Callcluster: extracción, análisis y visualización de callgraphs

José Ignacio Sbruzzi[1]

Facultad de Ingeniería de la Universidad de Buenos Aires, Av. Paseo Colón 850 - C1063ACV - Buenos Aires - Argentina jose.sbru@gmail.com

Resumen Callcuster es una plataforma extensible que permite juzgar y entender la estructura del código fuente en diversos niveles de abstracción, a partir de la visualización del mismo como un grafo, y su análisis por medio de técnicas de Ciencia de Datos. Callcluster comprende una herramienta de visualización y análisis, y dos extractores que generan el callgraph para programas C y C#. La arquitectura permite añadir nuevos lenguajes y métricas con facilidad. El analizador permite ejecutar un algoritmo de clustering sobre el grafo extraído y los clusters pueden interpretarse como entidades de software para inspirar una reorganización del código fuente. Existen pocas herramientas abiertas de visualización de código, ninguna de ellas permite realizar el tipo de análisis que implementa Callcluster.

Keywords: Ciencia de Datos, Lenguajes de programación, Clustering, Análisis de programas, Minería de software.

1. Introducción

1.1. Motivación

Existen pocas herramientas de análisis de código fuente, y aún menos que sean de código abierto. 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 requerida. La capacidad de ver y mejorar la arquitectura de un sistema de software puede mejorar 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:

1. Moose [6]
2. Proyecto Bauhaus [7]
3. ConQAT [8]
4. SoftVis3D [9]
5. Sourcetrail [10]

Proyecto Bauhaus y ConQAT son herramientas que inicialmente eran open-source 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.

Moose La plataforma Moose está basada en pharo, y es un proyecto compuesto por muchos equipos de investigación alrededor del mundo. 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#, los cuales generan un archivo que incluye información estructural (es decir, la organización del código en clases, namespaces, etcétera), pero no incluye un callgraph aproximado.

Sourcetrail Sourcetrail 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 no puede ser extendido fácilmente. No permite analizar el callgraph.

2. Implementación

2.1. Interfaz de usuario

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 mental simple y que permite poca personalización de los gráficos. El mismo está compuesto por los siguientes conceptos:

1. Función
2. Callgraph
3. Comunidad: es un conjunto de comunidades y/o funciones disjunto, que puede ser:
   a) La estructura jerárquica en que los programadores organizaron las funciones
   b) El resultado de la ejecución de un algoritmo de clustering
   c) Otra estructura jerárquica que el usuario genere al usar callcluster.
4. Visualización: una visualización creada por el usuario. Permiten cruzar información de hasta dos comunidades.

La decisión de diseño de interfaces más importante fue la simplificación de la construcción de las visualizaciones: El usuario debe escoger de entre una puñado de tipos de visualizaciones disponibles, con pocas parametrizaciones posibles. De esta manera se establece un lenguaje visual que facilita la comunicación entre los usuarios de Callcluster.

2.2. Dificultades con la aproximación del callgraph en C

El preprocesador El preprocesador hace imposible definir cuál será el código que va a ser compilado únicamente inspeccionando el programa. El uso de preprocesador incluso introduce lo que podría interpretarse como errores de sintaxis. La solución a este problema es utilizar una herramienta del proyecto LLVM que inspecciona el proceso de compilación y extrae las invocaciones al compilador y al enlazador, permitiendo de esta forma analizar el proceso mismo de compilación.

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, sólo se puede analizar una de ellas a la vez. Esta problemática se ejemplifica en la figura 1.

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 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 2 expone un ejemplo de esta problemática.

2.3. Aproximación del callgraph en C#

Herencia Una dificultad al extraer el callgraph de métodos C# es que todas las invocaciones de métodos no estáticos son polimórficas. Callcluster conecta todos los métodos que podrían ser llamados: se grafica la información que el programador puede averiguar (porción izquierda de la figura 3), no lo que está escrito en el código fuente (porción derecha de la misma figura).

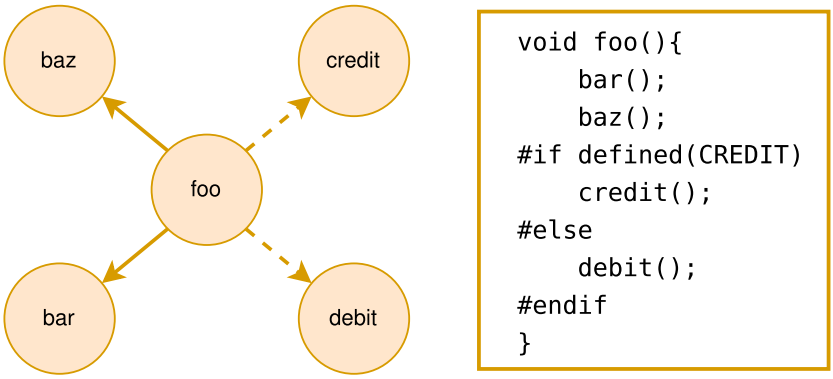


Figura 1. Un fragmento de código y callgraphs posibles. No es posible saber cuál de las dos aristas punteadas pertenece al programa a partir del código: es necesario saber si el flag `CREDIT` está definido al compilar.

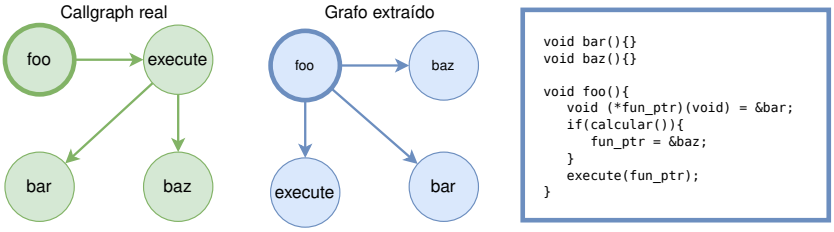


Figura 2. Dado el fragmento de código C de la figura, Callcluster calcula el grafo de la derecha. El grafo de la izquierda es una aproximación más fiel al código y al concepto de callgraph.

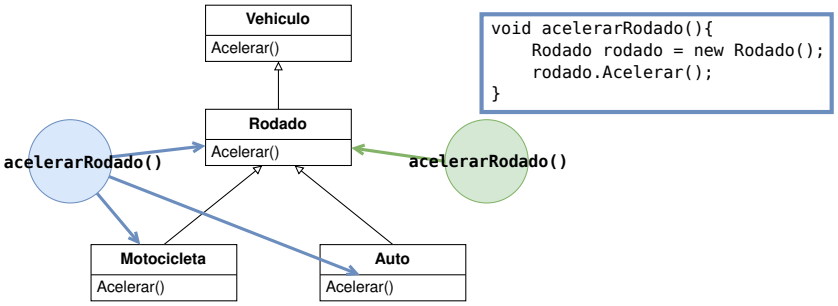
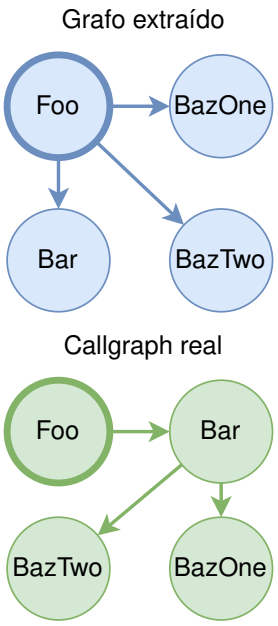


Figura 3. Esquema representativo de la información que Callcluster puede extraer del ejemplo de código de la figura. A la izquierda se representa la información que callcluster extrae, a la derecha la que extraería si respetara los principios de ocultamiento propios de la programación orientada a objetos. El mismo principio se utiliza para las llamadas a interfaces.

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 implica realizar un análisis estático más avanzado. El comportamiento de callcluster en este caso se describe en la figura 4.




```
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 4. Esquina superior izquierda: aproximación extraída por callcluster al analizar el fragmento de código. Esquina inferior izquierda: callgraph real del fragmento analizado. Derecha: framento de código analizado.

2.4. 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
- Complejidad ciclomática Basili
- Cantidad de líneas de código
- Cantidad de instrucciones contenidas en la función

Complejidad ciclomática Basili La complejidad ciclomática Basili consiste en una adaptación de la implementación realizada por Basili et. al. en [1] 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.

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:

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

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#.

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

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.

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.

2.5. 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.

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.

Roslyn Microsoft separó Visual Studio de las APIs de análisis 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#.

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.

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.

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. La interfaz acorde para CallCluster es libclang, que es la más estable.

2.6. 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 5 es un esquema de la arquitectura del sistema.

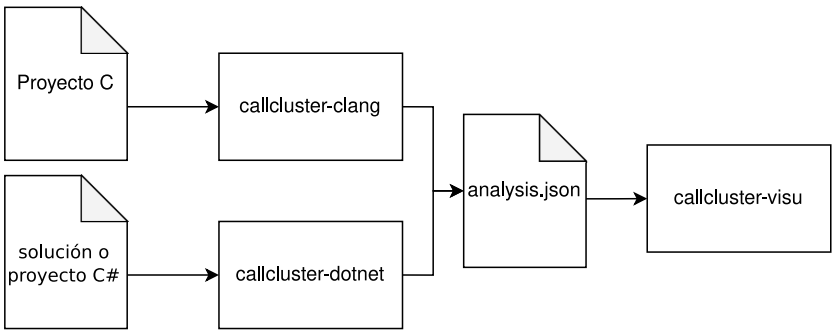


Figura 5. Arquitectura del sistema CallCluster

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 alcance inicial 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.

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.

- El análisis y la extracción pueden ser asincrónicos. Esto es un requisito para tener una experiencia de usuario ágil, 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.

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`.

Arquitectura de `callcluster-visu` El visualizador fue desarrollado usando el framework Quasar y electron.js, con el objetivo de que resulte amigable. Electronjs permite 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 una complejidad temporal superlineal 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 1 se detallan las responsabilidades del proceso de *Rendering* y del proceso *principal*.

Cuadro 1. Responsabilidad del proceso de rendering (dibujado) y del proceso principal en `callcluster-visu`.

Proceso de dibujado	Proceso principal
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.

3. Casos de estudio

3.1. LibUV

Libuv [2] es una librería C desarrollada para su uso en nodejs[3]. Tiene 38000 líneas de código en total, de las cuales 25000 son pruebas automáticas. 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". Callcluster permite observar que los archivos de la carpeta "unix"tienen una estructura disconexa, tal como observada en la figura 6. Callcluster, por medio de la aplicación del algoritmo de clustering, genera una organización en la cual cada cluster es más conexo. En la figura 7 se observa la estructura interna de uno de los clusters generados automaticamente por Callcluster, representativa de los demás.

3.2. Namespace HPack, parte de System.Net.Http del runtime de .Net

Se realizó un análisis restringido al namespace `System.Net.HPack` dentro del assembly `System.Net.Http.dll` del runtime de .NET[4] . 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 llevó adelante un análisis exhaustivo del espacio de nombres "HPack". Las figuras 8 y 9.

3.3. Arquitectura de PHP

El código fuente del intérprete del lenguaje PHP[5] incluye la librería stándard del lenguaje y un conjunto de extensiones de la misma. En la figura 10 se expone cómo Callcluster grafica la arquitectura de extensiones y la dependencia entre estas.

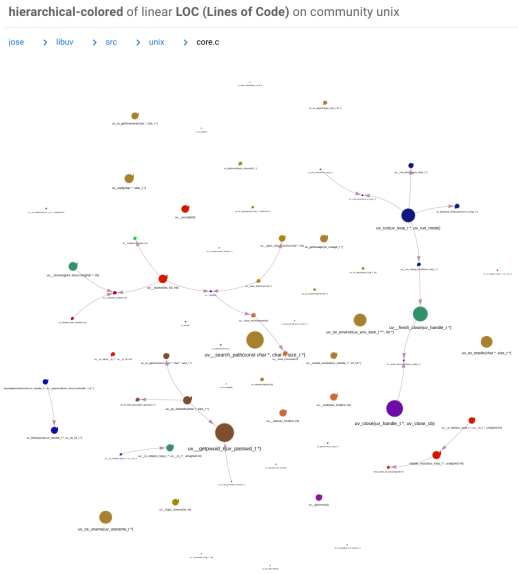


Figura 6. Callgraph del archivo `core.c`.

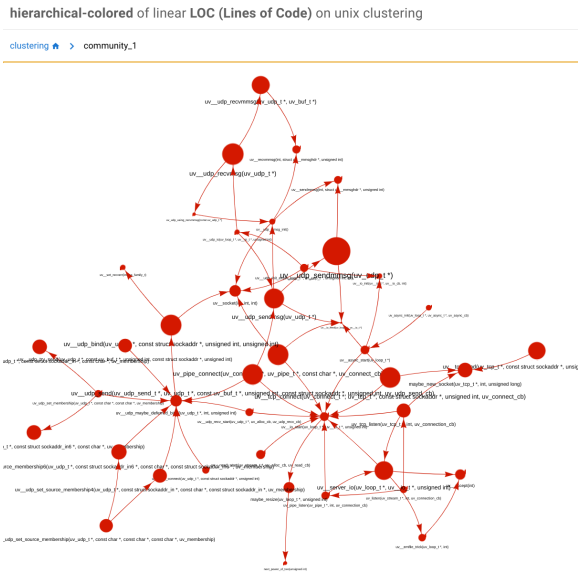


Figura 7. Callgraph de un cluster generado por el algoritmo de clustering.

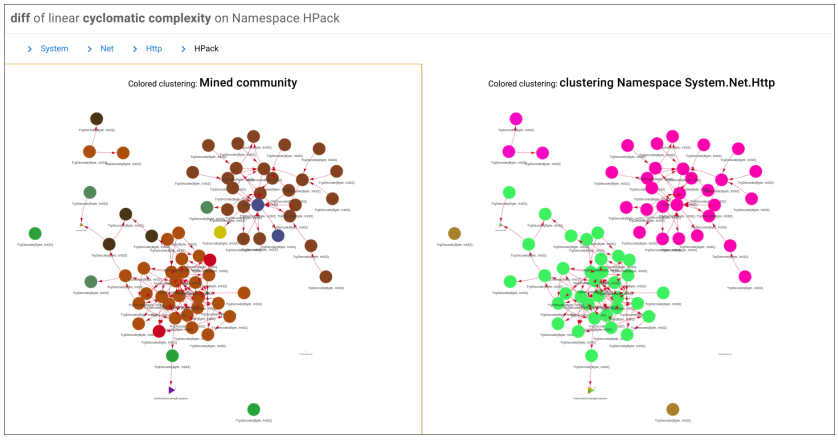


Figura 8. En la ilustración se representan las funciones incluidas en el sub-namespace `HPack` El algoritmo de clustering ubica los contenidos de HPack en dos clusters diferentes porque contiene dos componentes que no están conectados. La estructura descubierta por el algoritmo de clustering se condice sólo de forma aproximada con la estructura real (ver figura 9). 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.

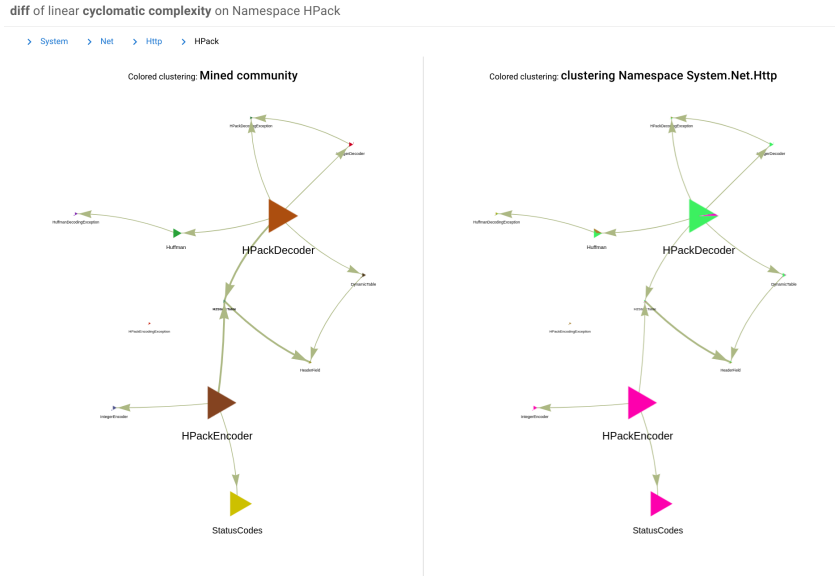


Figura 9. En la ilustración se representa el sub-namespace `HPack`, al igual que en la ilustración 8; pero se exhiben las clases del namespace. Este gráfico puede utilizarse para justificar la división de este namespace en dos sub-namespaces.

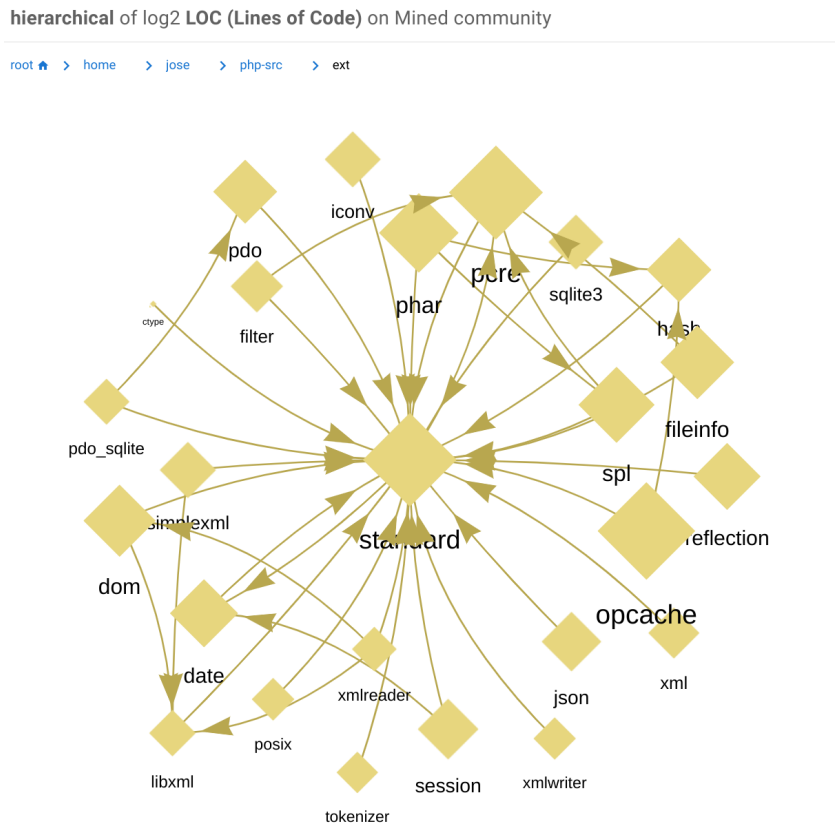


Figura 10. Dependencias entre la librería standard de PHP y las extensiones incluídas en el código fuente del intérprete.

4. 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"[13] 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 "casos de estudio", callcluster no puede automatizar el diseño de software sino que brinda información que puede llevar al diseñador a introducir una mejora.

Callcluster abre líneas de trabajo relacionadas a la extensión del visualizador. Además se puede profundizar en el desarrollo de herramientas matemáticas para asistir al diseño de software. Por sobre todas las cosas es necesario aplicar la herramienta a un proyecto real para comprobar y mejorar su efectividad.

El código fuente de callcluster puede conseguirse en [11]. El sitio web [12] contiene el manual de usuario y documentación técnica.

Referencias

1. 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.
2. Libuv `https://libuv.org/` acceso 22 05 2021
3. nodejs `https://nodejs.org/` acceso 22 05 2021
4. dotnet/runtime .NET is a cross-platform runtime for cloud, mobile, desktop, and IoT apps. `https://github.com/dotnet/runtime` acceso 22 05 2021
5. php/php-src The PHP Interpreter `https://github.com/php/php-src` acceso 22 05 2021
6. Moose `https://moosetechnology.org` acceso 22 05 2021
7. Bauhaus Project `https://en.wikipedia.org/wiki/Bauhaus_Project_(computing)` acceso 22 05 2021
8. ConQAT `https://en.wikipedia.org/wiki/ConQAT` acceso 22 05 2021
9. Softvis3d `https://softvis3d.com/` acceso 22 05 2021
10. Sourcetrail - The open-source cross-platform source explorer `https://www.sourcetrail.com/` acceso 22 05 2021
11. callcluster `https://github.com/callcluster` acceso 26 05 2021
12. Qué es (tl;dr) - Callcluster `https://callcluster.github.io/` acceso 26 05 2021
13. Frederick P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.