3 (Es un font-end de D3 para dagre, que es una librería que se ocupa del layout de los grafos)
- jsplumb (Es paga)
- Ogma (Es paga)
- popotojs (Es para armar querys cypher)

- Processing.js (es processing pero para la web)
- Protovis (Es simplemente el precursor de D3)
- ReGraph (Es paga)
- VivaGraphJS (Es un front-end de ngraph)

3. Conclusiones Para decidir se ponderaron los valores según la fórmula

$$docG + ejG \times 4 + canvas + webgl + docu + com \times 2$$

Se fija el umbral en las librerías con puntajes mayores o iguales a 10, que son:

Librería	Puntaje
El Grapho	10
vis.js	14
ngraph	10

A partir de una comparación de la rapidez de varias librerías, se determina realizar un estudio más profundo de las para determinar lo siguiente:

- ¿La librería permite editar y mover el grafo en tiempo real utilizando un layout dirigido por fuerzas?
- ¿Cuánto se deteriora la experiencia de usuario para el callgraph de $\text{DNN}_{\text{Platform}}$ generado en la etapa de prototipado?

Para esto se realiza un estudio más a fondo de la documentación y los ejemplos de las cuatro librerías seleccionadas.

3.7.2. Resultados del relevamiento de librerías de visualización de grafos

1. Experimento realizado

Con el objetivo de

- Determinar una arquitectura
- Probar distintas librerías de visualización de grafos

se construyó una aplicación electron.js con ejemplos para cada librería, para evaluar el comportamiento de las mismas.

2. Conclusiones respecto de las librerías de visualización de grafos

   Tal como detallado en el relevamiento inicial, se evaluaron las siguientes librerías:

   - El Grapho
   - vis.js
   - ngraph

   Para realizar esta prueba, se incluyó en el experimento una página para cada una de las librerías, exhibiendo la visualización de:

   - AppInspector (completo)
   - Las primeras 10000 aristas de DNN
   - Las primeras 10000 aristas de BouncyCastle

   a) El Grapho Tiene muy poca documentación y muy poca funcionalidad, permitiendo únicamente mostrar grafos muy grandes estáticos, es decir, no permite cambiar la posición de los grafos para de esta forma "asistir"al algoritmo de layout de fuerzas. Permite hacer zoom y visualizar distintas partes del grafo, pero no interactuar con éste.

      Aprovecha webgl y permite mostrar grafos enormes, mucho más grandes de lo que permiten las otras librerías. Debido a que no ejecuta el algoritmo de layout a lo largo del tiempo como las otras librerías, los resultados son muy malos si se desea generar la visualización rápidamente.

   b) ngraph Exhibe mejor desempeño que vis.js para los grafos de 10000 aristas (es decir, mayor cantidad de cuadros por segundo). Sin embargo, pudo observarse que está muy mal documentada, lo cual dificultó la realización del prototipo y es un factor de riesgo importante para el proyecto. Además, actualmente no es mantenida, lo cual también es negativo.

   c) Vis.js

      Si bien el desempeño fue peor que el de ngraph, la abundante documentación disponible y la facilidad del desarrollo del prototipo correspondiente son factores muy positivos.

   d) Conclusión

      Debido a las razones ya citadas, sumado al hecho de que es muy probable que no se requiera mostrar grafos demasiado grandes, se

escoge Vis.js. En caso de que sí se requiera exhibir grafos grandes, o que esto se requiera en alguna etapa, se utilizará El Grapho.

3.8. Arquitectura del sistema

El sistema se dividió en tres componentes:

- Extractor para C# (**callcluster-dotnet**)
- Extractor para C (**callcluster-clang**)
- Visualizador (**callcluster-visu**)

La comunicación entre los componentes se realiza a través de la escritura de un archivo por parte de los extractores, que puede ser leído por el visualizador. La figura 29 es un esquema de la arquitectura del sistema.

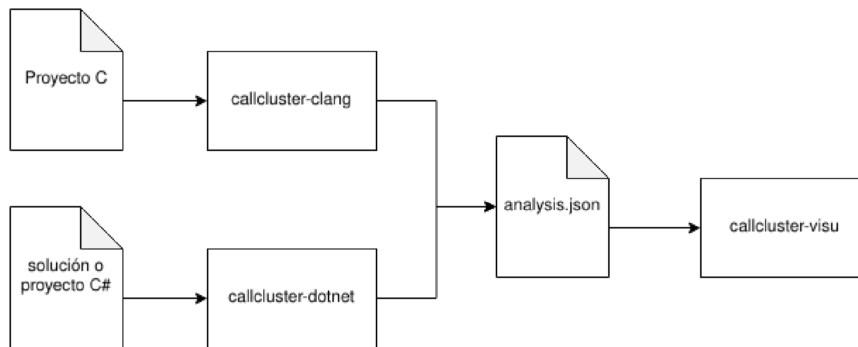

Figura 29: Arquitectura del sistema CallCluster

3.8.1. Principio guía

El principio en que se basó la arquitectura fue minimizar el acoplamiento entre los extractores y el visualizador con dos objetivos:

- que sea sencillo incluir nuevos lenguajes de programación
- que los extractores puedan estar acoplados a las tecnologías específicas de cada lenguaje.

Los extractores desarrollados como parte del alcance del presente proyecto requirieron este nivel de desacoplamiento, ya que C# y C pertenecen a ecosistemas distintos, y son lenguajes muy diferentes. La arquitectura soporta la creación de nuevos extractores sin que sea necesaria la modificación del visualizador.

3.8.2. Comunicación entre componentes: `analysis.json`

Para lograr el máximo desacoplamiento entre componentes, la comunicación entre los mismos se lleva a cabo a partir de un archivo en formato `json` que representa la información que graficará el visualizador. Se eligió usar este método de comunicación por los siguientes motivos:

- La amplia mayoría de los sistemas y lenguajes de programación incluyen la posibilidad de escribir un archivo json. De esta manera, el extractor puede utilizar cualquier tecnología.
- La extracción y el análisis del programa puede llevarse a cabo en un equipo distinto del que se utiliza para exhibir la información.
- El análisis y la extracción pueden ser asincrónicos. Esto es un requisito para tener una experiencia de usuario aceptable, ya que la extracción es una tarea de cómputo mucho más pesada que la visualización.

Los principios de diseño que guiaron la formulación del formato del archivo `analysis.json` fueron:

- Ocultar cualquier tipo de información sobre el lenguaje en que está escrito el programa, el extractor que generó el archivo, o el visualizador que lo debe leer.
- El formato permite extraer métricas arbitrarias.
- El formato está pensado para representar el callgraph y la estructura de un programa: no contempla otros posibles usos del visualizador.

3.8.3. Arquitectura de los extractores `callcluster-dotnet` y `callcluster-clang`

`callcluster-dotnet` fue escrito en el lenguaje C# y utiliza componentes del proyecto Roslyn. `callcluster-clang` fue desarrollado en C, aprovecha libclang. El diseño de ambos es muy similar, ya que tanto Roslyn como libclang aprovechan el patrón visitor, es decir; el cliente de las librerías provee la implementación de un visitor, que es aceptado por aquellas. Así, las librerías proveen el servicio de visitar cada nodo del Árbol de Sintaxis Abstracto (AST). Si bien C no posee objetos, libclang implementa una idea similar: en vez de aceptar un objeto, acepta un apuntador a una función que debe inspeccionar cada nodo para determinar su tipo.

Ambos extractores, al alcanzar la definición de una función, las analizan para obtener las métricas y las llamadas realizadas en el cuerpo de cada

una, almacenando toda esa información en memoria. Una vez terminado el análisis, se transforma la información extraída almacenada en memoria al formato de `analysis.json`.

3.8.4. Arquitectura de `callcluster-visu`

El visualizador fue desarrollado usando el framework quasar, que permite desarrollar interfaces rápidamente, y contiene más componentes que los definidos por el estándar de Material Design. El criterio para utilizar quasar fue el hecho de que permite configurar rápidamente electronjs, y la velocidad que brindaría al desarrollo en comparación con otros frameworks de interfaz de usuario.

Se eligió utilizar electronjs porque actualmente es la forma más popular de desarrollar aplicaciones de escritorio. Además, permite aprovechar el ecosistema de javascript sin quitar la posibilidad interactuar con otros procesos del equipo del usuario. Esto último es requerido para ejecutar el algoritmo Leiden, que sólo está disponible para python; con lo cual el visualizador debe ejecutar un subproceso python.

Electronjs propone una arquitectura dividida en dos procesos: un proceso de dibujado, que tiene acceso al navegador (*Rendering* en inglés) y un proceso llamado *principal*, que tiene acceso a NodeJS y, a través de NodeJS puede acceder a archivos, lanzar procesos, etc. Dado que el archivo `analysis.json` puede tener un tamaño significativo, y que los análisis que deben realizarse para generar las visualizaciones tienen complejidad como mínimo lineal respecto del tamaño de ese archivo, se delegó la mayor cantidad posible de responsabilidades al proceso principal. De esta manera, la interfaz de usuario se ve degradada en menor medida al incrementar el tamaño del proyecto analizado; pero ambos procesos tienen un acoplamiento conceptual muy significativo. En la tabla 2 se detallan las responsabilidades del proceso de *Rendering* y del proceso *principal*.

Cuadro 2: Responsabilidad del proceso de rendering (dibujado) y del proceso principal en `callcluster-visu`.

	Proceso de dibujado	Proceso principal
Responsabilidades	Mantenerse responsivo	Transformaciones del archivo `analysis.json`.
	Ofrecer una interfaz de usuario amigable	Precálculos para agilizar la experiencia de usuario.
		Persistencia y modelo de datos.
		Conexión con el proceso que ejecuta Leiden.

4. Casos de Estudio

4.1. LibUV

libuv es una librería C entrada-salida asincrónica, desarrollada para su uso en nodejs. Es relativamente pequeña, con 38KLOC en total, de las cuales 25K consisten en tests. Se analiza el componente "unix"y se propone una reorganización del código. Utilizando callcluster-visu, se extrajo el mismo, y se clusterizó, llamando a este nuevo componente "unix clustering".

4.1.1. Treemaps

En la figura 30 se muestra la cantidad de líneas de código en cada archivo de la carpeta "unix". Los archivos tienen una cantidad de líneas más bien uniforme, sin acumulaciones demasiado dispares. Al ejecutar el algoritmo de clustering sobre esta carpeta se obtiene una distribución de líneas de código por cluster que se puede observar en la figura 31. Se puede apreciar que el algoritmo de clustering propone una distribución de LOC más uniforme que la observada en la figura 30. En la figura 32, se aprecia que el algoritmo de clustering detecta muy poca correlación entre los archivos ya existentes y los clusters descubiertos, lo cual indica que el código fuente no está organizado a partir de su callgraph.

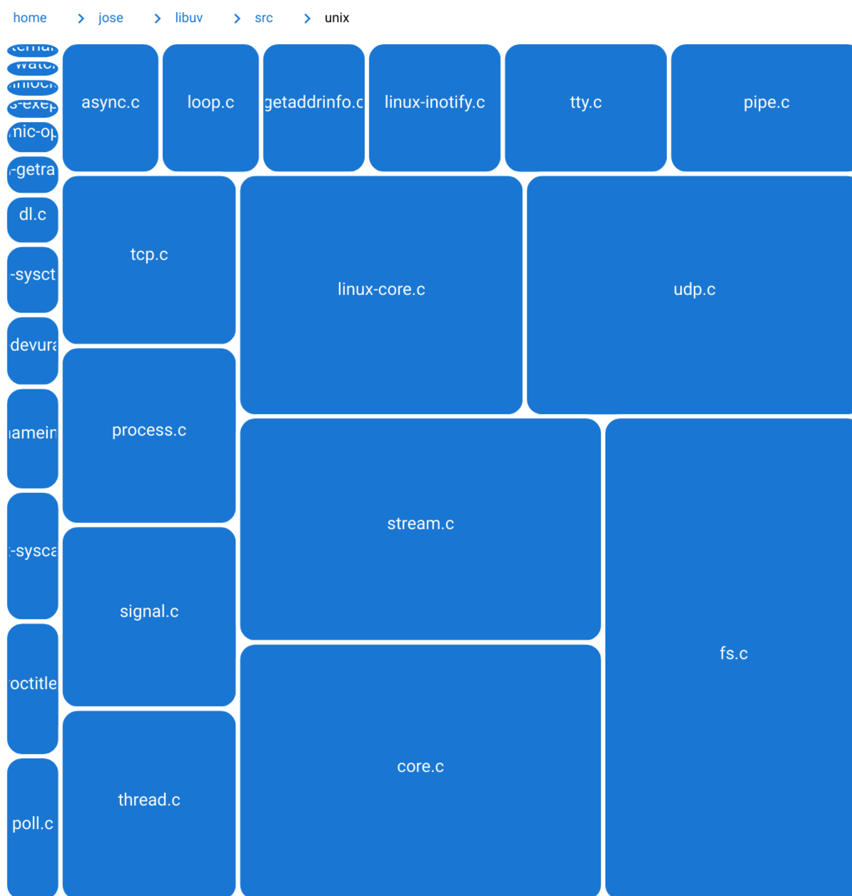

Figura 30: Treemap de la carpeta "unix".

Figura 31: Treemap de la comunidad "unix clustering".

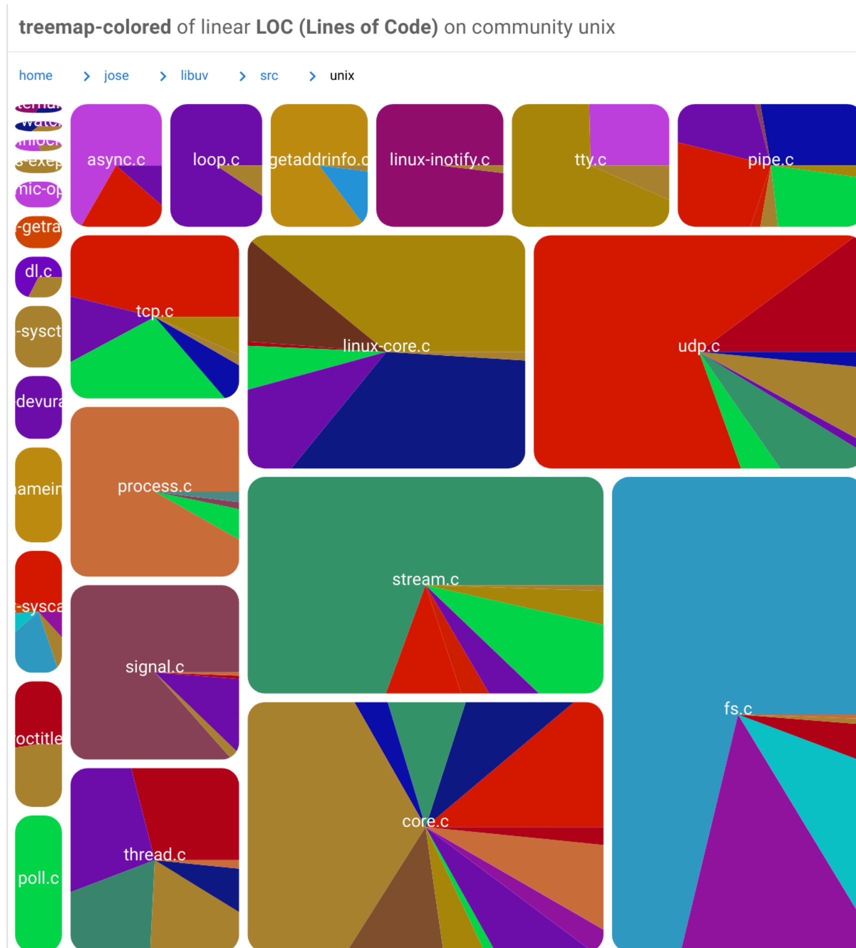

Figura 32: Treemap de la comunidad "unix"coloreado por la participación de cada archivo en cada cluster.

4.1.2. Callgraphs

La figura 33 representa el callgraph del archivo `core.c`, uno de los mayores que se encuentran en la carpeta "unix". Se observa que las funciones que componen el archivo son casi independientes entre sí, lo cual implica que casi ninguna de ellas está oculta hacia el resto de la librería. Los colores de cada vértice indican el cluster al cual fueron asignados. La mayoría de los archivos del módulo analizado exhiben una estructura poco conexa como esa.

Todos los clusters descubiertos exhiben el hecho de que se "capturó"el callgraph: tienen muchas aristas que los conectan, en contraste con el tipo

de estructura exhibida en la figura 33. El callgraph de uno de estos clusters puede observarse en la figura 34. En la figura 35 se observa otro cluster, superpuesto con el callgraph del resto del sistema. Esta imagen permite apreciar el escaso número de aristas que conectan ese cluster con el resto del sistema.

hierarchical-colored of linear **LOC (Lines of Code)** on community unix

jose > libuv > src > unix > core.c

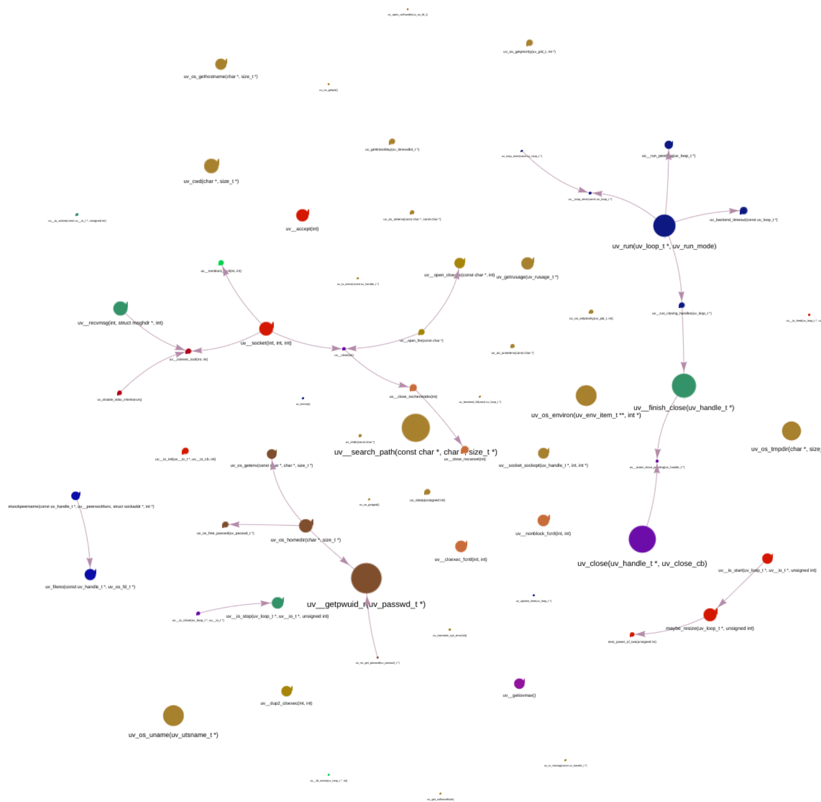

Figura 33: Callgraph del archivo `core.c`.

hierarchical-colored of linear **LOC (Lines of Code)** on unix clustering

clustering > community_1

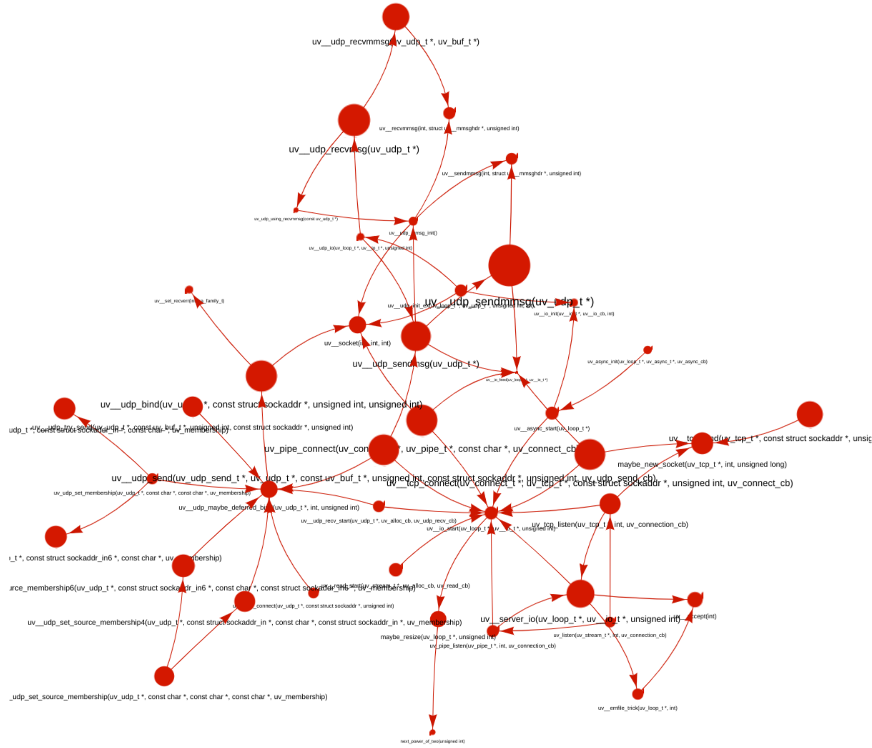

Figura 34: Callgraph de la comunidad 1 (roja) originada a partir del clustering.

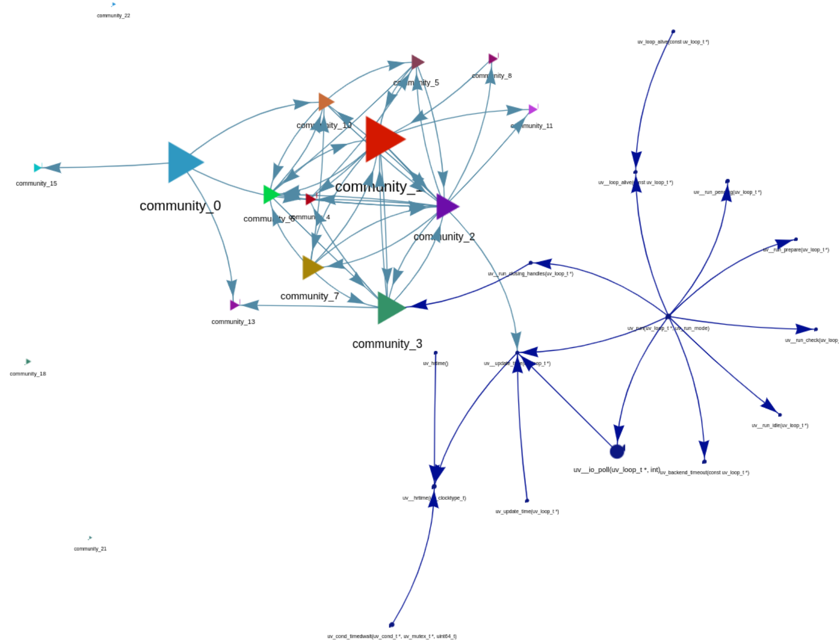

Figura 35: Callgraph de la comunidad 9 (azul oscuro) superpuesto con las demás comunidades. La comunidad 9 (al igual que todas las demás) está poco conectada al resto del programa. Puede apreciarse que tiene un único punto de entrada y un único punto de salida. Otra característica importante que expone este gráfico es la dependencia cíclica entre las comunidades 2 y 3 (violeta oscuro y verde oscuro) y las comunidades 6 y 10 (naranja y verde claro).

4.1.3. Conclusiones

El algoritmo de clustering permite asistir a los diseñadores a reducir las interfaces entre módulos de sus programas C. No es conveniente seguir ciegamente las recomendaciones del algoritmo por dos motivos:

- No puede deducir el criterio con el cual se distribuyeron las funciones en módulos. Al respecto de esto, resulta relevante mencionar que la distribución original del módulo analizado parece respetar "temas"generales: conexiones udp, tcp, threads, etcétera. El algoritmo de clustering ignora completamente esto y se basa únicamente en el callgraph detectado.
- Relaciones de dependencia cíclica. Las dependencias cíclicas no son

posibles en la mayoría de los lenguajes de programación, sin embargo; el algoritmo de clustering no puede evitar relaciones cíclicas entre clusters.

Es recomendable la utilización del algoritmo de clustering como una asistencia al programador, brindándole consejos respecto de cómo reorganizar algunas partes de la arquitectura. De esta forma, le permitiría al usuario tomar una decisión de compromiso más informada entre una orientación a "temas"y una orientación a "modularización"al distribuir las funciones.

4.2. Namespace System.Net.Http del runtime de .Net

Se realizó un análisis restringido al namespace `System.Net.Http` dentro del assembly `System.Net.Http.dll` del runtime de .NET . El runtime de .NET se divide en diversas partes: una nativa, compilada con clang, y un conjunto de bibliotecas, cada una de las cuales es una solución separada. Se realizó una extracción del callgraph de la solución correspondiente y se ejecutó un análisis similar al de libuv. Al contrario que en el ejemplo de libuv, se trabajó con un fragmento mucho menor en relación a la totalidad del código.

4.2.1. Treemaps

Los treemaps del namespace estudiado (figura 36) y del clustering generado (figura 37) permiten observar que el namespace expone una distribución de la complejidad más desigual que libuv. La figura 37 permite observar que el clustering genera una distribución más homogénea que la original, y que se observa poca correlación entre las clases originales y los clusters generados.

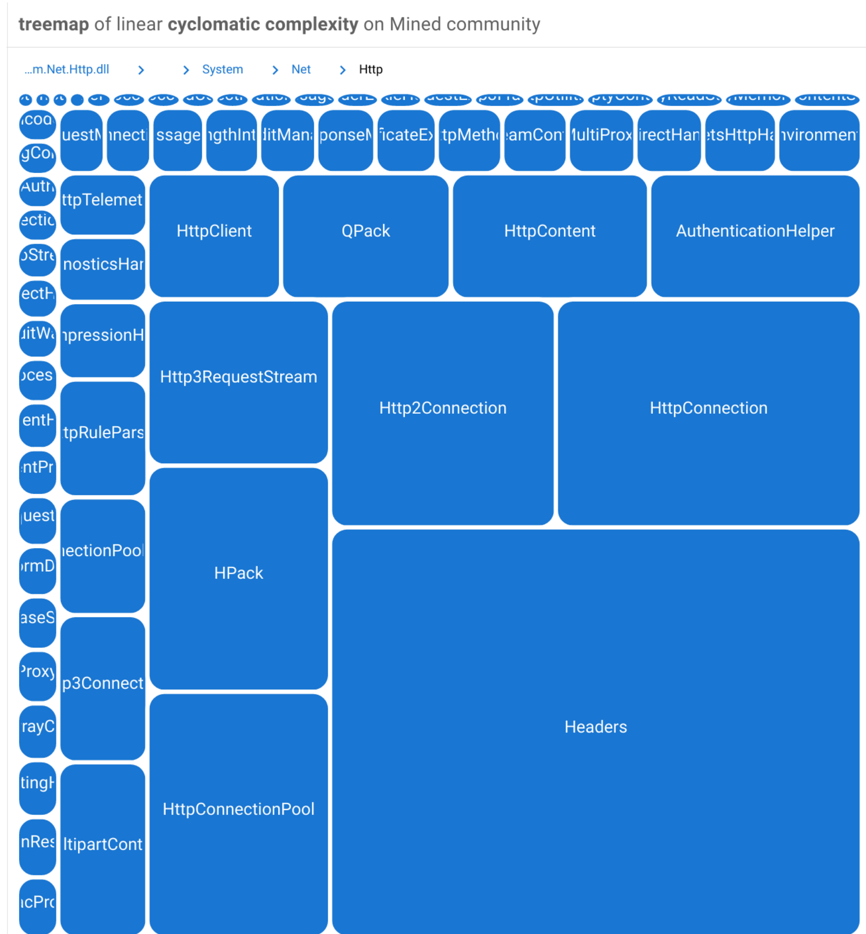

Figura 36: Treemap que representa la distribución de la complejidad ciclomática en las clases del namespace estudiado.

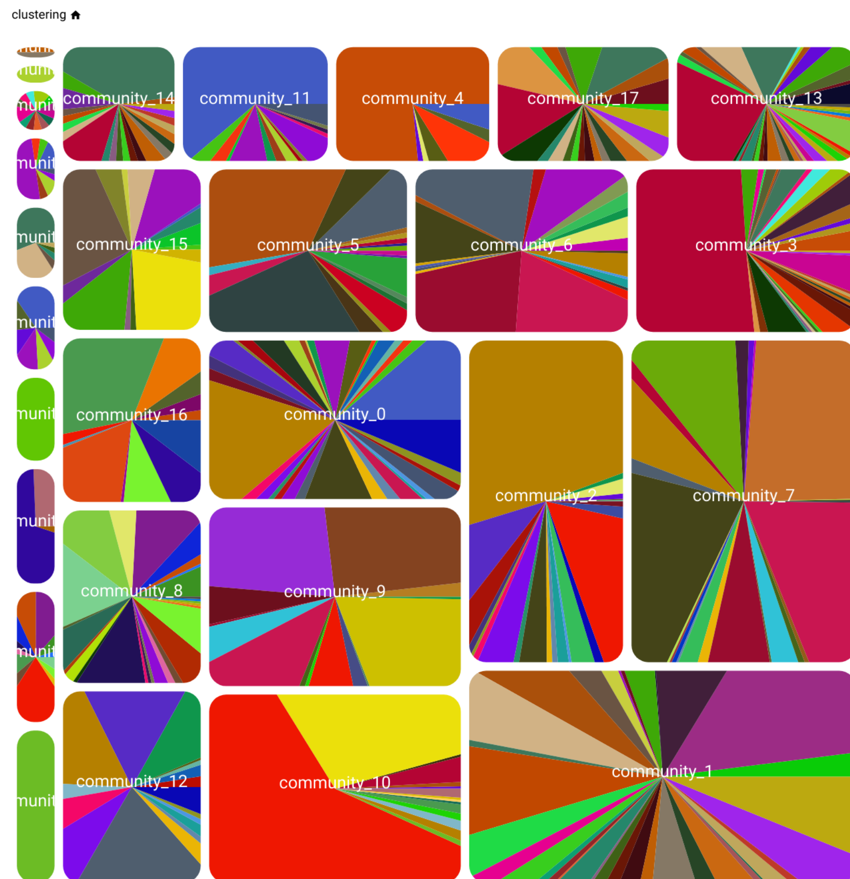

Figura 37: Treemap que muestra la participación de cada clase en cada cluster.

4.2.2. Callgraphs

La figura 38 exhibe el callgraph del namespace analizado, y la figura 39 exhibe el callgraph entre los clusters. Al contrario de lo observado en libuv, el callgraph que forman las clases es más complejo que el observado entre clusters. Dado que el extractor para C# relaciona llamadas que en el código son "implícitas", se observan ciclos en la figura 38. Uno de los motivos por los cuales el callgraph de la figura 39 es menor, es que el algoritmo de clustering detecta subgrafos no conexos y le asigna un cluster a cada uno de ellos: en la figura se incluye únicamente la componente conexa mayor. Al contrario que

en el caso de libuv, se puede apreciar un mayor número de puntos de contacto con el resto del sistema, en el callgraph inducido por el clustering (figura 35). Esto se debe a que las clases originales exhiben una mayor conectividad entre métodos de la misma clase y entre métodos de clases diferentes que en libuv (figura 33). La ilustración 40 ejemplifica esta observación.

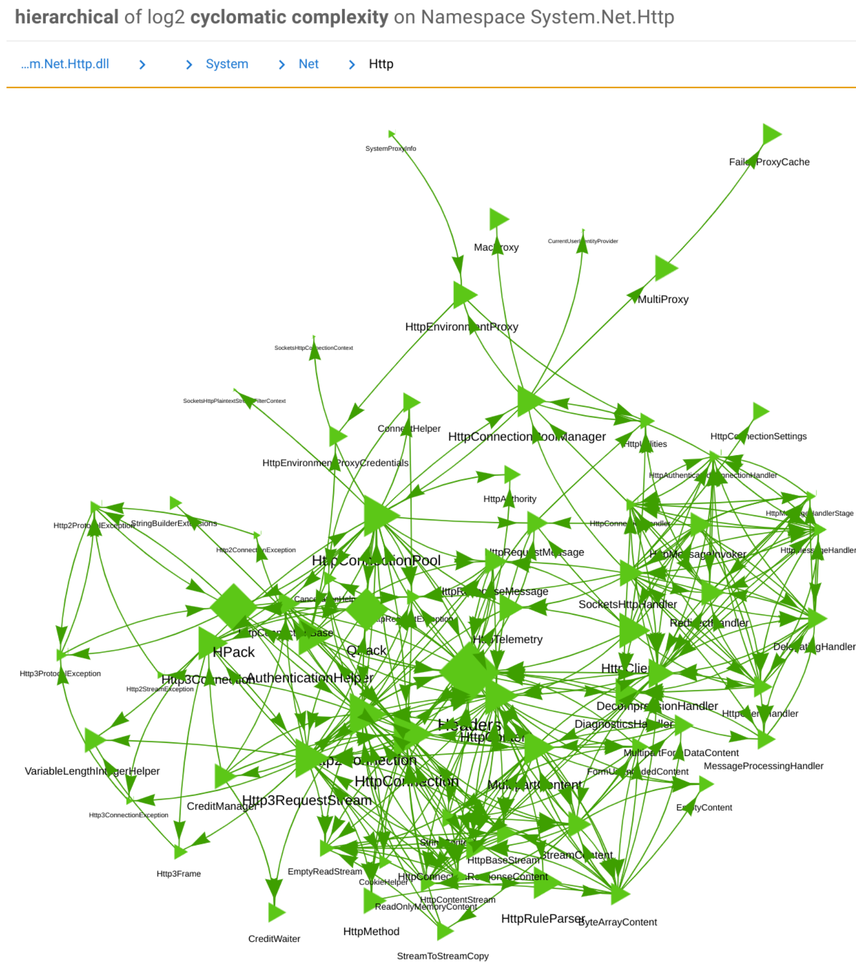

Figura 38: Callgraph general, donde cada clase está representada por un triángulo.

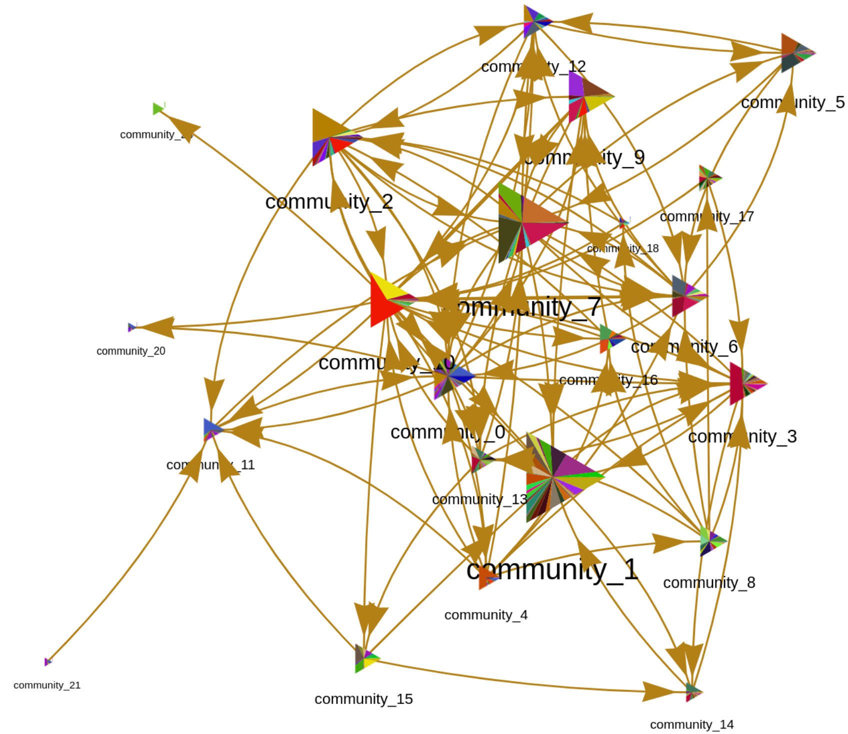

Figura 39: Callgraph inducido por el clustering: los nodos del grafo son clusters, y las aristas representan llamadas entre una función de un cluster y una de otro cluster.

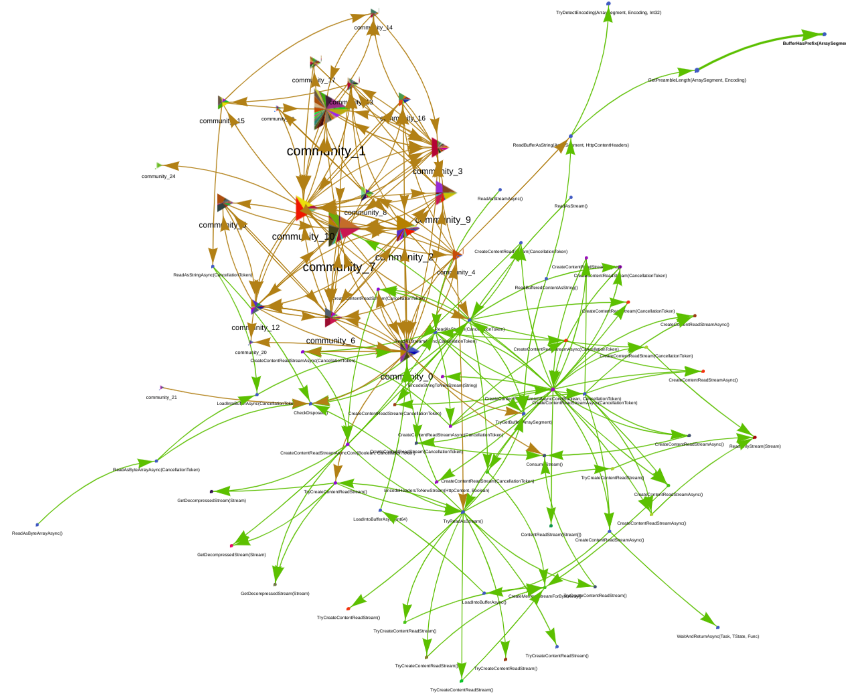

Figura 40: Grafo representado en la figura 39, en el cual se expandió la comunidad 11.

4.2.3. Análisis del namespace `HPack`, incluido en `System.Net.Http`

En las ilustraciones 41 y 42 se exhibe el gráfico "diff"del namespace `HPack`.

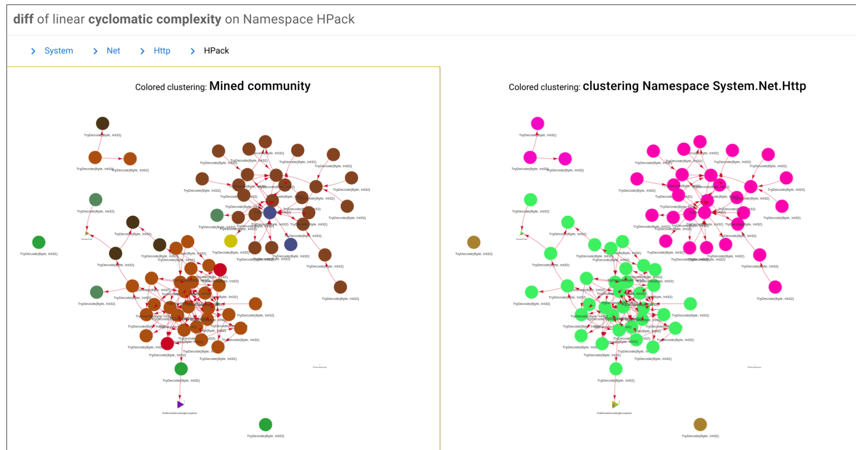

Figura 41: En la ilustración se representan las funciones incluidas en el sub-namespace `HPack` El algoritmo de clustering ubica los contenidos de HPack en 2 clusters diferentes porque contiene 2 componentes que no están conectadas. La estructura descubierta por el algoritmo de clustering se condice sólo de forma aproximada con la estructura real (ver figura 42). En el grafo de la izquierda, los métodos se colorean según la clase a la que pertenecen, en la de la derecha se colorean según su cluster.

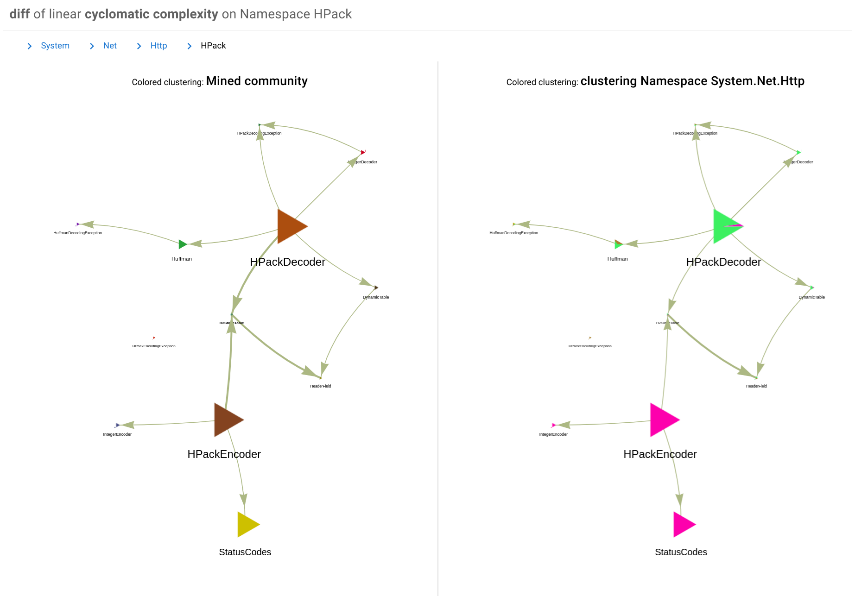

Figura 42: En la ilustración se representa el sub-namespace `HPack`, al igual que en la ilustración 41; pero se exhiben las clases del namespace. Esta gráfico puede utilizarse para justificar la división de este namespace en dos sub-namespaces.

4.2.4. Conclusión

Del análisis del namespace System.Net.Http se puede deducir que es recomendable utilizar callcluster para tomar decisiones más bien "pequeñas"es decir, respecto de la distribución de los métodos entre clases o entre namespaces. Callcluster puede graficar estructuras de más alto nivel pero la información que provee no es suficiente como para realizar modificaciones arquitecturales o de gran escala sobre la estructura del código.

Del análisis de HPack resulta notable la distinción entre el código de "Decodificación"y "Codificación". Esta distinción es evidente a partir del gráfico que genera Callcluster pero es menos evidente al navegar y leer el código fuente.

5. Manual de usuario

5.1. Extracción del callgraph de programas C

5.1.1. Requisitos

callcluster-clang requiere tener instalado el paquete `libclang-10-dev` para poder ejecutarse, busca la librería en `/usr/lib/llvm-10/lib/libclang-10.so` tal como detallado en el archivo `CMakeLists.txt`. Para obtener esta librería debe instalarse clang 10 como descripto en `https://apt.llvm.org/`.

5.1.2. Compilación

1. Clonar el repositorio de código fuente de callcluster-clang

   ```
   git clone https://github.com/callcluster/callcluster-clang
   ```

2. Compilar callcluster-clang

   ```
   cd callcluster-clang
   mkdir build
   cd build
   cmake ..
   cmake --build .
   chmod a+x callclusterClang
   ```

5.1.3. Extracción del callgraph

`callclusterClang` necesita, además de acceso al código fuente, una *compilation database*, que es una enumeración de las invocaciones al compilador de C y al linker. La *compilation databse*, por convención es un archivo denominado `compile_commands.json`. Existe un parámetro de línea de comando de `cmake` que permite generar `compile_commands.json` al compilar el proyecto cuyo callgraph se desea extraer. Si el proyecto analizado no utiliza cmake, puede usarse Bear, que funciona con cualquier *toolchain*. El archivo `compile_commands.json` utiliza rutas relativas para identificar los archivos de código fuente, con lo cual no se debe cambiar su ubicación.

En el caso de utilizar bear, debe tenerse en cuenta que Bear tiene que capturar todas las invocaciones al compilador, con lo cual suele ser necesario ejecutar `make clean` antes de ejecutar `bear make`.

Una vez obtenido `compile_commands.json`, puede ejecutarse el binario `callclusterClang` de la siguiente manera:

```
./callclusterClang <directorio donde se encuentra
compile_commands.json> --progress
```

Una vez que se termina de ejecutar `callclusterClang`, se escribe un archivo denominado `analysis.json` en la carpeta donde se ejecutó el comando (es decir, en el *current working directory*, **no** en el directorio de `compile_commands.json`).

5.1.4. Opciones recibidas por callclusterClang

```
./callclusterClang [CARPETA] [--progress]
```

- La `CARPETA` es obligatoria, y debe tener un archivo llamado `compile_commands.json`
- La opción `--progress` es opcional, e indica si se desea que el programa escriba a la consola información sobre el avance. Esta opción es importante para proyectos con muchos archivos de código fuente.

5.1.5. Ejemplo (php)

Asumiendo ubuntu, según `README.md`:

```
git clone https://github.com/php/php-src
cd php-src
sudo apt install -y pkg-config build-essential autoconf bison re2c \
libxml2-dev libsqlite3-dev
./buildconf
./configure
bear make
```

En el directorio donde se encuentra `callclusterClang`:

```
./callclusterClang <ubicación de la carpeta php-src> --progress
```

5.1.6. Ejemplo (redis)

```
git clone https://github.com/redis/redis
cd redis
bear make
```

En el directorio donde se encuentra `callclusterClang`:

```
./callclusterClang <ubicación de la carpeta redis> --progress
```

5.2. Extracción del callgraph de programas C

5.2.1. Instalación y ejecución

Se requiere contar con .Net 5.0 SDK instalado.

1. Clonar el repositorio que tiene el código de callcluster-dotnet:

   ```
   git clone https://github.com/callcluster/callcluster-dotnet
   ```

2. Invocar `dotnet run` en la carpeta donde se encuentra el extractor, pasando como argumento la ubicación del archivo `.sln` ó `.csproj` cuyo callgraph se desea extraer. Esto genera un archivo denominado `analysis.json` en la carpeta donde se ejecutó `dotnet run`.

   ```
   cd callcluster-dotnet/callcluster-dotnet
   dotnet run <ruta al archivo sln ó csproj>
   head analysis.json
   ```

5.2.2. Ejemplo (DNN.Platform)

```
git clone https://github.com/callcluster/callcluster-dotnet
wget https://github.com/dnnsoftware/Dnn.Platform/archive/master.zip
unzip master.zip
cd callcluster-dotnet/callcluster-dotnet
dotnet run ../../Dnn.Platform-master/DNN_Platform.sln
head analysis.json
```

5.2.3. Ejemplo (bc-csharp)

```
git clone https://github.com/callcluster/callcluster-dotnet
wget https://github.com/bcgit/bc-csharp/archive/master.zip
unzip master.zip
cd callcluster-dotnet/callcluster-dotnet
dotnet run ../../bc-csharp-master/csharp.sln
head analysis.json
```

5.3. Uso del visualizador callcluster-visu

5.3.1. Requisitos

El usuario debe tener instalados en su sistema:

1. nodejs y npm
2. yarn
3. python 3
4. pip

5.3.2. Instalación y ejecución

1. Clonar el repositorio

   ```
   git clone https://github.com/callcluster/callcluster-visu
   cd callcluster-visu
   ```

2. Instalar dependencias de javascript (yarn)

   ```
   yarn install
   ```

3. Instalar dependencias de python (se recomienda usar un ambiente virtual)

   ```
   python3 -m venv venv
   . venv/bin/activate
   pip install -r requirements.txt
   ```

4. Iniciar el visualizador. Si las dependencias python se instalaron en un ambiente virtual, entonces debe activarse el mismo antes de correr el comando a continuación.

   ```
   yarn run quasar dev -m electron
   ```

5.3.3. Conceptos básicos (modelo mental)

1. Comunidad El esquema de la figura 43 muestra cómo la jerarquía de un programa C# se traduce al modelo de callcluster. De forma análoga, para un programa C, las carpetas y archivos pasan a denominarse comunidades. Las comunidades describen la estructura jerárquica de un programa. Estas estructuras existen en todos los lenguajes de programación, aunque siempre tienen nombres distintos.

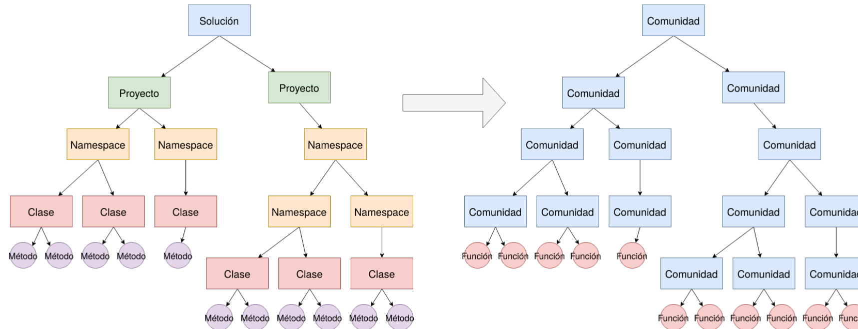

Figura 43: Interpretación de un programa C# en el modelo de CallCluster

2. Función y callgraph El concepto de *función* se encuentra en todos los lenguajes de programación con distintos nombres. Las funciones se vinculan entre sí formando un gafo. En callcluster este grafo se denomina *callgraph*. Una representación visual del concepto de callgraph se puede observar en la figura 44.

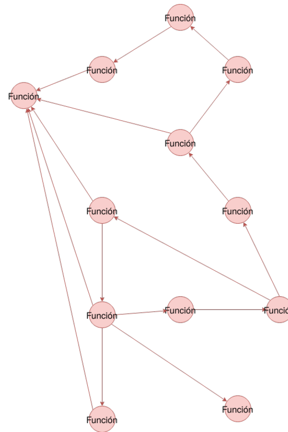

Figura 44: callgraph

3. Clustering Un *clustering* es un agrupamiento automático de funciones, que se calcula a partir del *callgraph* y detecta grupos de funciones más relacionadas entre sí. La figura 45 es un posible clustering del callgraph representado en la figura 44.

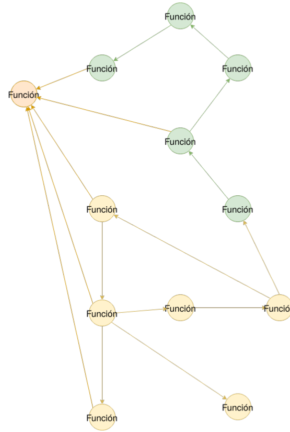

Figura 45: callgraph clusterizado

Un clustering es también una comunidad, ya que es una agrupación disjunta de funciones. La figura 46 representa este concepto para el callgraph clusterizado de la figura 45. *Callcluster* actualmente no extrae clustering jerárquicos, con lo cual la ejecución del algoritmo de tiene como resultado una jerarquía de un solo nivel.

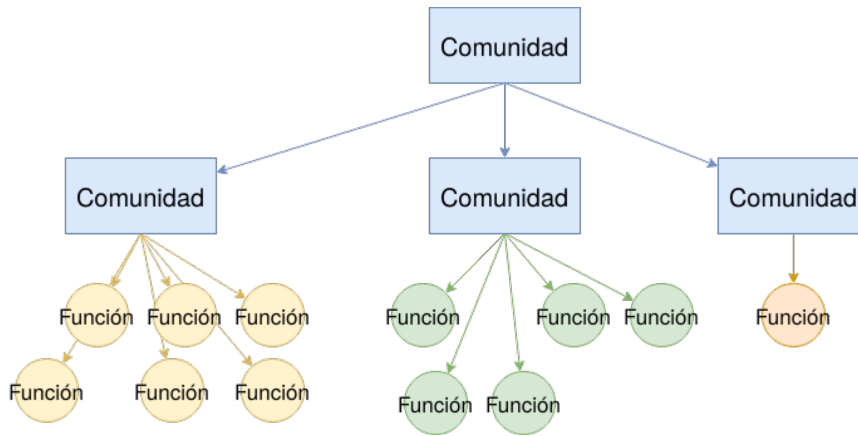

Figura 46: Comunidad generada

4. Visualización Callcluster permite crear 6 tipos de visualizaciones, que se encuentran explicadas en la interfaz, junto con los parámetros que reciben.

5.3.4. Descripción general de la interfaz

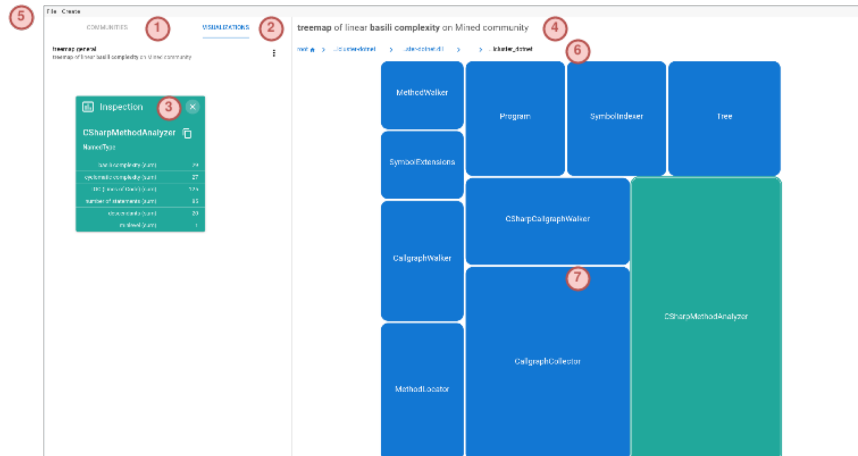

Figura 47: Interfaz de `callcluster-visu`

En esta sección se describen los componentes de la interfaz del visualizador enumerados en la figura 47.

1. **Pestaña "Communities":** muestra una lista de comunidades que incluye la importada, las extraídas y los clusterings.

2. **Pestaña "Visualizations":** muestra un listado de las visualizaciones creadas, permitiendo editarlas y eliminarlas.

3. **Ventana "Inspection":** muestra toda la información disponible sobre la comunidad seleccionada en la visualización (6)

4. **Título del gráfico:** Es un título generado automáticamente que describe el contenido del gráfico

5. **Menú general:** Permite acceder a las siguientes opciones:

   - Importar archivo analysis.json (*File* -> *Import analysis.json*)
   - Crear visualización (*Create* -> *Visualization*)
   - Crear clustering (*Create* -> *Clustering*)

6. **Barra de navegación:** Muestra en qué lugar de la jerarquía de comunidades se ubica la visualización actual. Al hacer click en una comunidad previa se puede navegar a la misma.

7. **Visualización:** En este sector se muestra la visualización.

5.3.5. Uso del visualizador

1. Cómo agregar una comunidad Existen 3 formas de agregar comunidades: - Importar un archivo analysis.json (requisito para todo lo demás) - Extraer una sub-comunidad - Crear un clustering

2. Cómo importar analysis.json

   a) Hacer click en File ->Import analysis.json (en el menú superior izquierdo), o presionar Ctrl+I

   b) Elegir el archivo `analysis.json` generado a partir de `callcluster-dotnet` ó `callcluster-clang`

   c) En la barra lateral se verá la primera comunidad: `Mined community`. Esta comunidad contiene todas las comunidades descriptas en `analysis.json`.

3. Cómo crear una visualización Antes de crear una visualización, debe importarse `analysis.json`. Según la visualización que será creada, puede ser recomendable crear un *clustering* previamente.

   a) Pasos a seguir

      1) Hacer click en Create ->Visualization (en el menú superior izquierdo), o presionar Ctrl+D
      2) Seguir paso a paso el asistente
      3) En la barra lateral, pestaña *Visualizations* se listan las visualizaciones creadas y se permite editarlas o eliminarlas.

4. Cómo extraer una sub-comunidad Las visualizaciones *Hierarchical Graph*, *Hierarchical Colored Graph* y *Diff Graph* permiten hacer click derecho sobre una comunidad para extraerla a la barra lateral. También se puede hacer click derecho sobre el espacio blanco en un gráfico para extraer la comunidad que se está viendo actualmente. Por ejemplo, en la captura de la figura 48, hacer click derecho sobre la parte blanca extrae la comunidad "Program", que es la última listada en la barra superior.

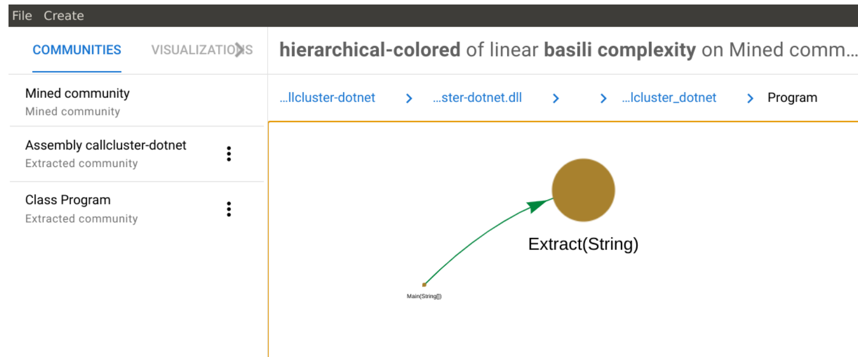

Figura 48: "Program"es la última comunidad listada en la barra superior.

5. Cómo crear un clustering

   a) Hacer click en Create ->Clustering (en el menú superior izquierdo), o presionar Ctrl+Q

   b) Completar el formulario.

   c) El clustering se lista en la barra lateral, junto con todas las comunidades.

   Nótese que es posible crear un clustering a partir de una sub-comunidad de la comunidad minada extraída de `analysis.json`. Esto permite restringir el análisis a una parte del código fuente (por ejemplo, una clase; o un namespace).

6. Cómo interactuar con las visualizaciones

Cuadro 3: Posibles maneras de interactuar con las distintas visualizaciones.

Tipo de visualización	Treemap	Treemap coloreado	Histograma	Grafo jerárquico	Grafo jerárquico coloreado	Grafo diff
Seleccionar (**click izquierdo**)	**sí**	**sí**	no	**sí**	**sí**	**sí**
Navegar (**doble click**)	**sí**	**sí**	no	**sí**	**sí**	**sí**
Explotar (**Menú contextual** ó **Ctrl + Click**)	no	no	no	**sí**	**sí**	**sí**
Listar contenido (**Menú contextual**)	no	no	no	**sí**	**sí**	**sí**
Extraer (**Menú contextual**)	no	no	no	**sí**	**sí**	**sí**

a) Acción *seleccionar* (**click izquierdo**) Al hacer click izquierdo sobre una comunidad o función, se abre una ventana flotante con toda la información disponible sobre ese elemento. Si la ventana ya existe, sus datos pasan a ser los de la última comunidad o función seleccionada. La ventana flotante se ilustra en las figuras 49 y 50.

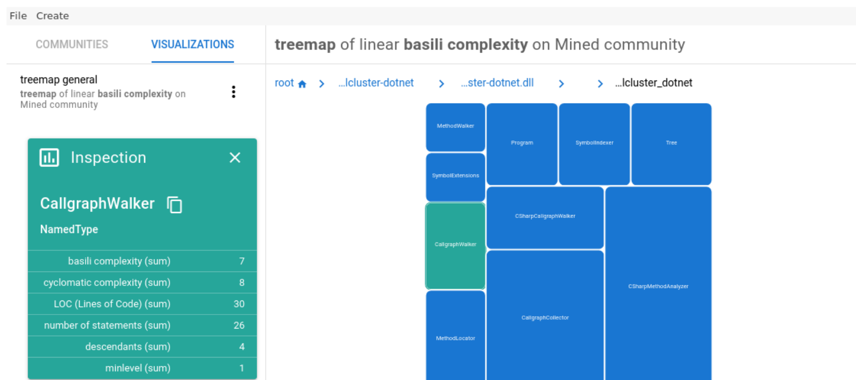

Figura 49: Selección de una comunidad en un gráfico treemap.

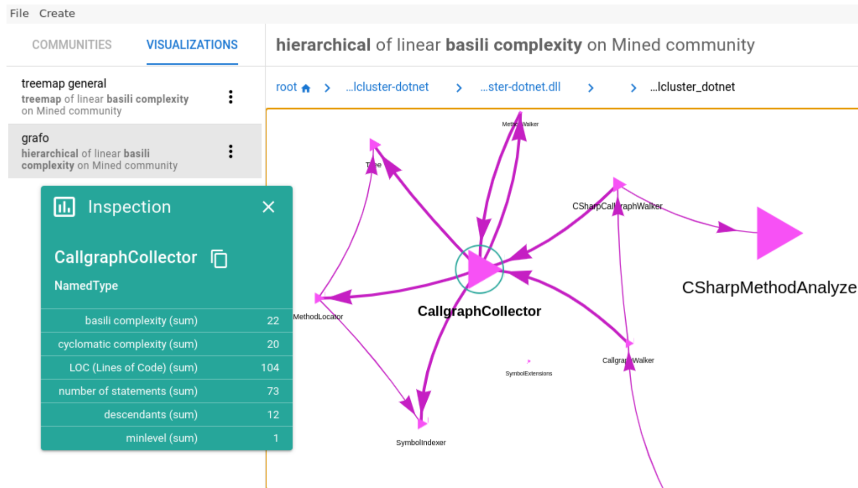

Figura 50: Selección de una comunidad en un callgraph.

b) Acción *navegar* (**doble click**) Hace que la visualización muestre la comunidad hacia la cual se navegó, modificando la ubicación reflejada en la **Barra de navegación**.

c) Acción *explotar* (**Ctrl + Click izquierdo** ó a través del menú contextual)

La acción explotar abre la comunidad elegida, de forma que sus componentes pasan a ser parte del grafo. De esta manera se puede ver su contenido y cómo está relacionado con el resto del callgraph.

d) Acción *extraer* (se accede a través del **Menú contextual**)

Esta acción puede llevarse a cabo sobre una sub-comunidad (al hacer click derecho sobre una de los comunidades que se visualizan) o sobre la comunidad raíz actual (al hacer click derecho sobre el espacio vacío del gráfico). Al realizarla, se abre una ventana con un formulario y la comunidad se agrega a la barra lateral. De esta forma pueden realizarse análisis que tengan en cuenta únicamente a esa comunidad.

e) Acción *listar contenido* (se accede a través del **Menú contextual**)

Esta acción puede llevarse a cabo sobre una sub-comunidad (al hacer click derecho sobre una de los comunidades que se visualizan) o sobre la comunidad raíz actual (al hacer click derecho sobre el espacio vacío del gráfico). Al realizarla, se exponen di-

versos datos sobre la composición de esa comunidad. Un ejemplo puede verse en la ilustración 51.

Contents

5 clusters contained

1 2 3 4 5

Image	Size	Type	Name	Ownership
	22	NamedType	CallgraphCollector	59% cluster 1 22% cluster 5 19% other
	7	NamedType	CallgraphWalker	100% cluster 3
	10	NamedType	CSharpCallgraphWalker	70% cluster 4 20% cluster 3 11% other
	29	NamedType	CSharpMethodAnalyzer	65% cluster 2 34% cluster 4
	8	NamedType	MethodLocator	50% cluster 1 37% cluster 5 13% other
	4	NamedType	MethodWalker	100% cluster 1
	8	NamedType	Program	100% cluster 3
	4	NamedType	SymbolExtensions	100% cluster 2
	8	NamedType	SymbolIndexer	50% cluster 5 37% cluster 1 13% other
	9	NamedType	Tree	44% cluster 1 44% cluster 5 12% other

Figura 51: Tabla que puede observarse al ejecutar la acción *listar contenido*.

5.4. Especificación del formato de callgraphs para callcluster `analysis.json`

Este permite expresar 3 datos extraídos del análisis de un programa, a saber: - callgraph (llamadas entre funciones) - funciones: meta datos y métricas sobre las funciones que componen el programa - estructura jerárquica del programa

Los campos marcados con `(*)` son opcionales: `callcluster-visu` los interpreta, pero no requiere su existencia.

5.4.1. Archivo emitido por los extractores

El archivo permite agregar métricas arbitrarias a las funciones, como campos en los diccionarios contenidos en el array de funciones.

```
{
    "calls":[
        {
            "from":<indice en el array ’functions’ (entero)>,
            "to":<indice en el array ’functions’ (entero)>,
        }
    ],
    "functions":[
        {
            "location":<Cadena de caracteres que permite vincular
            esta función con la ubicación donde se escribió la
            misma, formato libre>,
            "name":<Nombre de la función>,
            "written": < (*) valor booleano, indica si esta función
            fue escrita por el programador o no. Es opcional >,
            "<nombre de la métrica>":<(*) valor de la métrica>
        }
    ],
    "community":<un diccionario tal como definido en la sección
    ’diccionario Comunidad’>
}
```

5.4.2. diccionario Comunidad

Representa una comunidad, y contiene instancias de sí mismo para representar niveles inferiores

```
{
    "name":<nombre de la comunidad>,
    "functions":<Array índices del array ’functions’>,
    "communities":<Array de diccionarios Comunidad>,
    "type":<(*) nombre de este tipo de >
}
```

5.4.3. Extensibilidad

Este formato está diseñado para aceptar la adición de cualquier cantidad de métricas arbitrarias. Si bien los nombres hacen referencia al caso concreto de análisis de callgraphs, podría usarse con otros fines; o para expresar otras dependencias.

6. Conclusión

Dado el hecho de que existen pocas herramientas de código abierto para la visualización de software, y ninguna que ejecute análisis matemáticos sobre su estructura para proponer modificaciones; se evaluó la viabilidad del desarrollo de tal herramienta. Para determinar esto, se llevó adelante una etapa de investigación en profundidad, en la cual se evaluaron y descartaron diversas herramientas de análisis de programas escritos en varios lenguajes de programación. Esta etapa concluyó en la selección de libclang y Roslyn.

Respecto de los análisis matemáticos que incluiría la herramienta, se determinó que el más conveniente es el algoritmo Leiden, publicado hacia fines de 2019. Fueron determinantes para esta decisión los resultados de los benchmarks de tiempo de ejecución incluidos en la publicación, y la prueba matemática de que las comunidades descubiertas por el algoritmo son conexas.

Se tomaron decisiones de diseño de interfaz, optando por construir una herramienta que permita extraer fácilmente los callgraphs, y que también permita generar visualizaciones fácilmente. En el caso de la herramienta de visualización y análisis, se optó por un diseño que permita poca personalización pero que sea fácil de entender. Sin embargo, aún resulta dificil expresar de forma intuitiva por medio de la interfaz el funcionamiento de Leiden y sus heurísticas. Se optó por una estética guiada por los principios de material design y se priorizó la responsividad en el diseño del visualizador. Todas estas decisiones están alineadas con el objetivo de Callcluster: facilitar la comprensión del software.

Como parte del alcance de Callcluster se construyó una arquitectura que busca la independencia entre el visualizador y los extractores. Así, permite realizar la extracción y la visualización en momentos y lugares distintos, y desacopla el analizador y las tecnologías de extracción. Así, callcluster conforma una herramienta sumamente extensible, que puede ser extendida a cualquier lenguaje de programación o tecnología.

Una vez determinado un alcance aproximado y eliminados los riesgos técnicos, se desarrolló Callcluster de forma iterativa, con reuniones periódicas donde se realizaba una demostración y se revisaba el alcance.

Brooks en su ensayo "No Silver Bullet"[10] detalla cuatro fuentes de complejidad esencial en la Ingeniería de Software: Complejidad, Conformidad, Modificabilidad e Invisibilidad. Callcluster ataca la primera y la cuarta. Tal como postulado en ese ensayo, Callcluster no es una bala de plata, sino una herramienta para comprender el software. Tal como detallado en la sección çasos de estudio", callcluster no puede automatizar el diseño de software

sino que brinda información que puede llevar al diseñador a introducir una mejora.

6.1. Trabajos futuros

- Extender el visualizador para incluir más visualizaciones y filtros más avanzados sobre la información exhibida
- Abstraer la construcción del archivo `analysis.json` para que no sea necesario desarrollar un extractor para cada lenguaje.
- Adaptar el visualizador para que pueda ser ejecutado en un navegador web.
- Adaptar la herramienta para que pueda ser utilizada como extensión en una IDE
- Agregar la capacidad de modificar el código fuente del programa analizado, permitiendo llevar a cabo las reorganizaciones propuestas por Leiden.
- Diseñar un algoritmo de clustering que permita asegurar que las comunidades descubiertas, además de ser conexas, no tienen ciclos entre ellas.
- Llevar adelante casos de estudio en los cuales Callcluster manifieste ser útil para mejorar la arquitectura de un sistema monolítico productivo o separarlo en microservicios. También se puede estudiar la utilidad de callcluster en la comprensión y refactorización de sistemas legacy en otras áreas, tales como los programas de computación científica.

Referencias

[1] Ralph Grimaldi. *Discrete and combinatorial mathematics : an applied introduction*. Pearson Addison Wesley, Boston, 2004.

[2] Brian Kernighan. *The C programming language*. Prentice Hall, Englewood Cliffs, N.J, 1988.

[3] Flemming Nielson. *Principles of program analysis*. Springer, Berlin New York, 1999.

[4] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[5] Dexter Kozen and Wei-Lung Dustin Tseng. The böhm–jacopini theorem is false, propositionally. *Mathematics of Program Construction*, page 177–192.

[6] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, Jul 1970.

[7] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[8] Mariano Mendez. *Aplicaciones de Cómputo Científico: Mantenimiento del Software Heredado*. PhD thesis, 04 2016.

[9] V.R. Basili, R.W. Selby, and T. Phillips. Metric analysis and data validation across fortran projects. *IEEE Transactions on Software Engineering*, SE-9(6):652–663, Nov 1983.

[10] Frederick P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.