UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

EXTENSIÓN DE HERRAMIENTA PARA ANÁLISIS DE PROGRAMAS DESARROLLADOS EN SCRATCH

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

CINTHYA DE LOURDES ROBLES IBACACHE

PROFESORA GUÍA:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA RIVARA ZÚÑIGA
TOMAS VERA

SANTIAGO DE CHILE
2023

EXTENSIÓN DE HERRAMIENTA PARA ANÁLISIS DE PROGRAMAS DESARROLLADOS EN SCRATCH

En este trabajo de título se propone extender el sistema *Scratch Analyzer*, desarrollado por la ahora Ingeniera Civil en Computación María José Berger, correspondiente a una herramienta para la evaluación y análisis de proyectos desarrollados en Scratch en el contexto del curso impartido por investigadores del Departamento de Ciencias de la Computación de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile (DCC) para niños y niñas, con la finalidad de recabar información y determinar si este tipo de instancias genera un impacto positivo en la juventud, desarrollando sus rendimiento académico y capacidades relacionadas a la lógica por medio del acercamiento al pensamiento computacional, además de motivarlos a interesarse en el área de la programación.

Scratch Analyzer nace de la necesidad de automatizar y centralizar la revisión de proyectos desarrollados en Scratch por los niños y niñas que rinden el curso, puesto que la forma en que se está haciendo actualmente no es eficiente y requiere de bastante tiempo del cual los investigadores que imparten el curso no cuentan, sumado a la masiva (y creciente) cantidad de proyectos que se van acumulando con el paso de los años, impidiendo realizar los estudios correspondientes a los resultados con información actualizada.

La primera versión de la herramienta cuenta con la carga, procesamiento y análisis de múltiples proyectos a la vez, permitiendo clasificarlos por curso, permitiendo además que los evaluadores ingresen puntajes y comentarios en la rúbrica de evaluación dentro de la misma plataforma. Sin embargo, esta versión presenta *bugs* y casos de borde no abordados que son necesarios de corregir para mejorar el rendimiento de la herramienta, para que pueda entregar un valor real y ser utilizado por los investigadores. Para solucionar este problema, se realiza una nueva iteración en la que se identifican y resuelven *bugs* relacionados al procesamiento (en el intérprete o *parser* de la herramienta) y análisis de los programas (generación de CFGs y cálculo de métricas) y se realizan ajustes en la interfaz web para complementar los cambios y hacer la herramienta más intuitiva y simple de utilizar.

Para evaluar el valor de los cambios realizados en este trabajo de memoria se hace una validación funcional, en la que se comparan los resultados entregados por la herramienta en la versión previa a los cambios y la actual, de la cual se desprende que la nueva versión genera resultados que se alinean mejor con lo esperado de una revisión individual realizada por una persona, aunque aún hay detalles que requieren atención, los cuales se proponen solucionar en siguientes iteraciones.

I said "remember this feeling",
I pass the pictures around
of all the years that we stood there
on the sidelines, wishing for right now.

∼ Taylor Swift

Agradecimientos

Agradezco a mis padres, Fernando y Lilian, que me dieron todas las herramientas y el apoyo para llegar donde estoy. La firmeza y convicción de mi madre para inculcarme la importancia de estudiar y el valor de la responsabilidad. El apoyo incondicional de mi padre, que siempre ha estado para mí. Sin ellos nada de esto hubiera sido posible, los amo.

Agradezco a mis amistades y a mi pareja, que hicieron todo el proceso más ameno y me ayudaron a levantarme cuando sentía que no podía más. Hicieron mi paso por la universidad la época más linda e inolvidable de mi vida. Les deseo a todos un futuro brillante y lleno de felicidad.

Agradezco a Josué, que me acompañó durante gran parte del proceso para realizar este trabajo de memoria, me recibió en su casa y corazón con los brazos abiertos y me dio su apoyo en todo momento, me contuvo en mis bajos y celebró conmigo en mis altos. Fue mi rayo de sol cuando todo lo que veía era oscuridad.

Agradezco a todas las personas que me han acompañado en mi paso por la universidad, algunas desde el comienzo y otras solo recientemente, algunas continúan a mi lado y otras se fueron por distintos caminos, pero todas guardan un lugar especial en mi corazón.

Procuraré recordar este sentimiento y cada tanto revisitar en mi memoria o en imágenes todos los años que estuve deseando que llegara este momento, ahora levanto mi cabeza y la sostengo como en la imagen de un héroe en un libro de historia, pues es el fin de (casi) una década, pero también el comienzo de una era.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

Con el acelerado avance que ha presentado la computación durante las últimas décadas, ésta se ha logrado introducir en la cotidianidad, siendo parte importante en diversos aspectos de la sociedad. Así, resulta de gran utilidad e importancia que las personas tengan una noción sobre conceptos básicos de desarrollo tecnológico y computación desde una edad temprana, pues esto significaría no solo conocimiento en el área para las y los niños, sino que además desarrollarían habilidades relacionadas a la lógica que mejorarían su rendimiento escolar/académico en general [1, 2].

1.1. Antecedentes

Para satisfacer las necesidades de la sociedad moderna, el sistema educativo se ha ido adaptando añadiendo cursos de programación y desarrollo digital a su currículum tanto en escuelas como universidades. Estos cursos, sin embargo, priorizan la funcionalidad por sobre la calidad del software, provocando un vacío no menor en el conocimiento y aplicación de buenas prácticas en la programación [3].

Introducir y desarrollar el pensamiento computacional en niños y niñas desde temprana edad resulta ser beneficioso a largo plazo para su rendimiento académico tanto en el área de la computación como el resto, pues les da una base general para la resolución de problemas aplicable en distintos ámbitos [1].

Una forma para enseñar programación y pensamiento computacional en niños y niñas usualmente empleada debido a su simplicidad es la programación en bloques [4], la cual consiste en arrastrar y soltar bloques de instrucciones, haciendo la programación más interactiva e intuitiva. Un lenguaje de programación basado en la programación en bloques es Scratch, el cual está destinado principalmente a niños y niñas entre los 8 y 16 años (más información al respecto en la Sección 2.1).

El Departamento de Ciencias de la Computación de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile (DCC) contribuye a la aproximación de mentes jóvenes al mundo de la programación y pensamiento computacional por medio de un curso impartidos por académicos de la facultad (con apoyo de otros miembros, auxiliares y ayudantes, que en conjunto forman el equipo investigador), centrados enseñar a niños y niñas a desarrollar

proyectos en Scratch, evaluando sus resultados con fines investigativos [5, 6, 7]. Para evaluar el conocimiento y desempeño de los y las estudiantes, el equipo investigador debe revisar los proyectos desarrollados al finalizar el curso, para luego utilizar los datos extraídos de estas evaluaciones para la investigación que busca determinar si el curso, incluyendo los temas y metodologías empleadas, es efectivo en la educación temprana de niños y niñas, y si también éste motiva a los mismos a interesarse en el área de la programación.

Actualmente los investigadores del DCC que dictan este curso cuentan con la primera versión del prototipo de una herramienta que ayuda a la revisión y evaluación de los proyectos en Scratch, desarrollado por la ahora Ingeniera Civil en Computación, María José Berger, el año 2021: *Scratch Analyzer* [8].

1.2. Motivación

Para evaluar el desempeño de los y las estudiantes y así poder recopilar los datos necesarios para la investigación, el equipo de investigadores debe revisar sus proyectos, lo cual, debido a la falta de herramientas de automatización y centralización del análisis de proyectos realizados en Scratch, resulta en un retraso de 5 años (dato proporcionado por Jocelyn Simmonds, quien forma parte del equipo de investigación) en el análisis de los programas, generando un déficit de información y datos valiosos para la investigación [8]. En términos de cantidad de proyectos, se tiene que de cada taller se obtienen 30 aproximadamente, y cada año se realizan de 1 a 3 talleres, por lo que cada año se añaden entre 30 y 90 proyectos a la ya extensa lista.

Previo a *Scratch Analyzer*, los investigadores utilizaban dos herramientas independientes de análisis, las cuales serán descritas brevemente a continuación (más información en la Sección 2.2):

- La rúbrica de evaluación diseñada por los investigadores, la cual requiere una revisión individual y manual de los proyectos.
- La aplicación web Dr. Scratch [9], una herramienta que automatiza de manera simple el análisis de proyectos desarrollados en Scratch.

El problema con el uso de estas herramientas es que los investigadores debían emplear muchas horas de su ya limitado tiempo para revisar uno a uno los proyectos con la rúbrica (lo cual también deja expuesto el proceso a posibles errores humanos), sumado a que el análisis realizado por Dr. Scratch es inconsistente con el tamaño de los proyectos, además de favorecer ciertas prácticas por sobre otras, sin importar el contexto del programa a estudiar y ser poco transparente en la asignación de puntajes, pues no hay un desglose lo suficientemente específico que indique a los investigadores qué descontó puntaje y qué no. A lo anterior también se suma el manejo de datos masivos debido a la gran cantidad de proyectos que deben ser evaluados y analizados, junto con la falta de un espacio virtual en el cual los datos recabados por cada herramienta se unan de manera automática para facilitar su estudio [8]. Gran parte de estas problemáticas están siendo atendidas por *Scratch Analyzer*.

Scratch Analyzer corresponde al prototipo de una herramienta que será empleada por los investigadores del DCC que dictan el curso para automatizar la evaluación de los proyectos

realizados por los niños y niñas. Esta herramienta cuenta con funcionalidades que abarcan la administración de cursos, subida y análisis de múltiples proyectos a la vez y la generación y representación gráfica de los Grafos de Control de Flujo (CFG, por sus siglas en inglés) [10] que describen el comportamiento de los proyectos. Con esto se condensa y automatiza el trabajo de los investigadores a la hora de revisar, pues les permite realizar el análisis y evaluación en una sola plataforma de una manera más rápida y eficiente.

1.3. Objetivos

1.3.1. Objetivo general

El objetivo general de este trabajo de memoria es hacer más robusto el parseo de los proyectos desarrollados en Scratch y su representación como CFG a través de la herramienta *Scratch Analyzer*.

1.3.2. Objetivos específicos

Con el fin de alcanzar el objetivo general presentado anteriormente, se propone cumplir con los siguientes objetivos específicos:

1. Identificar y corregir los casos borde donde el *parser* de programas desarrollados en Scratch actualmente se cae.

2. Refactorizar la generación de CFGs, para separar esto del parseo de los proyectos en Scratch.

3. Mejorar la representación visual de los CFGs generados, enfatizando los eventos que gatillan acciones para cada actor. Esto con el fin de que sean más comprensibles.

1.4. Solución propuesta y metodología

Para definir qué tareas realizar y cómo abordar los problemas que presentaba la primera versión de la herramienta, primero se realizó una exploración con programas de ejemplo, los cuales fueron analizados manualmente y posteriormente cargados en la herramienta para estudiar los resultados entregados. Al igual que en la primera iteración de la herramienta [8], fue necesario realizar ingeniería inversa con los proyectos para determinar la forma en que se procesan los datos y cómo estructurarlos para simplificar el cálculo de métricas y la generación de CFGs.

Tras la exploración se determinó que se debía refactorizar parte del código, específicamente lo que abarca el procesamiento de los datos (*parser*) y el posterior cálculo de métricas para el análisis de los proyectos. También se definieron ajustes para la interfaz web con fines tanto estéticos como de funcionalidad.

Con respecto la organización de plazos y tareas, se agendaron reuniones remotas cada dos semanas con la profesora guía para comunicar dudas y recibir retroalimentación de los avances realizados. En conjunto con lo anterior, se mantuvo una bitácora con todas las minutas

de reuniones, observaciones, avances y dudas que surgieron durante el desarrollo del presente trabajo de memoria.

La base del proyecto se encuentra en Github, en 2 repositorios separados en *front-end* y *back-end*. Al tratarse de la extensión de un sistema (correspondiente a una aplicación web), las tecnologías utilizadas para el desarrollo se mantendrán y corresponden a las siguientes: React JS 18.2.0 [11] para el *front-end* y Python 3.8.14 – más específicamente Django REST framework 3.1.1 [12] – para el *back-end*. Se usará el editor de texto VSCode [13] en una máquina virtual con sistema operativo Ubuntu 20.04 [14] para trabajar.

1.5. Estructura del documento

Primero se encuentra el marco teórico (Capítulo 2), en donde se describen los conceptos que serán empleados en el presente documento. Luego está el diseño y análisis de la solución (Capítulo 3), donde se definen el problema y las tareas a realizar para solucionarlo. Posteriormente está la implementación (Capítulo 4), en donde se describe el trabajo realizado, seguido de la validación (Capítulo 5), en donde se estudia y discute el valor que otorgan los cambios realizados a la herramienta. Finalmente se encuentran las conclusiones (Capítulo 6) y la bibliografía utilizada, seguidas de los anexos.

Capítulo 2

Marco teórico

Durante la realización de este trabajo de memoria se utilizaron diversos conceptos y definiciones que serán descritas a continuación para dar un mayor contexto para la información entregada en los siguientes capítulos.

2.1. Scratch

Como se mencionó anteriormente, Scratch es un lenguaje de programación en bloques destinado para niños y niñas entre los 8 y 16 años, con el cual se pueden crear historias digitales, juegos y animaciones [15]. Es de uso gratuito está disponible para uso *online* en https://scratch.mit.edu/ (haciendo *click* en la pestaña "Crear") o también se puede descargar la aplicación de escritorio (sin necesidad de conexión a Internet) en https://scratch.mit.edu/download. Actualmente, Scratch se encuentra en su tercera versión como lenguaje de programación (exportable a formato `.sb3`) y existe retrocompatibilidad con las versiones anteriores.

La interfaz de la plataforma web de Scratch, que se puede observar en la Figura 2.1, tiene 2 secciones principales:

1. En la sección de la izquierda -en la figura- hay 3 pestañas: Code, Costumes y Sounds.
   a) En Code, en la parte más a la izquierda, se encuentran los bloques de instrucciones disponibles para el personaje seleccionado (que se puede observar en la esquina superior derecha del recuadro en blanco), divididas por tipo: Motion, Looks, Sound, Events, Control, Sensing, Operators, Variables y My Blocks (descripciones breves para cada tipo se pueden observar en la Tabla 2.1). En la parte más a la derecha se encuentra el espacio de trabajo en el que se pueden ensamblar los bloques de instrucciones para el personaje seleccionado.
   b) En Costumes (Figura A.1.1) se encuentra la configuración estética del personaje seleccionado.
   c) En Sound (Figura A.1.2) se encuentra la configuración de los sonidos asociados al personaje seleccionado.
2. En la sección de la derecha se pueden observar dos recuadros, además de botones destinados para iniciar/detener el programa y para ajustar la vista de la interfaz.

a) El de la parte superior es el recuadro de ejecución del programa/proyecto, en el cual se puede observar qué ocurre en él.

b) El de la parte inferior izquierda se encuentra el recuadro de personajes (Sprite) del programa con sus características, mientras que a la derecha están los escenarios (Stage) del programa.

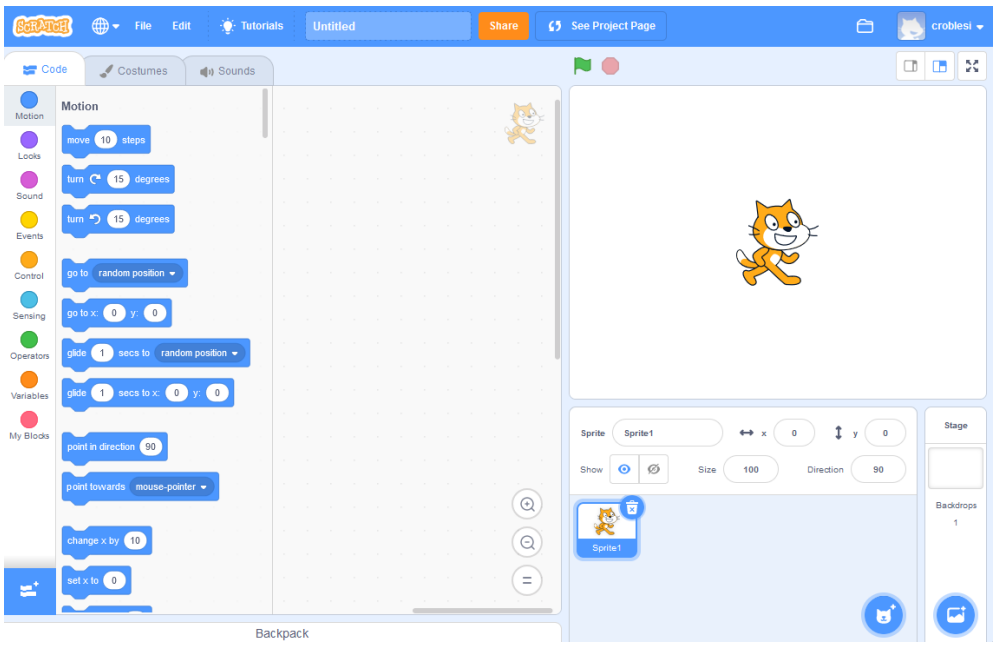

Figura 2.1: Interfaz web de Scratch

La estructura de los programas es prácticamente libre, con la condición de que, para que una secuencia de bloques asociada a un personaje se ejecute efectivamente, ésta debe comenzar con un bloque del tipo Events, pues de no ser así la secuencia nunca será ejecutada y quedará como código inalcanzable dentro del programa. Los bloques se pueden ensamblar de dos maneras: secuencial y anidada. En base a esto un programa puede ser tan simple o complejo de acuerdo a la cantidad de bloques y la forma de ensamblaje que el usuario utiliza. La ejecución de los bloques es secuencial, con excepción de los ciclos para los bloques anidados (generalmente del tipo Control), los cuales pueden repetir la ejecución de los bloques anidados según lo que esté indicado en las condiciones -si es que existen- del ciclo. En la Figura 2.2 se pueden apreciar dos ejemplos de ensamblaje de bloques, una secuencial (izquierda) y una anidada (derecha), ambos comienzan con un bloque de tipo Evento, pero el secuencial es seguido por instrucciones del tipo Variables, Looks, Motion y Sound, los cuales se ejecutarán uno tras otro en orden de aparición, mientras que el anidado es seguido por bloques de tipo Looks, Motion y un bloque de tipo Control `if-then` que anida bloques de tipo Motion, Looks y Variables.

Cada personaje cuenta con un ensamblaje de bloques, y para cada personaje hay un espacio de trabajo separado en la plataforma. Como se aprecia en la Figura 2.2, el ensamblaje de la izquierda está asociado al Sprite *Cat* y el de la derecha está asociado al Sprite *Dino*, y para cada uno solo se ejecutan los bloques que están asociados a ellos.

Tabla 2.1: Resumen de las categorías de bloques de instrucciones en Scratch

Categoría	Descripción
Motion	Movimiento del personaje
Looks	Diálogos, cambio de apariencia de personajes o escenarios
Sound	Reproducción de sonidos
Events	Captura de eventos que gatillan una secuencia de instrucciones
Control	Condicionales y/o ciclos (`wait`, `repeat`, `forever`, `if-then`, etc.)
Sensing	Interacciones entre elementos o de elementos con el dispositivo
Operators	Operaciones matemáticas, lógicas y/o de cadenas de texto
Variables	Creación y manejo de variables del programa
My Blocks	Creación de bloques personalizados

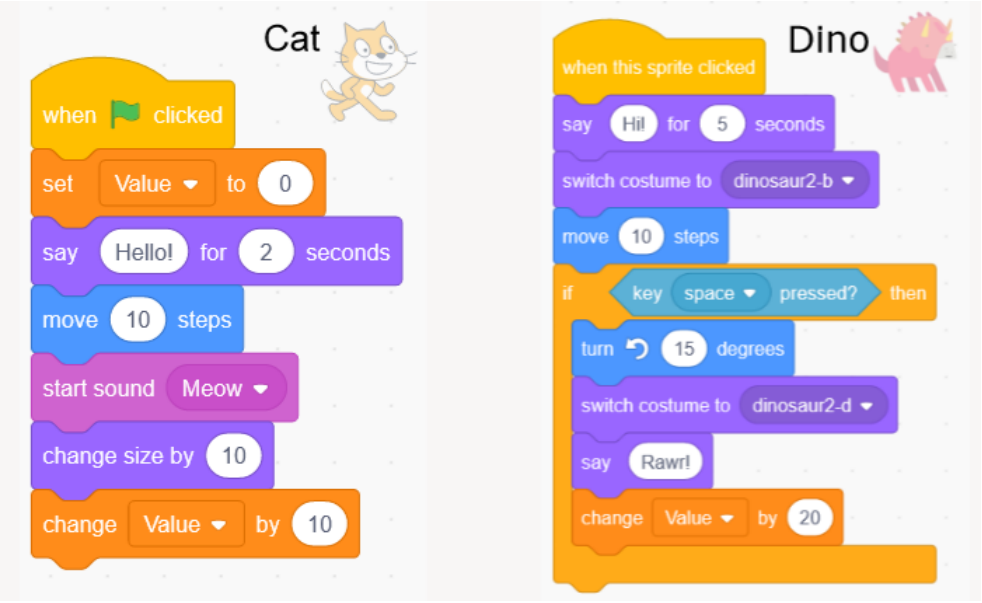

Figura 2.2: Ejemplos de ensamblaje de bloques en Scratch

La plataforma permite descargar el proyecto (`file` > `save to your computer`), generando un archivo en formato `.sb3`. Los estudiantes de los cursos de Scratch impartidos por los investigadores del DCC deben entregar este archivo resultante que contenga su proyecto para ser evaluado. Este formato de archivo no puede ser abierto de forma directa para ejecutarse, sino que debe subirse a la plataforma. Sin embargo, el archivo puede convertirse manualmente al formato `.zip` (añadiendo la extensión al nombre desde el gestor de archivos) para ver su contenido (ver Figura 2.4, que muestra los archivos que se generan al descargar el Ejemplo 0), el cual se compone de las imágenes y archivos de audio utilizados en el proyecto, además de un archivo con el nombre `project.json` que contiene la codificación en formato `json` de todos los elementos y bloques que conforman el proyecto. En el Código 2.1 se puede observar

parte del archivo `project.json` de un proyecto de ejemplo (Ejemplo 0) asociado al actor Cat. En la Figura 2.3 se puede ver el Ejemplo 0 en la interfaz web de Scratch con el código asociado al actor Cat.

Ejemplo 0

Enlace de acceso: https://scratch.mit.edu/projects/748167123.

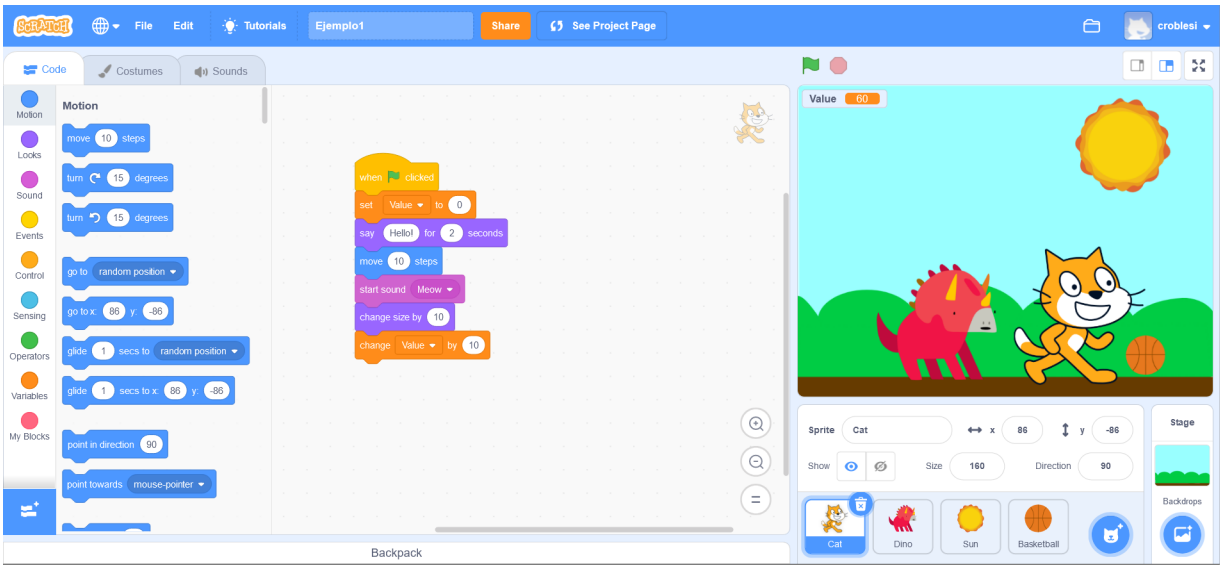

Figura 2.3: Interfaz de la plataforma Scratch con programa de ejemplo 0

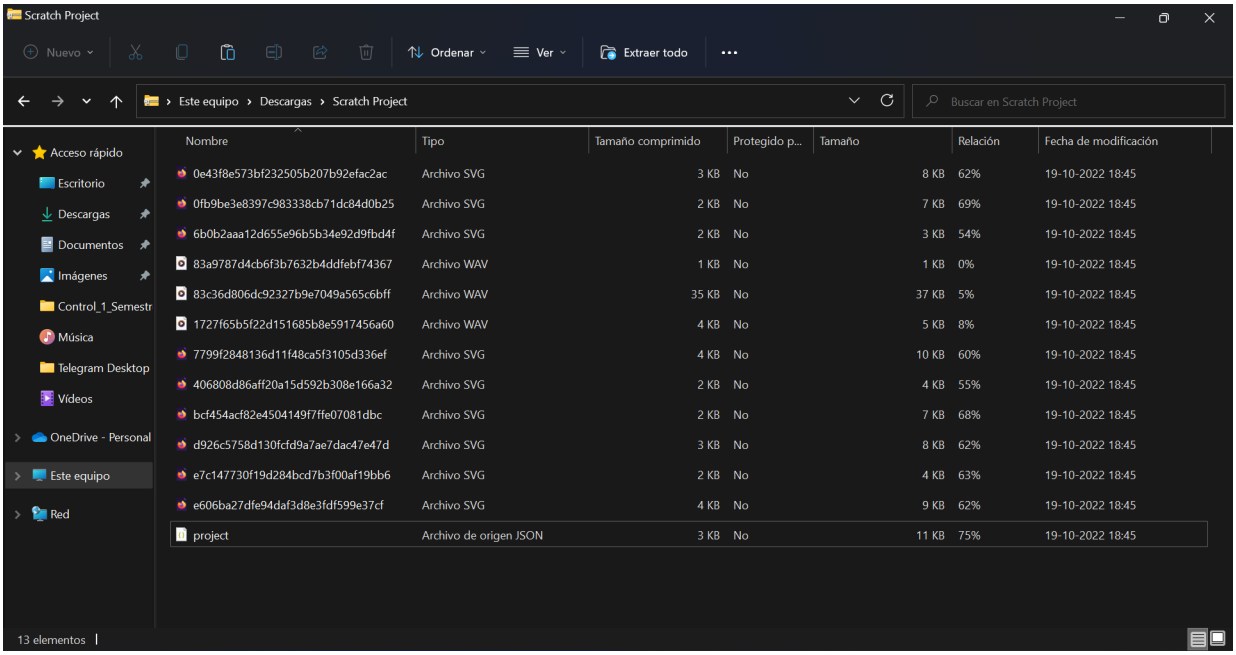

Figura 2.4: Contenido de archivo `.sb3` del programa de ejemplo 0

Código 2.1: project.json ejemplo 0 (extracto)

```
{
"targets": [
    ...
    {
        "isStage": false,
        "name": "Cat",
        "variables": {},
        "lists": {},
        "broadcasts": {},
        "blocks": {...},
        "comments": {},
        "currentCostume": 0,
        "costumes": [...],
        "sounds": [...],
        ...
    },
    ...
],
"monitors": [...],
"extensions": [],
"meta": {...}
}
```

2.2. Sistema de evaluación actual

Previo a la implementación de la primera versión de *Scratch Analyzer*, los investigadores del DCC utilizaban una combinación de dos herramientas para la evaluación y análisis de los proyectos desarrollados en Scratch por los y las estudiantes: Dr. Scratch y una evaluación manual empleando una rúbrica diseñada por los investigadores. A continuación se proporcionará más información con respecto a ambos métodos.

2.2.1. Dr. Scratch

Corresponde a una aplicación web en la que se pueden evaluar proyectos desarrollados en Scratch según varios aspectos del pensamiento computacional, permitiendo a docentes y estudiantes a automatizar el análisis de proyectos y motivar el uso de buenas prácticas en la programación, detectando errores generales y entregando retroalimentación mediante puntuaciones y clasificaciones para distintos criterios de evaluación. En la Figura 2.5 se puede ver la página de inicio de la aplicación, en la cual se dan dos opciones para cargar un proyecto: a través de su enlace a la página de Scratch o subiendo el archivo `.sb3` que se obtiene al descargar el proyecto al ordenador.

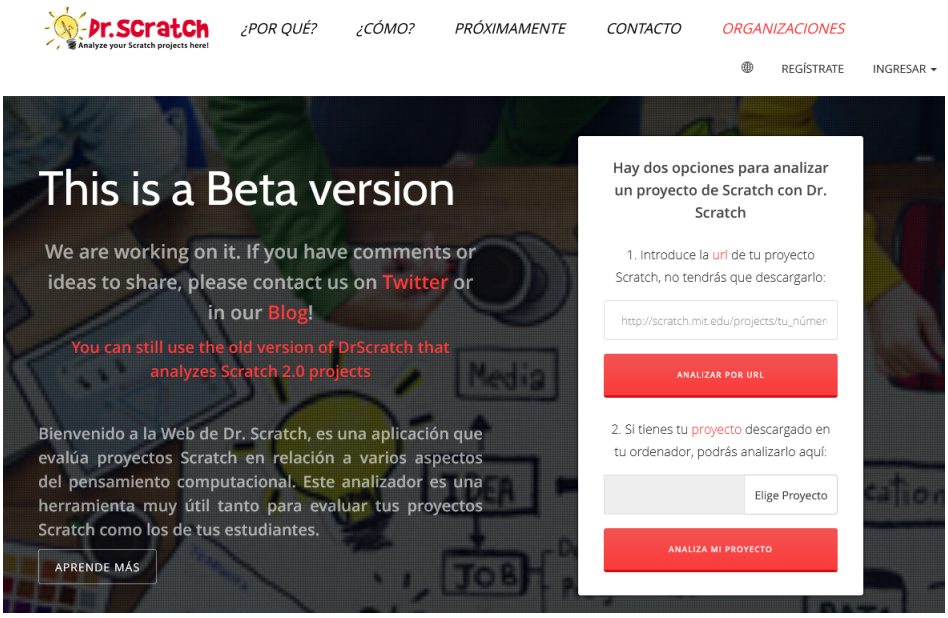

Figura 2.5: Página de inicio Dr. Scratch

Una vez tiene acceso al proyecto, la plataforma lo analiza y entrega puntuaciones de 0 a 3 para distintos ítems: paralelismo, representación de la información, pensamiento lógico, interactividad con el usuario, control de flujo, abstracción y sincronización, optando a un máximo de 21 puntos totales. Dependiendo de la puntuación que reciba cada ítem, éste será clasificado entre 4 categorías: *Ninguno* (0 puntos), *Principiante* (1 punto), *Desarrollador* (2 puntos) y *Competente* (3 puntos) [16]. El proyecto en su conjunto es a su vez clasificado entre las categorías *Básico*, *Medio* y *Alto* [17]. Además, identifica y reporta la presencia de malas prácticas de programación, como por ejemplo duplicación de código, código inalcanzable, entre otras. En la Figura 2.6 se puede observar un ejemplo de resultado de análisis del Ejemplo 0 en Dr. Scratch.

Figura 2.6: Resultado de análisis de un programa en Dr. Scratch

2.2.2. Rúbrica de evaluación

Como se menciona al inicio de esta sección, los investigadores diseñaron una rúbrica de evaluación (que se puede ver en la Figura 2.7) que los ayuda a analizar proyectos de forma manual bajo los mismos 4 niveles de competencia que emplea Dr. Scratch, pero para distintos criterios relacionados con el pensamiento computacional y buenas prácticas de programación.

Este tipo de evaluación, a pesar de ser más certera con respecto a lo que se quiere evaluar, también posee limitaciones que reducen su efectividad, principalmente la necesidad de emplear muchas horas de trabajo y el sesgo que pueden tener los distintos evaluadores a la hora de revisar un proyecto, pudiendo generar discordancias.

Table 1: Rubric for evaluating student projects

	None (0)	Beginner (1)	Developing (2)	Competent (3)
(A) % of sprites that only control themselves	[0%, 10%]	(10%, 40%]	(40%, 70%]	(70%, 100%]
(B) % of blocks that have a single purpose	[0%, 10%]	(10%, 40%]	(40%, 70%]	(70%, 100%]
(C) documentation	0 comments	1-2 comments	3-6 comments	> 6 comments
(D) % of reachable blocks	[0%, 10%]	(10%, 40%]	(40%, 70%]	(70%, 100%]
(E) % of items with appropriate names	[0%, 10%]	(10%, 40%]	(40%, 70%]	(70%, 100%]
(F) % of superficial changes w.r.t sample projects	(70%, 100%]	(40%, 70%]	(10%, 40%]	[0%, 10%]
(G) project novelty	the expected behavior is not clear	quite similar to a sample project	extends a sample project in a novel way	quite different from the sample projects, or integrates 2 or more sample projects

Figura 2.7: Rúbrica de evaluación para proyectos de los cursos de Scratch del DCC [6]

2.3. Scratch Analyzer v.1

La primera versión de la herramienta unifica los procesos principales del sistema de evaluación actual, incluyendo la carga y almacenamiento de los proyectos dentro de la misma plataforma, ordenados según el semestre que les corresponde.

Esta versión fue diseñada para ser utilizada localmente, por lo que no contempla la implementación de usuarios y autenticación, esto debido a que se priorizaron las funcionalidades de unificación y automatización. Se definieron tres ejes principales, en base a los cuales fue diseñada la herramienta (ver Figura 2.8, descritos más en detalle en la Sección 3.2.1). El procesamiento contempla el *parser* o intérprete de Scratch como lenguaje de programación, traduciendo las instrucciones, actores y escenarios de los programas a una estructura de datos que pueda ser manejada con Python, y el algoritmo de generación de CFGs. La parte de análisis y métricas trabaja con la información ya procesada y calcula indicadores que permiten evaluar la calidad de los proyectos. Finalmente, en resultados y reporte se condensan los datos obtenidos en el análisis y se muestran de forma gráfica y/o en tablas al usuario, con la posibilidad de descargar archivos `.csv` con parte de la información mostrada.

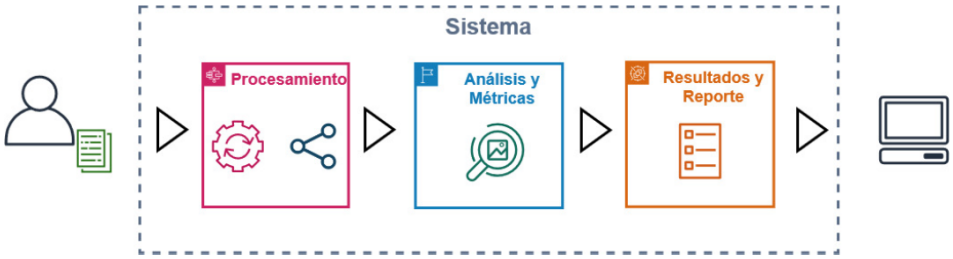

Figura 2.8: Diagrama de la solución propuesta Scratch Analyzer v.1

El *parser* fue implementado por medio de ingeniería inversa, estudiando la estructura de programas de ejemplo para detectar patrones y así generar una estructura de datos adecuada para el manejo de la información. Esto fue realizado debido a que el lenguaje Scratch no contaba con la documentación adecuada ni un *parser* que pudiera interpretar automáticamente las instrucciones [8], por lo que se tuvo que priorizar este eje por sobre el análisis, puesto que no se podía realizar el análisis sin primero traducir los programas a una estructura manejable. Esto presentó un desafío, pues los programas exportados presentaban más de un bloque para cada instrucción, por lo que se tuvo que estudiar qué instrucciones generaban más bloques de los esperados para poder determinar qué debía hacer el *parser* en estos casos. Para no perder información que podría resultar importante para el análisis posterior, se optó por guardar todos los bloques, clasificándolos como *primarios* y *no primarios* en la base de datos, usando finalmente solo los primarios para el conteo de instrucciones de los programas.

Otro desafío que se presentó en la implementación de la primera versión de la herramienta fue el algoritmo de generación de CFGs, puesto que existían más bloques que instrucciones de programa, algunos de los cuales correspondían a variables u opciones que no representaban acciones relevantes, por lo que el marcar bloques como *primarios* o *no primarios* en el *parser* no era suficiente para filtrar los bloques que resultaban útiles para mostrar en los CFGs y los que no. Por temas de plazo, se decidió no modificar el parser y generar y mostrar los CFGs de todas formas, pero no utilizarlos en el cálculo de métricas para el análisis de los proyectos, debido a que los grafos generados no representaban fielmente la ejecución del programa.

Con respecto al análisis y métricas, los cálculos se realizaron con los bloques marcados como primarios por el *parser*, y éstos consisten principalmente en el conteo de bloques totales y por actor, clasificados según los distintos tipos de bloques (en la Tabla 2.1 se pueden ver los nombres y descripción de cada tipo), excluyendo los bloques personalizados (denominados My Blocks en la plataforma de Scratch y `procedures` en el JSON del proyecto una vez exportado). Otras métricas calculadas son la cantidad de grafos generados, interacciones con el usuario (User Interaction) [18], bloques y grafos inalcazables (que nunca se ejecutan) y la complejidad ciclomática [19]. Todos estos valores son mostrados en forma de tablas y gráficos en la vista de cada proyecto (ver Figura 2.9), incluyendo la rúbrica de evaluación empleada por los evaluadores (más información en la Sección 2.2.2).

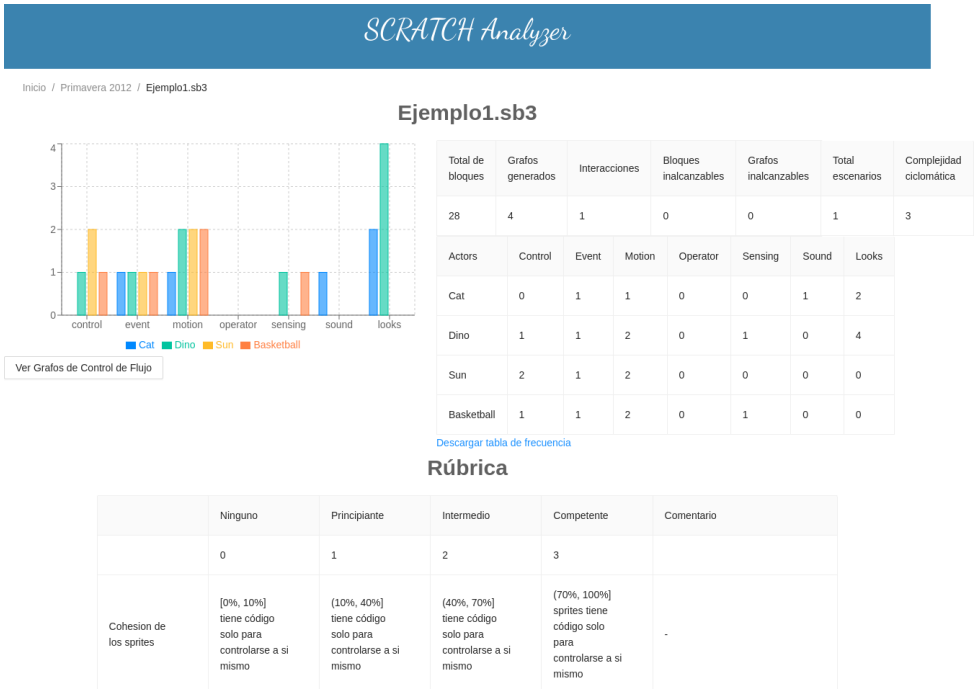

Figura 2.9: Vista de proyecto en Scratch Analyzer v.1

2.4. Grafos de control de flujo, métricas y análisis

Existen diversas métricas y análisis en base a los cuales se pueden evaluar programas y proyectos. El área de interés para este trabajo de título es la medición del desarrollo de habilidades asociadas al pensamiento computacional. Las métricas se aplican principalmente sobre programas representados en Grafos de Control de Flujo (CFG). En la Figura 2.10 se pueden observar ejemplos de CFGs que representan gráficamente los distintos caminos que puede seguir un programa de acuerdo a las instrucciones que lo conforman. Cada nodo representa una instrucción en el código, que en programas desarrollados en Scratch correspondería a un bloque. Los nodos que generan bifurcaciones en el grafo corresponden a nodos de decisión, que en Scratch es equivalente a los bloques tipo Control con condición.

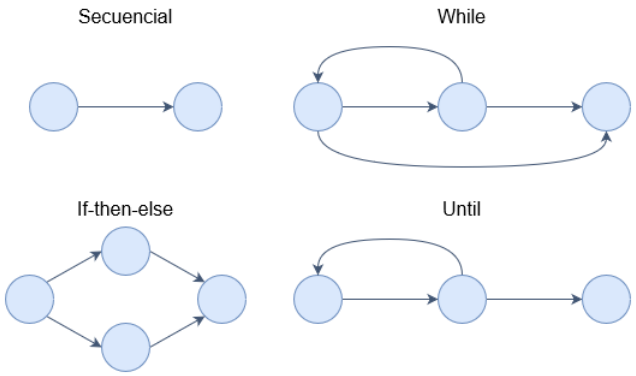

Figura 2.10: Ejemplo de CFGs para distintos tipos de intrucciones

Una métrica relevante y que se asocia directamente al uso de CFGs para analizar un programa es la Complejidad Ciclomática (CC) definida por McCabe [19], la cual mide cuantitativamente la cantidad de caminos linealmente independientes presentes en un CFG sin necesidad de contar únicamente la cantidad de `if`s presentes en el programa que representa. El cálculo de esta métrica se realiza a través de la fórmula $E - N + 2P$ donde, dado un grafo $G = (N, E)$, E corresponde al número de arcos, N al número de nodos y P al número de componentes conexas del grafo. McCabe también demostró que la CC de un programa con sólo un nodo de entrada y un nodo de salida es igual a la cantidad de nodos de decisión presentes en el programa (π) más 1, es decir, $\pi + 1$.

Con respecto a análisis se tiene el Análisis de Flujo de Datos (*Data Flow Analysis*) [20], con el cual se deriva información sobre el comportamiento dinámico de un programa basándose en el código estático del mismo, por medio de su CFG. A partir del análisis de flujo de datos, se pueden calcular -de una forma aproximada- distintas propiedades de un programa, como:

- **Reaching definition (alcance de definición):** una definición de una variable v alcanza un nodo n si existe un camino entre la definición y n en el cual v no es redefinido.
- **Liveness analysis (análisis de vida de una variable):** una variable está "viva" en un determinado punto del programa si es usada al menos una vez en el futuro.
- **Available expression (expresión disponible):** una expresión $x + y$ está disponible en un nodo n si todo camino desde el nodo inicial a n evalúa $x + y$ y las variables x e y no son redefinidas. Esto significa que si la expresión está disponible en un nodo no debería ser recalculada para usarse en ese nodo.

El cálculo y estudio de estas propiedades son útiles en el contexto de evaluación y análisis de proyectos en Scratch debido a que permiten identificar errores en el código que no necesariamente se podrían obtener a simple vista (en los bloques o ejecución del programa). Las propiedades descritas corresponden a las que son relevantes para este trabajo de título, pues existen otras como *Definite assignment analysis* [21] y *Constant folding and propagation* [22], que son útiles para el análisis de programas desarrollados en lenguajes de programación clásicos como *C*.

2.5. Resumen

En este capítulo se definieron conceptos que serán empleados a lo largo del presente documento. Primero tenemos Scratch, que es a la vez una plataforma y un lenguaje de programación basado en bloques, con el cual se desarrollan los proyectos que la herramienta debe procesar y analizar. También se describe el sistema de evaluación actual, que cuenta con dos partes: cargar los programas (uno a uno) en la plataforma de análisis Dr. Scratch y una evaluación a mano empleando una rúbrica de evaluación. Posteriormente se describe la primera versión de la herramienta y las funcionalidades que cumple, correspondientes a la unificación y automatización del proceso de evaluación de proyectos desarrollados en Scratch. Finalmente, se describen los conceptos relacionados a los grafos de control de flujo y métricas asociados al análisis de proyectos.

Capítulo 3

Análisis y diseño de la solución

En el presente capítulo se estudian los problemas y limitaciones que presenta la primera versión de la herramienta con el fin de determinar los cambios necesarios para solucionarlos, proporcionando propuestas para la implementación.

3.1. Definición del problema

La primera versión de la herramienta presenta limitaciones con respecto al procesamiento de los proyectos, lo cual genera problemas a la hora de cargar algunos programas que cuentan con características no consideradas en el *parser*, impidiendo que éstos se guarden correctamente (o no los guarda, dejando la herramienta en un *loop* que solo se puede finalizar refrescando la ventana). Esto además genera inconsistencias en el cálculo de métricas útiles para el análisis de los proyectos, pues se pierde información importante de los programas.

En la etapa de exploración realizada entre finales del Semestre Primavera 2022 y comienzos del Semestre Otoño 2023 se estudió el código base del *parser* (cuya ruta en el repositorio corresponde a `scratch-analizer-back/scratchAnalizer/core/views.py`) y se encontraron problemas de legibilidad, por lo que el estudio debió ser complementado con la lectura del informe de memoria de Berger [8] y pruebas con programas de ejemplo para definir correctamente el comportamiento de la plataforma, añadiendo comentarios en el código para facilitar el trabajo posterior.

A continuación se presentarán los programas utilizados para el estudio y se describirán los resultados de la exploración realizada, la cual contribuyó a la definición de las tareas realizadas para este trabajo de memoria.

3.1.1. Exploración con programas de ejemplo

Para comprender mejor la estructura de los datos que la herramienta debe procesar, se crearon múltiples programas de ejemplo de distinta complejidad.

Programas de ejemplo

A continuación se listarán los enlaces de acceso de otros proyectos de ejemplo utilizados para el diseño e implementación de la solución:

- **Ejemplo 1:** https://scratch.mit.edu/projects/748167123
- **Ejemplo 2:** https://scratch.mit.edu/projects/823818160
- **Ejemplo 3:** https://scratch.mit.edu/projects/829145118
- **Ejemplo 4:** https://scratch.mit.edu/projects/829177525
- **Ejemplo 5:** https://scratch.mit.edu/projects/829320256

Estudio de los JSON

Para algunos de los ejemplos se contabilizaron la cantidad de bloques por actor presentes en el programa tanto en la plataforma de Scratch como en el JSON generado al exportarlo. A continuación se muestran los datos recopilados con respecto a la cantidad de bloques por actor para un programa de ejemplo (Ejemplo 1).

Tabla 3.1: Cantidad de bloques por actor para el Ejemplo 1

	Tipo	Cat	Dino	Sun	Basketball
Scratch	Secuenciales	7	5	2	4
	Anidados	0	4	3	1
	Condicionales	0	1	0	1
	Bloques Scratch	7	10	5	6
	Motion	1	2	2	2
	Looks	2	4	0	0
	Sound	1	0	0	0
	Events	1	1	1	1
	Control	0	1	2	1
	Sensing	0	1	0	1
	Operators	0	0	0	0
	Variables	2	1	0	1
	My Blocks	0	0	0	0
JSON	Bloques JSON	8	13	5	7

A partir de este estudio se determinó lo siguiente:

- Los bloques Sensing pueden corresponder a acciones, variables o condiciones.
- Los bloques Sensing no tienen *next* (es *null*).
- Los bloques Sensing, Sound y algunos Looks (correspondientes a cambios de *costume* para el actor) tienen 2 bloques: uno con la acción y el otro con las opciones, los que contienen las opciones serán denominados *duplicados* desde este punto.
- Los bloques que tienen duplicado se relacionan entre sí a través de las llaves *input* y *parent*, no por *next*, como ocurre con las secuencias de bloques simples.
- Los bloques en el JSON no necesariamente están en el orden de ejecución.

- Los bloques tipo My Blocks son `procedures_xxxx` en el JSON. Éstos también cuentan con duplicados: primero está `procedures_definition`, que en su llave *next* señala al bloque que inicia la secuencia de bloques que lo componen, y luego está `procedures_call`, que señala a `procedures_definition` como *next*.
- No todos los bloques tipo Control presentan anidación, pues también existen los `control_wait`, que representan una acción simple y secuencial en el programa.

Definición de CFGs

Se hicieron CFGs a mano para los programas de ejemplo con el fin de estudiar y comparar con el algoritmo que emplea la herramienta para generarlos a partir del parseo del JSON. A continuación se pueden observar los CFGs esperados para el Ejemplo 1:

Para el actor Cat sólo se tiene código secuencial, y el CFG esperado se puede observar en la Figura 3.1.

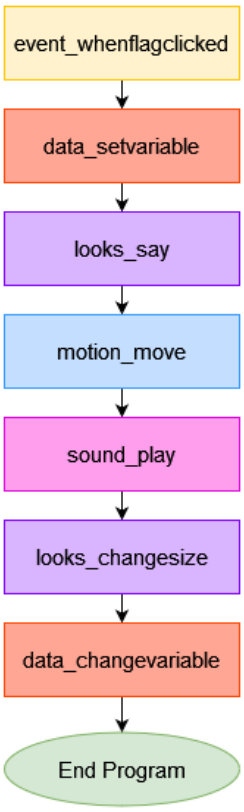

Figura 3.1: CFG esperado para el código asociado al actor Cat en el Ejemplo 1

Para Dino se tiene un bloque/nodo condicional `if`, para lo cual se plantearon dos opciones (ver Figura 3.2): incluir o no la condición, que en este caso corresponde a un bloque tipo Sensing. Finalmente se optó por mantener la estructura del grafo de la derecha, es decir, sin incluir el bloque de condición.

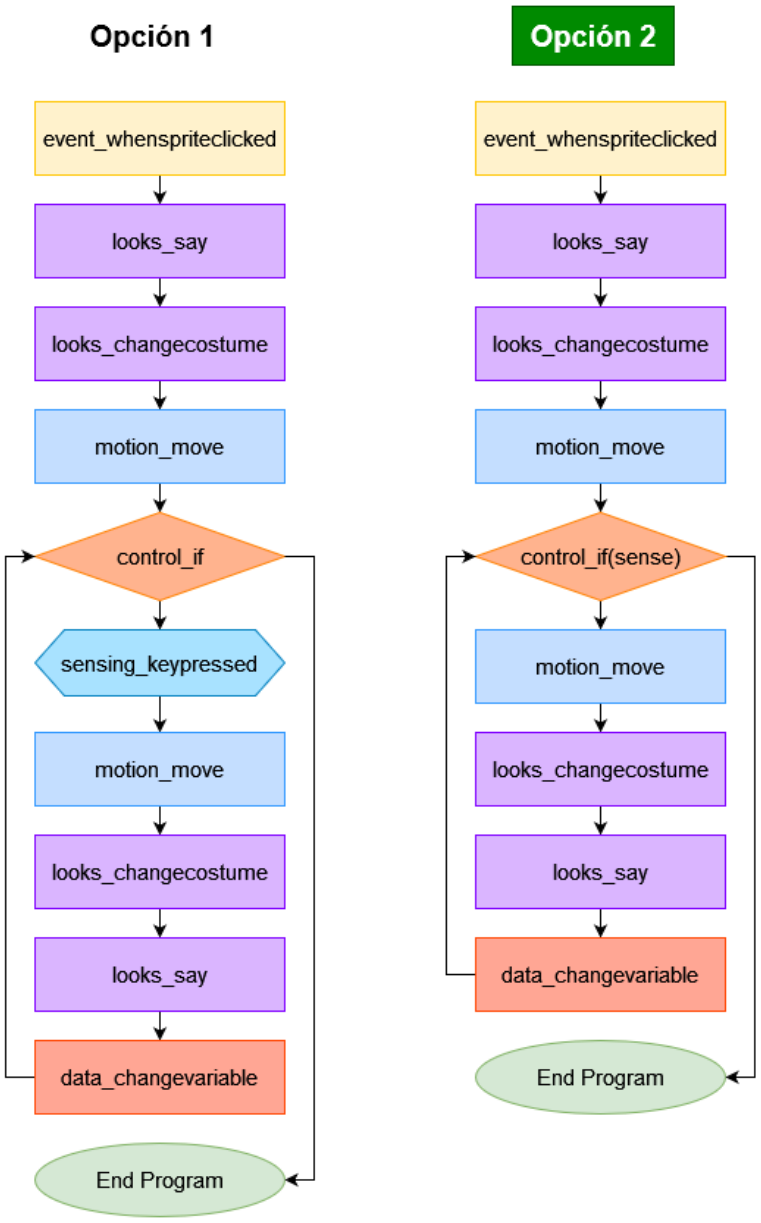

Figura 3.2: Opciones para el CFG esperado para el código asociado al actor Dino en el Ejemplo 1

Para Sun se tiene un ciclo infinito (*forever*), por lo que en este caso el código asociado a este actor se sigue ejecutando mientras no se detenga el programa. Para mantener la estructura de los CFGs (un nodo de entrada y uno de salida), se decidió crear un arco que vaya desde el inicio del ciclo hacia el nodo final del programa, un camino que sólo ocurre cuando se detiene el programa. En la Figura 3.3 se puede observar el CFG esperado.

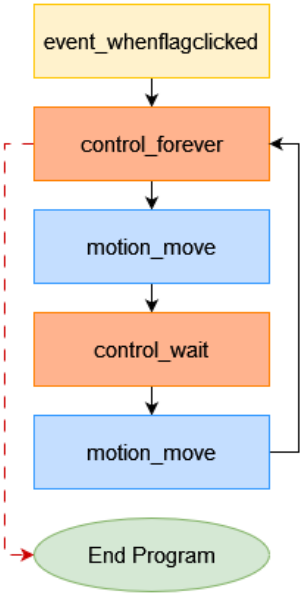

Figura 3.3: CFG esperado para el código asociado al actor Sun en el Ejemplo 1

Para Basketball se tiene un ciclo finito con condición *until* (se ejecuta el código dentro del ciclo hasta que se cumpla la condición), por lo que también se presentaron 2 propuestas (ver Figura 3.4): incluir o no la condición, y de forma análoga al caso de Dino, se optó por no incluirla en el grafo.

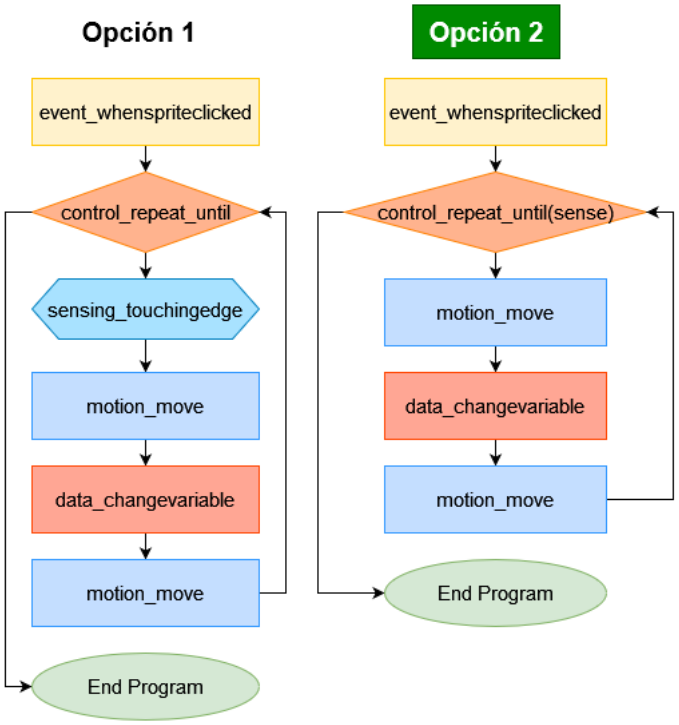

Figura 3.4: Opciones para el CFG esperado para el código asociado al actor Basketball en el Ejemplo 1

Durante el desarrollo de esta tarea se presentaron los siguientes cuestionamientos:

- ¿La condición de un bloque tipo Control (si es que la tiene, pues no todos requieren condición) debe considerarse como bloque en el CFG o no?
- ¿Qué pasa con el nodo final cuando hay un `control_forever` en el programa?
- ¿Los bloques correspondientes a parámetros o variables deben ser considerados como bloques en el CFG?

Para responder estas dudas se consultó con la profesora guía y se definieron las siguientes convenciones:

- Los nodos de un CFG solo deberían representar bloques de acción, por lo que no se consideran parámetros ni variables.
- Los bloques tipo Operator no representan acciones en el programa, por lo que no se consideran como nodos en el CFG.
- Las condiciones de un bloque tipo Control no se cuentan por separado, sino que son parte del bloque control, puesto que no representan acciones individuales en el código.
- Para los casos en que se presenten bloques `control_forever` se creará un arco que conecte el nodo que representa a este bloque con el nodo final, sabiendo que este arco nunca será ejecutado. Esto con el fin de mantener la estructura de los CFGs, pues todos deben contar con un nodo inicial (tipo Event o de definición de bloque personalizado) y un nodo final.

Además, se determinó que pueden existir múltiples CFGs por actor, uno por cada conjunto de bloques válido presentes en el espacio de trabajo del actor en la plataforma de Scratch. Un conjunto de bloques válido se define como una secuencia de bloques que comienza con un bloque tipo Event o un bloque `procedures` de definición de un bloque personalizado.

3.2. Diseño de la solución

Tras el estudio realizado de la estructura de los datos que la herramienta debe procesar y analizar, se debe definir en qué partes del proyecto se presentan los problemas y casos borde no abordados en la primera iteración y se plantean las soluciones propuestas para cada uno. Primero se encuentran los ejes principales de la solución, en donde se clasifican las distintas partes de la herramienta y se determina cuáles de ellas serán abordadas en este trabajo de memoria. Posteriormente, se definen los problemas y limitaciones para las distintas partes de la herramienta, proponiendo las soluciones correspondientes.

Cabe mencionar que los cambios realizados no afectaron la estructura del modelo de datos definido por Berger en la primera iteración de la herramienta, el cual se puede observar en el Anexo B (Figura B.1.1).

3.2.1. Ejes principales de la solución

Se trabajará en torno a los ejes principales identificados por María José Berger en la primera iteración de esta herramienta [8], los cuales serán descritos a continuación:

- **Procesamiento** se refiere a la implementación del parser para los proyectos y la generación de el/los CFGs correspondientes. En esta dimensión se centra la primera parte del trabajo a realizar, pues es la que tiene la relación más estrecha con los objetivos propuestos, consistentes en la detección y corrección de casos borde presentes en el parser para así facilitar y mejorar la generación de CFGs y proceder con los análisis y métricas.

- **Análisis y métricas** corresponde al diseño de consultas o cálculos que permitan extraer la mayor información posible a partir de los datos. Entre los análisis a realizar se encuentran la Complejidad Ciclomática [19] y la detección de código inalcanzable. Esta dimensión será abordada después de mejorar la dimensión de procesamiento. Aquí se extenderá el análisis que la herramienta ya es capaz de realizar, añadiendo el Análisis de Flujo de Datos mencionados en la Sección 2.4.

- **Resultados y reporte** de los análisis realizados, con detalles para cada dimensión relevante del proyecto (paralelismo, pensamiento lógico, control de flujo, interactividad con el usuario, representación de la información, abstracción y sincronización). Esta dimensión es la de menor prioridad, considerando que la base presentada en la primera iteración es suficiente para cumplir con los objetivos propuestos.

A partir de la base del proyecto realizado por María José Berger se hizo un estudio y *debugging* del proyecto. El enfoque principal de este trabajo de memoria corresponde a mejorar la etapa de procesamiento y análisis.

3.2.2. Parser

Las limitaciones encontradas en el *parser* fueron las siguientes:

1. Existe la posibilidad de que los escenarios (Stage) del programa contengan código compuesto por bloques de cualquier tipo excepto Motion. Esto provoca que se pierdan bloques que representan acciones en el programa.

2. No se consideran los bloques personalizados, por lo que se pierde información y bloques que representan acciones en el programa.

3. Considera todos los bloques tipo Control y Operators para la búsqueda y almacenamiento de bloques anidados. El problema con esto es que, en el caso de los bloques Control, no todos presentan anidación, por lo que se estaría contando erróneamente una cantidad mayor de bloques que la que realmente existe. En el caso de los bloques Operators, se determinó que éstos no representan acciones, por lo que no se cuentan como nodos.

En base a estos problemas, se determinó como solución refactorizar el *parser* para considerar posibles bloques en un Stage, los bloques personalizados y cambiar la condición de búsqueda de bloques anidados.

En un principio se había considerado mantener el comportamiento de la herramienta con respecto a los bloques personalizados, con la diferencia de que esta vez se alertaría de la presencia de estos bloques al evaluador para instarlo a cargar el proyecto a la plataforma de Scratch y revisarlo de una manera individual, pero tras la exploración y estudio tanto con programas de ejemplo y proyectos reales entregados por estudiantes, y considerando la inconveniencia que significaría revisar individualmente los proyectos que presentaran bloques personalizados (siendo que el fin de la herramienta es evitar esto mismo), se optó por darle al *parser* la capacidad de procesar estos bloques, pues éstos podrían representar información y acciones relevantes de los programas que los contienen. También se había considerado filtrar los bloques que se guardarían o no en la base de datos, para simplificar el procesamiento a la hora de generar los CFGs, pero esta idea descartó durante la implementación (más información al respecto en la Sección 4.1).

3.2.3. Generación de CFGs

El algoritmo de generación de CFGs se encuentra dentro del *parser* y consiste en crear arcos y asociarlos a los bloques correspondientes. El problema es que la herramienta crea más arcos de los necesarios y no considera casos borde como la presencia de bloques `control_forever` (que repiten la secuencia anidada sin condición de término) o no crea arcos que vayan desde el bloque final de una secuencia anidada hacia el bloque Control que da inicio a la anidación para formar el ciclo correspondiente. También se encontró que el nodo final de los CFGs no era creado dentro del *parser*, lo cual sería la razón por la que se crean arcos adicionales.

Como solución a estos problemas se optó por definir un nuevo algoritmo de generación de CFGs, el cual se mantendría dentro del *parser* ya refactorizado, y crear el nodo final dentro del *parser*.

3.2.4. Análisis automatizados

Debido a la refactorización del *parser* y el algoritmo de generación de CFGs, se hace necesario estudiar y revisar el cálculo de métricas que emplea la información recopilada de los bloques y arcos. En particular se estudiaron cuatro métricas:

1. **Complejidad ciclomática (CC)**
   Cuenta la cantidad de bloques de decisión en el programa y se le suma 1 ($\pi + 1$, donde π corresponde a la cantidad de bloques condicionales presentes en el código). El problema con esta métrica es que entrega un solo valor para todo el programa, independiente de la cantidad de actores. Tras consultar con la profesora guía, se llegó a la conclusión de que resultaría más útil e informativo entregar un valor por actor. Otro problema encontrado en este punto es que se cuentan todos los bloques de decisión, incluyendo los que nunca se ejecutan en el programa.

2. **Bloques inalcanzables**
   Cuenta la cantidad de bloques que nunca son ejecutados en el programa. El problema con esta métrica es que considera todos los bloques que no son precedidos por una secuencia de bloques que comience con un bloque tipo Event, por lo que no toma en cuenta la existencia de bloques personalizados, marcándolos como inalcanzables. Para

solucionar esto, se propone redefinir el concepto de bloque inalcanzable y refactorizar la función que los cuenta.

3. **Grafos inalcanzables**
   Cuenta la cantidad de grafos que nunca son ejecutados en el programa. El problema con esta métrica es análogo al del punto anterior, pues cuenta todas las secuencias de bloques que no comiencen con un bloque tipo Event. La solución propuesta es redefinir el concepto de grafo inalcanzable y refactorizar la consulta que los cuenta.

4. **Interacciones**
   Cuenta la cantidad de interacciones con el usuario, las cuales corresponden a los bloques tipo Sensing. El problema con el cálculo de esta métrica es que la primera versión solo cuenta un subtipo de bloque Sensing (los que representan a la acción `touchingobject`, que es cuando un actor "toca" otro elemento del programa, ya sea otro actor o un borde).

Otro problema encontrado, pero que no fue abordado en la implementación priorizando otros casos borde, es que la herramienta genera grafos inválidos, es decir, que solo tienen el nodo final o que no son formados por una secuencia de bloques válida (que comienza con un bloque tipo Event o la definición de un bloque personalizado). Estos grafos inválidos son contados en las estadísticas de la vista de proyecto, pero pueden o no ser mostrados en la vista de grafos.

3.2.5. Interfaz web

Durante la exploración en el *front-end* se encontraron los siguientes problemas en la interfaz de la herramienta:

1. En la vista de CFGs de un proyecto, el tamaño de fuente del nombre del actor es muy pequeño y poco visible. Además, no hay separadores claros entre cada CFG. Esto genera confusión con respecto a dónde comienza y termina cada CFG, lo cual dificulta la revisión y evaluación de los proyectos.

2. En la vista de CFGs no existe un desglose de qué significa cada nodo y qué tipo de acción representa en el programa. El tener presente algún cuadro explicativo que indique los colores, nombres y una breve descripción de cada tipo de nodo podría ser de ayuda para los evaluadores y, por ende, agilizar el proceso de revisión.

3. En todas las vistas, en la parte superior izquierda bajo la barra de navegación, hay enlaces que muestran el historial de navegación del usuario en la herramienta. Éstos son *clicables* (se puede hacer clic con el cursor), pero el puntero no cambia al posarse sobre ellos, escondiendo esta funcionalidad del usuario.

4. En la vista de curso, bajo el cuadro de carga de proyectos, los botones están mal alineados. Esto es principalmente un problema estético.

5. En la vista de proyecto, el gráfico y las tablas muestran la cantidad de bloques tipo Operator presentes en el programa, la cual siempre es cero debido a que estos bloques ya no son considerados en el *parser*. Análogo a esto, no se muestra información con respecto a los bloques personalizados, que sí son considerados, así como también los bloques tipo Variables.

6. En la vista de proyecto se muestra la Complejidad Ciclomática (CC) calculada para todo el programa. Con los cambios planteados en las subsecciones anteriores surge la necesidad de mostrar una CC por cada actor.

3.3. Resumen

En este capítulo se presentaron las limitaciones que presenta la primera versión de Scratch Analyzer y se proponen soluciones para abordar estos problemas. Primero se encuentra la definición del problema, en donde se realiza una exploración de la estructura de los datos que la herramienta debe procesar y posteriormente analizar, buscando casos borde que no fueron abordados en la primera iteración. Tras determinar los puntos a corregir a partir de la exploración, se encuentra el diseño de la solución, en donde se señalan los problemas que serán abordados y cómo se procede al respecto. Los cambios propuestos no interfieren con la estructura del modelo de datos definido en la primera iteración de la herramienta.

Capítulo 4

Implementación

De acuerdo a los problemas encontrados y expuestos en el Capítulo 3, se plantearon diversas tareas a realizar para este trabajo de memoria enfocados principalmente en las etapas de procesamiento y análisis de los proyectos, incluyendo algunos ajustes relacionados con la etapa de resultados y reporte. A continuación se describirán los cambios y mejoras que se realizaron a las distintas partes de la herramienta.

4.1. Correcciones en el parser

Tal y como fue mencionado en la Sección 3.2.2, se determinó que la solución a los problemas en la fase de procesamiento de los proyectos era refactorizar el *parser*.

En primera instancia se quería filtrar desde un principio los bloques que se consideraban irrelevantes para los CFGs (en particular los *duplicados* como `looks_costume`), eliminando así la necesidad de categorizar los bloques como *primarios* y *no primarios*. En el Código 4.1 se puede observar el algoritmo en pseudo código propuesto para el filtro de bloques innecesarios (líneas 9 a 25), incorporando el conteo de bloques condicionales y un *boolean* que indicara la presencia de bloques personalizados en el programa (líneas 1 y 2).

Código 4.1: Extracto en pseudo código de views.py para almacenamiento de bloques por actor (propuesta inicial para la versión actual)

```
1  conditionals = 0
2  my_blocks_presence = False
3  # Get blocks asociated to each actor
4  for actor in actors:
5     for block in blocks:
6          # Filter useful blocks
7          # action => event, sound, sensing, looks, motion, etc
8          # pred => movesteps, play, whenflagclicked, etc
9          if 'looks' in action:
10             if pred == 'costume': # it's a duplicate
11                 continue
12         if 'sound' in action:
13             if pred == 'sounds': # it's a duplicate
14                 continue
15         if 'sensing' in action:
16             if pred == 'answer': # it's a duplicate
```

```
                continue
        if 'operator' in action: # it's considered just to calculate the CC
            conditionals += 1
            continue
        if 'precedures' in action: # my blocks type -> don't count as block
            my_blocks_presence = True
            continue
        if 'argument' in action: # a variable -> don't count
            continue

        block = createBlock(block)
        block.save()
```

Este filtro se descartó debido a que al implementarlo se generaban inconsistencias a la hora de generar los CFGs, por lo que la implementación que quedó fue la que se puede observar en el Código 4.2 [1], en el cual las líneas 8 a 17 corresponden a los cambios realizados en ese extracto del código original, en donde se filtra y descarta bloques correspondientes a los duplicados de los bloques personalizados, Motion y Sensing, que no guardan información relevante para el análisis. También se descartan los bloques tipo *argument* debido a que entorpecen los cálculos del análisis, pero se aconseja reincorporarlos en próximas iteraciones, puesto que podrían otorgar información relevante.

Código 4.2: views.py, extracto de función add_proyect para el almacenamiento de bloques por actor (versión actual)

```
# Get blocks asociated to each actor
for actor in Actor.objects.filter(project=project):
    blocks = eval(actor.blocks)
    for block in blocks:
        full_type = blocks[block]['opcode'].split('_')
        type = full_type[0]
        action = blocks[block]['opcode'][len(type)+1:]
        pred = full_type[1:]
        # Create and save the block
        if 'procedures' in type:
            if 'prototype' in pred:
                continue
        if 'sensing' in type:
            if 'keyoptions' in action or 'touchingobjectmenu' in action or 'answer' in
    ↪ action:
                continue
        if 'argument' in type:
            continue
        if 'motion' in type:
            if 'menu' in pred:
                continue
        block = Block(name=block, type=type, action=action,
                      next=blocks[block]['next'],
                      parent=blocks[block]['parent'],
                      actor=actor,
                      project=project)
        block.save()
```

[1] `add_proyect` es el nombre de la función, es un *typo* que está pendiente por corregir.

Con respecto a los Stage que pueden presentar bloques de acción se decidió que, en caso de existir bloques para un Stage, se creará y guardará en la base de datos un actor que lo represente y que guarde a su vez sus bloques. En el Código 4.3 se puede observar la implementación, en donde las líneas 6 a 13 corresponden a los cambios realizados en ese extracto del código original.

Código 4.3: views.py, extracto de función add_proyect para almacenamiento de actores y de escenarios que presentan bloques

```
1  # Create objects for stages and actors but also keep the lists for deeper exploration
2  for element in targets:
3      if element['isStage']:
4          stage = Stage(name=element['name'], project=project)
5          stage.save()
6          # Stages can have blocks too
7          # If a stage has blocks, create an actor representing said stage
8          if len(element['blocks']) > 0:
9              print('The stage has blocks')
10             stg_actor = Actor(name=element['name'],
11                               blocks=element['blocks'],
12                               project=project)
13             stg_actor.save()
14     else:
15         actor = Actor(name=element['name'],
16                     blocks=element['blocks'],
17                     project=project)
18         actor.save()
```

Para la búsqueda de bloques anidados, bastó con cambiar la condición de las llamadas a la función que obtiene los bloques anidados (`get_nested_blocks`), la cual será descrita en la Sección 4.2. El cambio realizado para este punto se puede observar en el Código 4.4, el cual se debió aplicar a todas las ocasiones en que el *parser* y el algoritmo de generación de CFGs se encontraban con un bloque tipo Control.

Código 4.4: views.py, extracto de función add_proyect, condición para llamada recursiva en procesamiento de bloques

```
1  # If the block is control type and it is not a wait action, there is a recursive call
2  if parent_block.type=='control' and 'wait' not in parent_block.action:
3      get_nested_blocks(block, graph)
```

4.2. Algoritmo de generación de CFGs

Para resolver los problemas presentes en la generación de CFGs en la herramienta, se optó por definir un algoritmo por medio del estudio de programas de ejemplo simples de forma incremental para detectar casos borde. Para cada ejemplo se hizo un CFG que describe el comportamiento del programa, correspondiente al resultado esperado.

A partir del estudio realizado, se determinó un nuevo algoritmo de generación de CFGs, el cual fue implementado dentro del *parser*, en el archivo `views.py`, que contiene toda la lógica de la carga, procesamiento y análisis de los proyectos, motivo por el cual los cambios se

introdujeron de forma gradual y tomando en cuenta los nombres de las variables y funciones que ya se encontraban dentro del código, haciendo la refactorización correspondiente a lo largo de todo el archivo.

4.2.1. Nuevo algoritmo de generación de CFGs

El algoritmo descrito en palabras se puede encontrar a continuación:

4.2.1.1. Bloques secuenciales

1. Para cada actor, se obtienen todos los bloques que no posean un bloque padre. Estos bloques se denominan `heads`.
2. Por cada `head` se crea un CFG, `graph`, y se añade `head` a `graph`
3. Se crea un nodo final `end_node` y se añade a `graph`. Se define el bloque actual como `curr` y el siguiente como `next`
4. (Ciclo) Mientras `next` no sea *null*, se añade `next` a `graph` y se crea un arco desde `curr` a `next`.
5. Dentro del ciclo, se chequea si `next` es tipo Control y no corresponde a una acción *wait*. En el caso verdadero se hace una llamada recursiva a `get_nested_blocks` con `next` como bloque padre. En el caso falso se define el bloque actual como `curr` y el siguiente como `next` y se reanuda el ciclo.
6. Al finalizar el ciclo, se chequea si existe un arco desde `curr` a `end_node`. Si es verdadero, no se hace nada. Si es falso, se crea un arco desde `curr` a `end_node`.

A continuación se pueden observar los bloques de programas de ejemplo (a la izquierda) y los CFGs esperados que los representa (a la derecha) utilizados para la definición del algoritmo. En la Figura 4.1 se tiene un programa compuesto sólo por bloques secuenciales. A la derecha se muestra el CFG esperado para este caso.

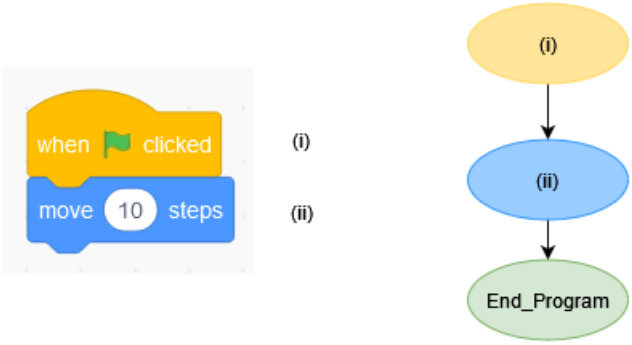

Figura 4.1: Estudio CFG esperado de programa con bloques secuenciales

En la Figura 4.2 se tiene una combinación de bloques secuenciales y anidados con un bloque condicional `if`. En el primer grafo (en medio de la figura) se tiene lo esperado para el código de la izquierda, mientras el segundo grafo representa el caso en que no existe una instrucción después de ejecutar las instrucciones dentro del condicional. En ambos casos se añadieron arcos que van desde el final de la secuencia anidada hacia el nodo siguiente que corresponde en cada caso.

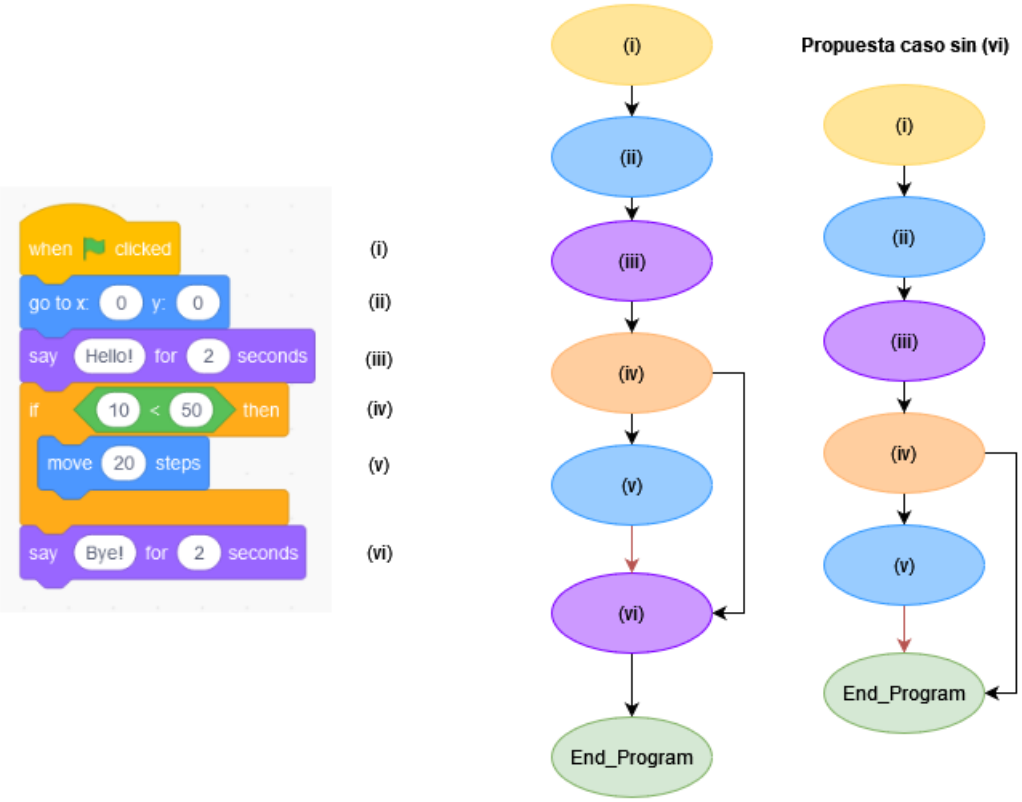

Figura 4.2: Estudio CFG esperado de programa con bloques secuenciales y anidados con *if*

El Código 4.5 corresponde al procesamiento de bloques secuenciales según el algoritmo anteriormente descrito. La función `get_nested_blocks` reemplaza a `get_conditional_blocks`, cuyo nombre fue cambiado para evitar confusiones, puesto que el objetivo de la función es obtener los bloques anidados y no solo los condicionales, considerando además que la función original fue modificada casi en su totalidad.

Código 4.5: views.py, extracto de función add_proyect para procesamiento de bloques secuenciales (pseudo código)

```
for actor in actors:
    # get all nodes that don't have a parent
    heads = get_orphan_nodes()
    for head in heads:
      # create a new graph
      graph = create_cfg()
      # add the head node to the graph
      graph.add(head)
```

```
    # create and add an end node for the graph
    end_node = create_block('End_Program')
    graph.add(end_node) #or end_node.add(graph)
    # keep track of the current and next nodes
    curr = head
    next = head.next
    # while there is a next node
    while(next):
      # add next node to graph
      graph.add(next)
      # create an arch from the current node to the next node
      create_arch(curr, next, graph)
      # if block is control type and it is not a wait action, there is a recursive call
      type = parent_block.actionType()
      if next.isControl() and type != 'wait':
        get_nested_blocks(next, graph)
      # update the current and next nodes
      curr = next
      next = next.next
    # if the current node has not arch to the end node, create one
    if  curr.hasArchTo(end_node, graph) == False:
      create_arch(curr, end_node, graph)
```

4.2.1.2. Bloques anidados

1. Dado un bloque `parent_block` (de tipo Control y que no corresponde a una acción *wait*) y el grafo al que pertenece, `graph`, se obtienen todos los bloques que lo señalan como padre. Estos bloques se denominan `nested_blocks`.

2. Por cada `nested_block`, se revisa si éste corresponde al *next* de `parent_block`. En el caso verdadero, no se hace nada, puesto que el arco que conecta ambos bloques ya fue creado fuera de esta función. En el caso falso, se añade `nested_block` a `graph` y se crea un arco desde `parent_block` a `nested_block`.

3. Se chequea si `nested_block` es tipo Control y no corresponde a una acción *wait*. En el caso verdadero se hace una llamada recursiva a `get_nested_blocks` con `nested_block` como bloque padre. En el caso falso se define el bloque actual como `curr` y el siguiente como `next`.

4. (Ciclo) Mientras `next` no sea *null*, se añade `next` a `graph` y se crea un arco desde `curr` a `next`.

5. Dentro del ciclo, se chequea si `next` es tipo Control y no corresponde a una acción *wait*. En el caso verdadero se hace una llamada recursiva a `get_nested_blocks` con `next` como bloque padre. En el caso falso se define el bloque actual como `curr` y el siguiente como `next` y se reanuda el ciclo.

6. Al finalizar el ciclo, se chequea si la acción de `parent_block` corresponde a un *loop* (es decir, si es una instrucción `repeat`, `repeat_until` o `forever`). En el caso verdadero, se crea un arco desde `curr` a `parent_block` para crear el ciclo. En el caso falso no se hace nada.

7. Se chequea si la acción de `parent_block` corresponde a un condicional (es decir, si es una instrucción `if` o `if-else`). En el caso verdadero, se revisa si `parent_block` tiene un bloque que le sigue secuencialmente (atributo *next* en el JSON).

   a) En el caso verdadero, se crea un arco desde `curr` hacia el bloque *next* de `parent_block`. Esto se hace para mantener la secuencialidad correspondiente entre los bloques anidados dentro de la condición con el bloque que se encuentra inmediatamente después del bloque condicional (en las Figuras 4.5 y 4.2, los CFGs de la izquierda representan este caso).

   b) En el caso falso, se crea un arco desde `curr` hacia el nodo final del grafo (en las Figuras 4.5 y 4.2, los CFGs de la derecha representan este caso).

8. Una vez revisados todos los bloques anidados en `nested_blocks`, se finaliza la función (retorna).

A continuación se pueden observar los bloques de programas de ejemplo (a la izquierda) y los CFGs esperados que los representa (a la derecha) utilizados para la definición del algoritmo. En la Figura 4.3 se tiene un programa compuesto de bloques anidados con un ciclo (*loop*) que se repite 10 veces.

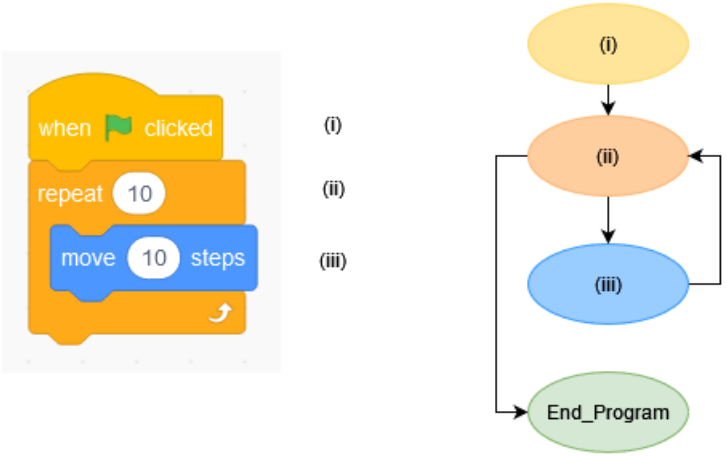

Figura 4.3: Estudio CFG esperado de programa con bloques anidados con *loop*

En la Figura 4.4 se muestra un programa compuesto por bloques secuenciales y anidados con un ciclo que se repite 10 veces y tras ello se ejecuta una instrucción más antes de finalizar.

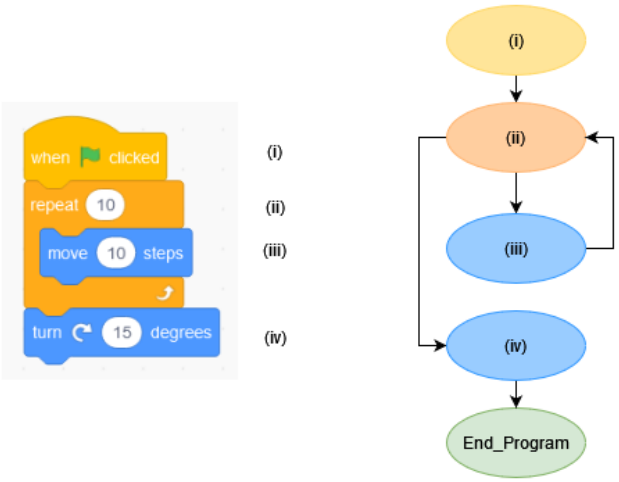

Figura 4.4: Estudio CFG esperado de programa con bloques secuenciales y anidados con *loop*

En la Figura 4.5 se tiene un programa con bloques secuenciales y anidados con un bloque condicional *if-else*. En el primer grafo (en medio de la figura) se tiene lo esperado para el código de la izquierda, mientras el segundo grafo representa el caso en que no existe una instrucción después de ejecutar las instrucciones dentro del condicional. En ambos casos se añadieron arcos que van desde el final de la secuencias anidadas hacia el nodo siguiente que corresponde en cada caso.

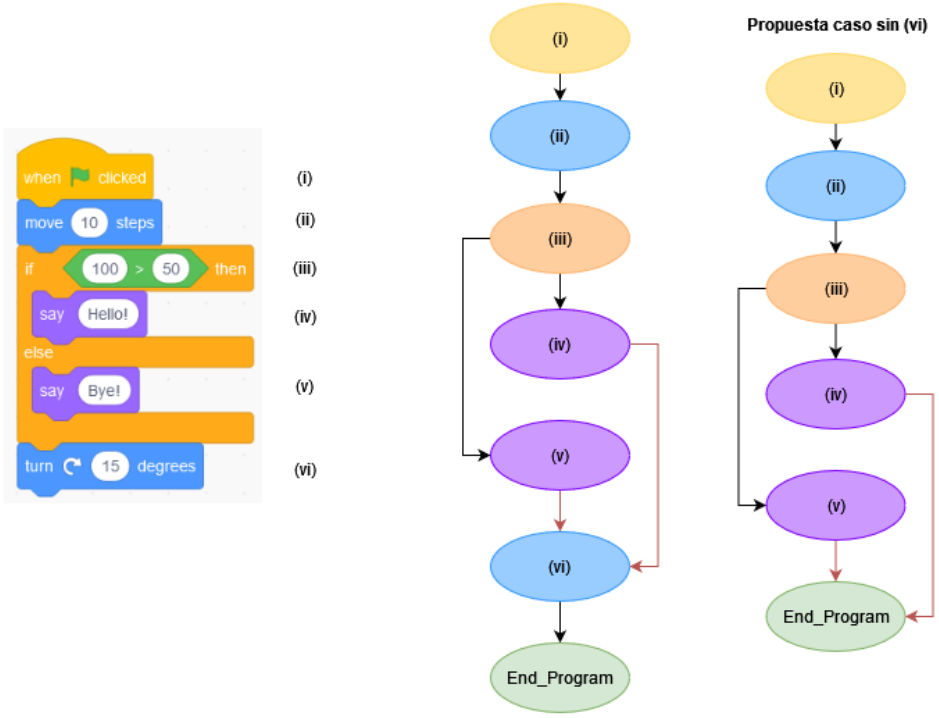

Figura 4.5: Estudio CFG esperado de programa con bloques secuenciales y anidados con *if-else*

El Código 4.6 corresponde al procesamiento de bloques anidados, donde se define la función `get_nested_blocks`, la cual recibe un bloque padre (`parent_block`) y su grafo correspondiente (`graph`). En los comentarios se indica paso a paso, y en inglés, el algoritmo anteriormente descrito. Esta función se ocupa de procesar y guardar todos los nodos hijos de `parent_block`, añadirlos `graph` y crear los arcos correspondientes para plasmar la lógica y secuencialidad del código en bloques en la estructura de grafo.

Código 4.6: views.py, extracto de función add_proyect para procesamiento de bloques anidados (pseudo código)

```
# creates arches for nested blocks
def get_nested_blocks(parent_block, graph):
  # get all blocks that have parent_block as a parent
  nested_blocks = parent_block.getChildren()
  # for each nested block
  for block in nested_blocks:
    # if the block is not the next block of the parent block
    if block != parent_block.next:
      # add block to graph and create an arch from the parent block to the block
      graph.add(block)
      create_arch(parent_block, block, graph)
      # if block is control type and it is not a wait action, there is a recursive call
      type = parent_block.actionType()
      if block.isControl() and type != 'wait':
        get_nested_blocks(block, graph)
      # update current and next nodes
      curr = block
      next = block.next
      # while there is a next node
      while(next):
        # add next node to graph and create an arch from the current node to the next
    ↪ node
        graph.add(next)
        create_arch(curr, next, graph)
        if next.isControl() and next.actionType() != 'wait':
          get_nested_blocks(next, graph)
        # update current and next nodes
        curr = next
        next = next.next
      # if the parent block is a loop block, create an arch from the current node to the
    ↪ parent block
      if type == 'loop':
        create_arch(curr, parent_block, graph)
      # if the parent block is a conditional block
      elif type == 'conditional':
        # if the parent block has a next node, create an arch from the current node to
    ↪ the next node, to maintain the flow
        if parent_block.next:
          create_arch(curr, parent_block.next, graph)
        # if the parent block doesn't have a next node, create an arch from the current
    ↪ node to the end node
        else:
          create_arch(curr, graph.end_node, graph)
  return
```

4.2.2. Representación visual de CFGs

Dado que el *parser* guarda los duplicados de algunos tipos de bloque (como Looks), se hizo necesario filtrar qué bloques se mostrarían en los CFGs. En la primera versión de la herramienta esto se hacía seleccionando sólo los bloques marcados como *primarios*, lo cual no pudo ser aplicado debido a los cambios realizados en el *parser* que fueron descritos anteriormente. Además, dado que la herramienta ahora considera los bloques personalizados, se debió añadir la capacidad de mostrar estos bloques y sus CFGs correspondientes. Finalmente, ya que el nodo final de los CFGs se crea dentro del *parser*, se debe eliminar tanto la creación de este nodo como la de los arcos que van dirigidos hacia él fuera de la función `add_proyect` (que contiene al *parser*).

La función encargada de generar la representación visual de los CFGs es `plot_info`. La nueva versión de esta función se puede observar en el Código 4.7, que muestra una versión resumida del código real. Esta función obtiene todos los grafos almacenados para el proyecto y, para cada uno de ellos, filtra los arcos de acuerdo a los nodos que conectan, descartando los que se asocian a bloques que no representan acciones relevantes en el programa (como los duplicados), almacenando en `final_data` los arcos (y sus respectivos nodos) que pasaron estos filtros.

Código 4.7: views.py, plot_info

```
def plot_info(request):
    colors = {
        ...
        'final' : '#93db86',
        'procedures': '#ff6680'
        ...
    }
    if request.method == 'GET':
        ...
        for g in graphs:
            final_data = {"nodes": [], 'edges':[]}
            arch_list = []
            added_nodes = []
            # Edge case: an argument block generates an empty graph => skip
            if g.archs.all().__len__() <= 1:
                continue
            # Filter blocks by type and action
            for arch in g.archs.all():
                if arch.block1 not in added_nodes:
                    if arch.block1.type not in ['operator', 'argument'] and arch.block1.
    ↪ action not in ['costume', 'touchingobject', 'keyoptions', 'answer']:
                        final_data['nodes'].append(arch)
                        added_nodes.append(arch.block1)
                if arch.block2 not in added_nodes:
                    if arch.block2.type not in ['operator', 'argument'] and arch.block2.
    ↪ action not in ['costume', 'touchingobject', 'keyoptions', 'answer']:
                        final_data['nodes'].append(arch)
                        added_nodes.append(arch.block2)
                edge= {'from': arch.block1.id, 'to': arch.block2.id}
                final_data['edges'].append(edge)
            nodes = final_data['nodes']
```

```
            # Remove unnecesary nodes
            i=0
            while(i<nodes.__len__()):
                if(nodes[i]['label'] in ['sensing_touchingobjectmenu']):
                    final_data['nodes'].remove(nodes[i])
                i += 1
            all_graphs[g.actor.name].append(final_data)
        return Response(all_graphs)
```

4.3. Cambios en análisis automatizados

De acuerdo a lo señalado en la Sección 3.2.4, se realizaron cambios en tres métricas de análisis: complejidad ciclomática, bloques inalcanzables y grafos inalcanzables. Los cambios implementados a cada una de ellas se describen a continuación.

4.3.1. Complejidad ciclomática

En un principio, y considerando los cambios realizados en el *parser* y el algoritmo para la generación de CFGs, se tenía contemplado comenzar a utilizar la fórmula de McCabe: $E - N + 2P$ [19] donde, dado un grafo $G = (N, E)$, E corresponde al número de arcos, N al número de nodos y P al número de componentes conexas del grafo.

Empleando los ejemplos realizados para el estudio de la estructura y comportamiento de los programas en Scratch, junto con algunos proyectos reales entregados por los niños y niñas, se calculó la CC por actor de acuerdo a los CFGs mostrados en la herramienta, considerando como cantidad de componentes conexas para un actor la cantidad de CFGs generados para este mismo. En la Tabla 4.1 se puede observar una comparación de la CC por actor (y general) calculada con las fórmulas $E - N + 2P$ y $\pi + 1$ (donde π corresponde a la cantidad de bloques/nodos de decisión presentes en el programa). De acuerdo a lo expuesto en la tabla, se puede notar que se presentan inconsistencias entre los valores entregados por las fórmulas. En honor al tiempo, no se pudo seguir ahondando en esta métrica, por lo que se optó mantener la fórmula empleada en la primera iteración ($\pi+1$), con la diferencia de que la métrica es calculada por actor y no para el programa completo. Se tomó esta decisión puesto que tener un valor de complejidad por actor otorga más información con respecto al comportamiento del programa.

En el Código 4.8 se puede observar la implementación de la función que calcula la CC por actor empleando la fórmula $\pi + 1$.

Tabla 4.1: Cálculo y comparación de CC con distintas fórmulas

Programa	**Actor**	E	N	P	$E - N + 2P$	$\pi_{actor} + 1$	$\pi_{total} + 1$
Ejemplo1	Cat	7	8	1	1	1	3
	Dino	12	11	1	3	2	
	Sun	5	6	1	1	1	
	Basketball	6	6	1	2	2	
Ejemplo2	Monkey	30	29	1	3	3	3
Ejemplo3	Ballerina	13	15	2	2	1	1
Ejemplo4	Kid	24	24	2	4	2	2
Proyecto 1	bulbasur	15	18	3	3	1	2
	hierbita	6	9	3	3	1	
	ash	28	33	6	7	2	
	Mankey	-	-	-	-	-	
Proyecto 2	Squirtle	27	31	6	8	3	6
	pokeball	18	23	6	7	3	
	Pikachu	26	31	6	7	2	

Código 4.8: views.py, extracto de función get_data para cálculo de Complejidad Ciclomática

```
1  def calculate_cyclomatic_complexity_by_actor(a):
2      # Every graph has at least 1 possible path
3      complexity = 1
4      blocks = project.blocks.all()
5      for block in blocks:
6          # Filter by actor
7          if (block.actor.name == a):
8              if block.action in ['if', 'if_else', 'repeat_until']:
9                  complexity = complexity + 1
10     return complexity
```

4.3.2. Bloques inalcanzables

Como se propone en la Sección 3.2.4, se redefine el concepto de bloques inalcanzables a "bloques que no tienen un bloque padre tipo Event o My Blocks ni son precedidos por una secuencia de bloques válida", donde una secuencia de bloques válida corresponde a "aquella que comienza con un bloque tipo Event o My Blocks". En base a esto se implementó el Código 4.9, correspondiente a la función que cuenta la cantidad de nodos inalcanzables presentes en el programa. Esta función obtiene todos los nodos sin padre, excluyendo el nodo final, los nodos tipo Event y los nodos personalizados de definición (tipo `procedures` y acción `definition`), lo cual corresponde a la instrucción de la línea 4. La función `count_conditional_blocks` es análoga a `get_nested_blocks` presentada en el Código 4.6.

Código 4.9: views.py, extracto de función get_data para cálculo de bloques inalcanzables

```
# Hanging Nodes: Those with no Event-type block or Procedures definition as node head
def count_hanging_nodes():
    count = 0
    nodes = Block.objects.filter(project=project, parent=None).exclude(type='final').
    ↪ exclude(type='event').exclude(type='procedures', action='definition')
    for node in nodes:
        prev = node
        next = node.next
        if node.type == 'control' and node.action != 'wait':
            count = count + count_conditional_blocks(node, count)
        count = count + 1
        while (next):
            block = Block.objects.get(name=next, project=project)
            # If a block is type Control (not wait), there's a recursive call
            if block.type == 'control' and block.action != 'wait':
                count = count + count_conditional_blocks(block, count)
            count = count + 1
            prev = block
            next = block.next
    return count
```

4.3.3. Grafos inalcanzables

De forma análoga a la métrica de bloques inalcanzables, se considera que la cantidad de grafos inalcanzables de un programa corresponde a la cantidad de secuencias no válidas, es decir las que cuyo nodo padre no corresponde a un bloque tipo Event o a la definición de un bloque personalizado. En base a esto se implementó el Código 4.10, en donde se cuentan todos los bloques sin padre que no sean tipo Event ni `procedures`.

Código 4.10: views.py, extracto de función get_data para cálculo de grafos inalcanzables

```
# Count hanging graphs
statistics['general']['hanging_graphs'] = Block.objects.filter(
                project=project, primary=True, parent=None).exclude(type='event').exclude
    ↪ (type='procedures', action='definition').count()
```

4.3.4. Interacciones

En este caso se modificó la consulta para el conteo de interacciones para que considere los bloques tipo Sensing que representen una acción, como se puede observar en el Código 4.11.

Código 4.11: views.py, extracto de función get_data para cálculo de interacciones

```
# Interactions
statistics['general']['interactions'] = Block.objects.filter(
            project=project, type='sensing').exclude(action='keyoptions').exclude(action=
    ↪ 'touchingobjectmenu').exclude(action='answer').count()
```

4.4. Ajustes en la interfaz web

De acuerdo a lo expuesto en la Sección 3.2.5, se realizaron cambios en la interfaz web de la herramienta con el fin de mejorar la experiencia de usuario (UX) [23].

A continuación se listan los ajustes realizados:

- Se cambia el cursor en *hover* para los Breadcrumbs (enlaces en la esquina superior derecha de la ventana del navegador que muestran el historial de navegación dentro de la herramienta) a *pointer* cuando se posa el cursor sobre ellos en todas las páginas de la herramienta.
- En la vista de curso se movieron los botones que están debajo del cuadro de carga de proyectos para que quedaran en columna y no en fila (ver Figura 4.6).

Figura 4.6: Ajustes en vista de curso, sección de carga de proyectos

- En la vista de proyecto (ver Figura 4.7):
  - Ya no se muestran los datos de los bloques tipo Operator, puesto que estos bloques ya no son considerados para el conteo de bloques dado que no corresponden a acciones relevantes (se usan principalmente como condiciones dentro de un bloque Control). En cambio, ahora se muestran los bloques personalizados My Blocks y los bloques tipo Variables.
  - Ya no se muestra la CC general del proyecto, sino que se da un valor de CC calculado por actor.
  - Se añade un comentario de aclaración sobre el cálculo de la CC mostrada.

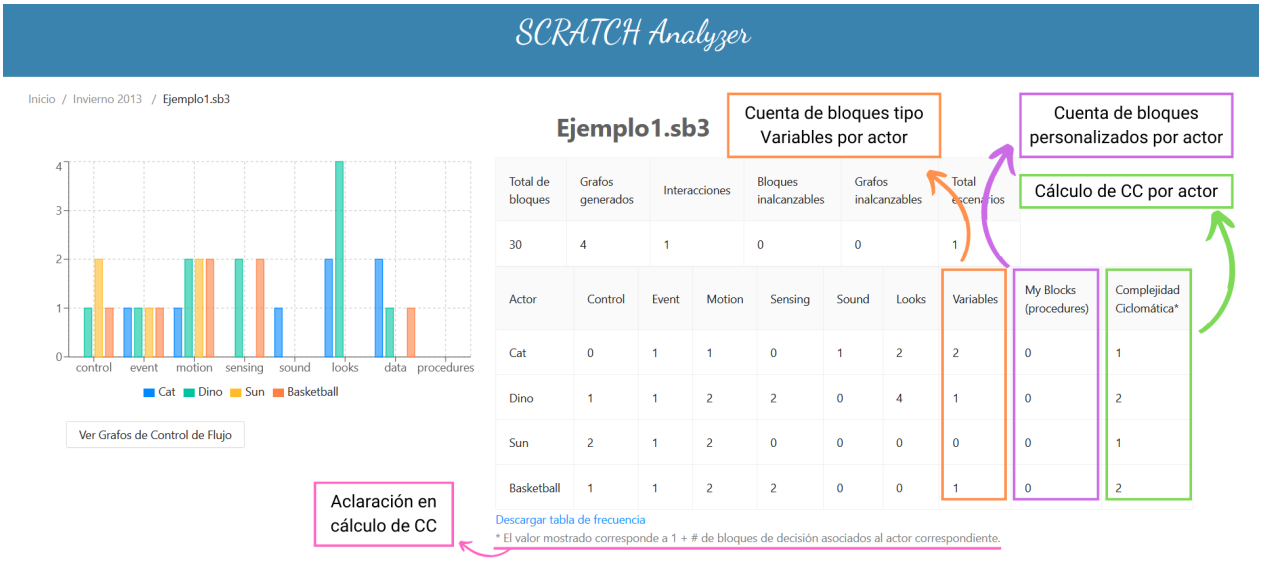

Figura 4.7: Ajustes en vista de proyecto

- En la vista de CFGs para un proyecto:
  - Se añade una tabla con los nombres, colores y descripciones de los tipos de nodos que presentan los CFGs (ver Figura 4.8).
  - Se aumenta el tamaño de los títulos de los CFG, se añadieron márgenes y se dejaron en negrita. Además, se añaden líneas punteadas como separadores entre CFGs, se aumenta la altura máxima que ocupa cada CFG de 400px a 500px y se aumenta el tamaño de fuente de los nodos de los CFG de 14px a 16px (ver Figura 4.9).

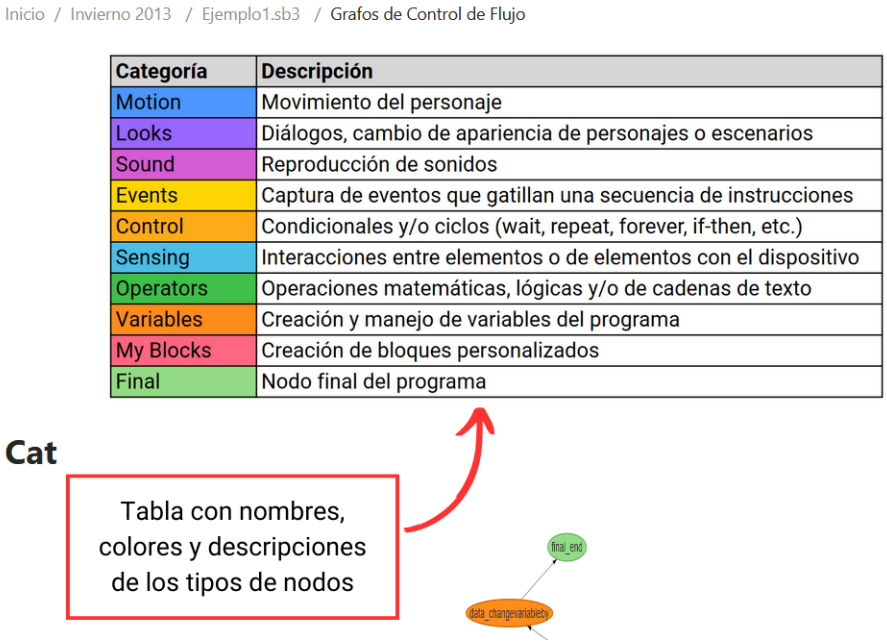

Figura 4.8: Ajustes en vista de CFGs 1

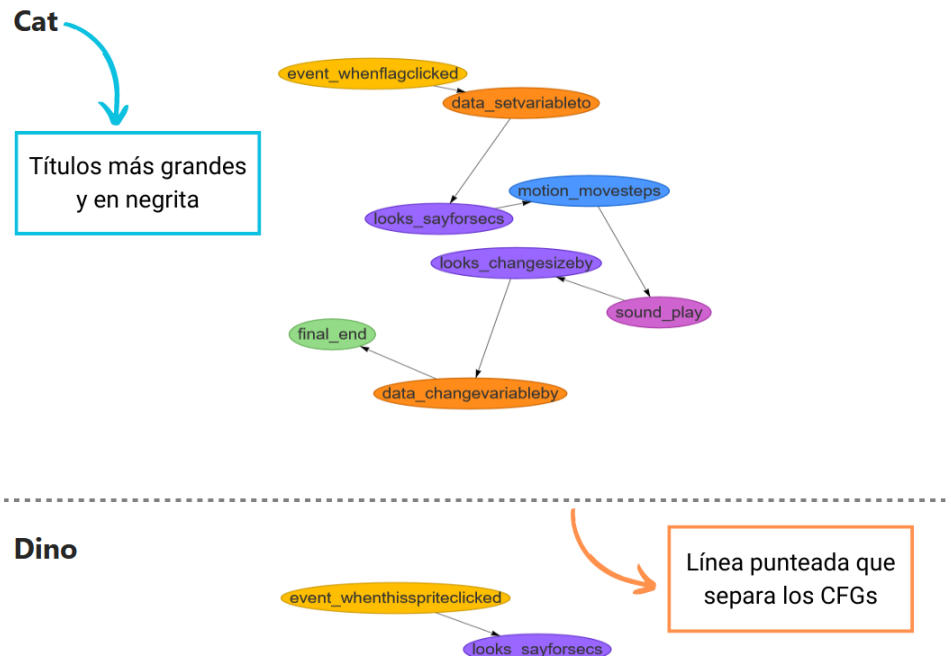

Figura 4.9: Ajustes en vista de CFGs 2

- Se cambia el nombre del *tab* de la herramienta en el navegador, así como también el icono, para personalizar (ver Figura 4.10).

Figura 4.10: Ajustes en *tab* del navegador web

4.5. Resumen

En este capítulo se describen las decisiones tomadas con respecto a la implementación de los cambios propuestos en el Capítulo 3. Las correcciones en el *parser* abarcan desde qué bloques se almacenan efectivamente en la base de datos, incluyendo los bloques asociados a escenarios, hasta el cambio en la condición de búsqueda de bloques anidados, que ahora considera sólos los bloques cuyo bloque padre es de tipo Control y acción distinta a `wait`.

El algoritmo de generación de CFGs fue redefinido e implementado en `views.py` tanto para bloques secuenciales como anidados; la definición del nuevo algoritmo se hizo por ingeniería inversa, estudiando programas de ejemplo y los CFGs que se espera sean generados a partir del código correspondiente. Para la representación gráfica de los CFGs, se incorporaron los bloques personalizados (My Blocks) y se descartaron los bloques que no representaban una acción en el código (como los Operators).

En los cambios en los análisis automatizados se consideraron cuatro métricas: complejidad ciclomática, bloques inalcanzables, grafos inalcanzables e interacciones. Para la complejidad ciclomática se mantuvo el uso de la fórmula $\pi + 1$ (donde π corresponde a la cantidad de bloques/nodos de decisión), con la diferencia de que en la versión actual se calcula por actor y no para el programa completo, pues así otorga más información con respecto al comportamiento del programa. Para el cálculo de bloques y grafos inalcanzables, se redefinió el filtro para la cuenta de bloques/grafos que se utilizaba en la primera versión, pues éste no consideraba los bloques personalizados. Con respecto a las interacciones, también se modificó el filtro para la cuenta de bloques, que ahora considera sólo los bloques Sensing que representan una acción en el programa.

Finalmente, se realizaron ajustes en la interfaz web para complementar los cambios anteriormente mencionados, además de mejorar la experiencia de usuario. Entre los cambios relevantes se encuentra la incorporación del conteo de tipos de bloques que no fueron considerados en la primera versión (como My Blocks y Variables), además de mostrar la complejidad ciclomática por actor en vez del proyecto completo. También se hicieron ajustes en la vista de grafos generados, añadiendo una tabla resumen con los tipos de bloques, aumentando el tamaño de letra de los títulos de los grafos y agregando separadores entre ellos para evitar confusiones.

Capítulo 5

Validación

Tras la implementación de los cambios descritos en el Capítulo 4, es necesario realizar una validación de que éstos, en efecto, resultan en un impacto positivo en la herramienta. A continuación se describirá sobre qué aspectos de la herramienta se realizará la validación y cómo se llevará a cabo la misma, exponiendo sus resultados y su respectiva discusión.

Un aspecto importante que da valor a la herramienta es la veracidad de los resultados entregados en las estadísticas, por lo que en la Sección 5.1 se tomará una muestra de proyectos que serán evaluados manualmente y se cargarán a la herramienta en su versión original y la actual y se compararán los resultados.

También está la facilidad de uso, es decir que la herramienta sea lo suficientemente intuitiva para los usuarios. Para esto se realizarán pruebas de usabilidad con usuarios finales (investigadores/evaluadores). La evaluación de esta prueba de usabilidad se realizará por medio de SUS (System Usability Scale) [24], cuyos resultados se expondrán en la Sección 5.2.

La legibilidad del código para facilitar la mejora y extensión de la herramienta en futuras iteraciones. Para este aspecto se considerará la documentación del código, en donde se espera que las funciones más relevantes de la herramienta cuenten con una breve descripción y un desglose de las instrucciones más importantes que la componen. En la Sección 5.3 se realizará una comparación entre partes del código antes y después de los cambios realizados, junto con un análisis estático del código [27] para la detección de *bugs* y advertencias sobre malas prácticas.

5.1. Validación funcional

Para la validación funcional se emplearon programas de ejemplo (dos de ellos correspondientes a proyectos entregados por estudiantes reales del curso), los cuales fueron analizados manualmente y posteriormente cargados a la plataforma para realizar el análisis y capturar los resultados que éste entrega tanto en la primera versión de la herramienta como en la actual.

5.1.1. Cantidad de bloques/nodos

En las Figuras 5.1 y 5.2 se pueden observar los gráficos de barras de bloques por actor generados por ambas versiones de la herramienta. La primera corresponde a un proyecto con solo un actor, mientras que la segunda a un proyecto con múltiples actores. La principal diferencia entre ambos es que en la primera versión sí considera los bloques tipo Operator, pero no los tipo Variables (*data*) y My Blocks (*procedures*). Además, en el ejemplo con múltiples actores se puede observar que en la segunda versión aparece un nuevo actor, Stage, lo cual se debe a que la herramienta actualmente es capaz de detectar y guardar bloques presentes en los Stages del programa, creando un actor nuevo que lo represente y almacene la información de sus bloques y así no perder información con respecto al comportamiento del programa.

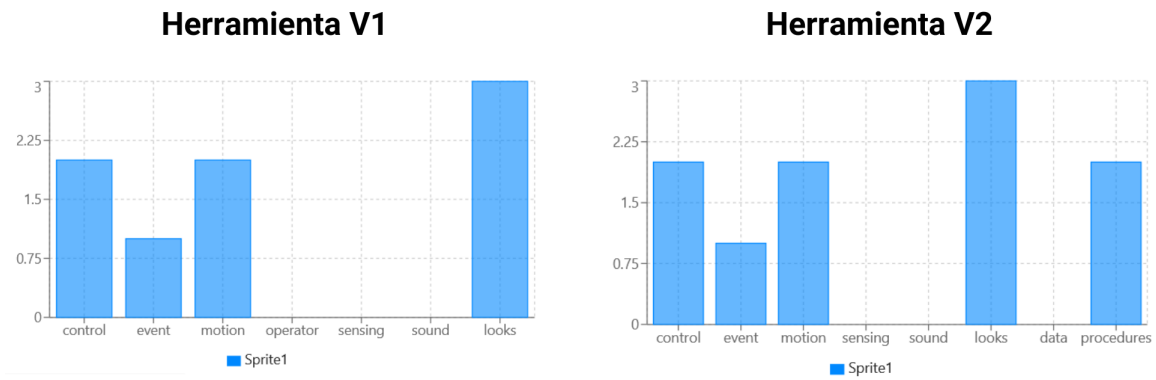

Figura 5.1: Gráfico de barras de bloques por actor (1 actor)

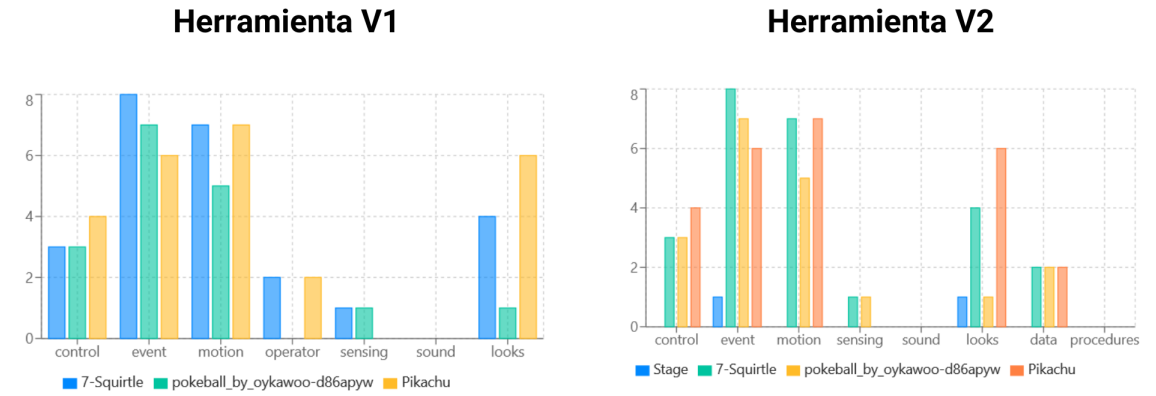

Figura 5.2: Gráfico de barras de bloques por actor (múltiples actores)

En la Figura 5.3 (en el Anexo C se pueden ver los datos en la Tabla C.1.1) se puede observar una comparación entre la cantidad de bloques totales presentes en cada programa estudiado en las diferentes plataformas, en donde se puede ver que la cantidad de bloques procesados por la herramienta corresponde con lo que se vería en la plataforma de Scratch si se cargaran los proyectos en ella, a pesar de que la información que procesa la herramienta es la proporcionada por el JSON, cuya cantidad de bloques es siempre mayor o igual a la cantidad de bloques presente en el programa real, debido a la presencia de duplicados (más

información al respecto en la Sección 3.1.1).

Otra distinción notable se puede observar en el Ejemplo 5, donde en la primera versión de la herramienta la cantidad de bloques es cero puesto que no muestra ningún CFG, esto se debe a que este ejemplo cuenta con bloques personalizados (My Blocks), que no fueron considerados en la primera iteración, mientras que en la versión actual sí se consideran y, por ende, se generan los CFGs correspondientes.

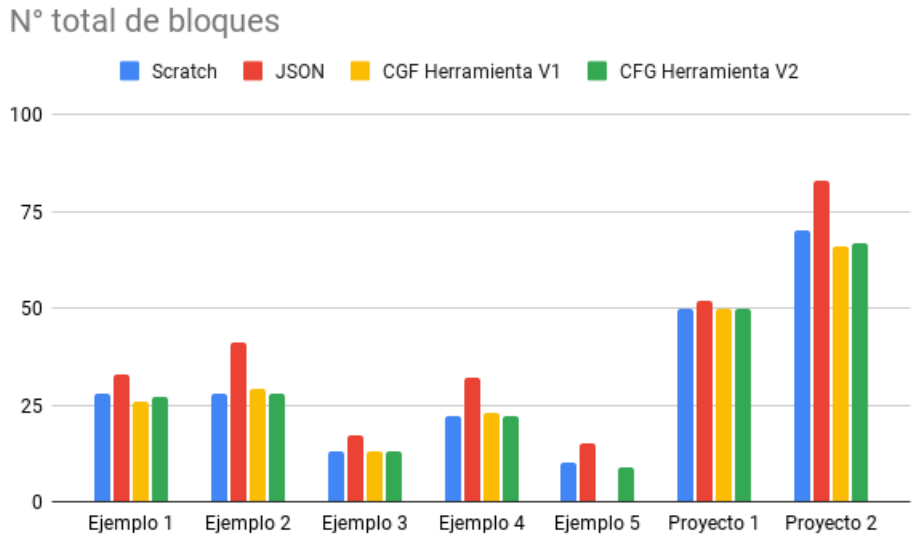

Figura 5.3: Gráfico de barras de bloques totales por programa

5.1.2. Cálculo de métricas

En las Figuras 5.4 y 5.5 se pueden observar comparaciones de las distintas métricas calculadas por la herramienta en ambas versiones y los valores esperados. Estos valores fueron tomados de las estadísticas mostradas en la vista de proyecto de la herramienta (ver Figura 4.7).

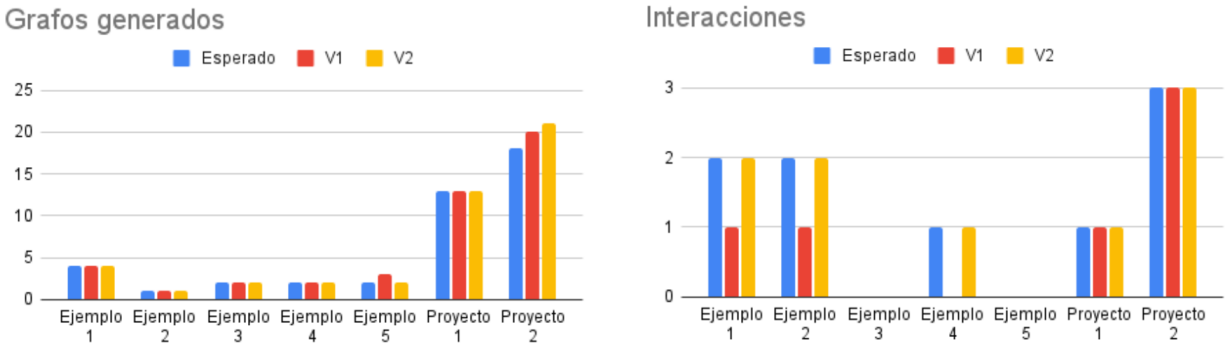

Figura 5.4: Comparación de métricas (grafos generados e iteracciones) calculadas por la herramienta en ambas versiones con lo esperado

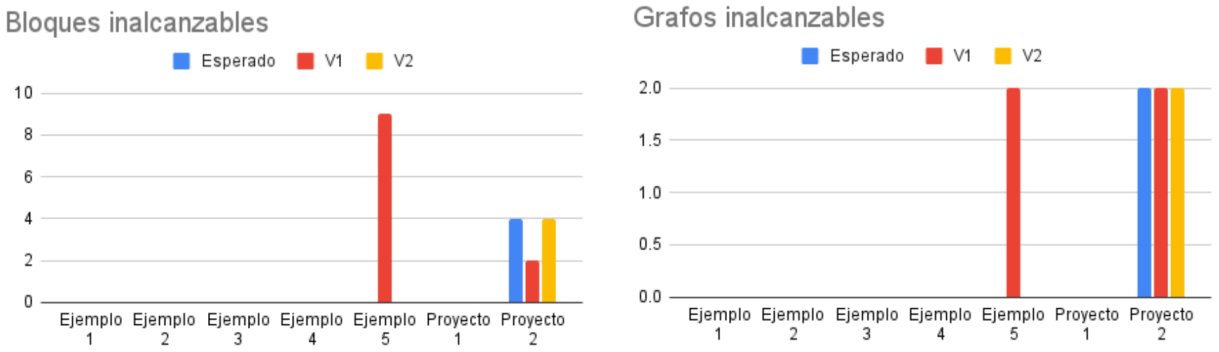

Figura 5.5: Comparación de métricas (bloques y grafos inalcanzables) calculadas por la herramienta en ambas versiones con lo esperado

De los gráficos se desprende que en general los cálculos de la versión actual de la herramienta son consistentes con lo esperado en la mayoría de los casos, pero en el Proyecto 2 se puede ver que la cantidad de grafos generados no concuerda con el valor esperado. Ahora, en la Figura 5.6 se puede ver la comparación entre la cantidad de grafos mostrados realmente por la herramienta con lo esperado, donde se puede apreciar que la versión actual sí muestra la cantidad de grafos adecuada para el Proyecto 2, pero en las estadísticas señala un valor distinto. Este es un error que se propone corregir en las siguientes iteraciones.

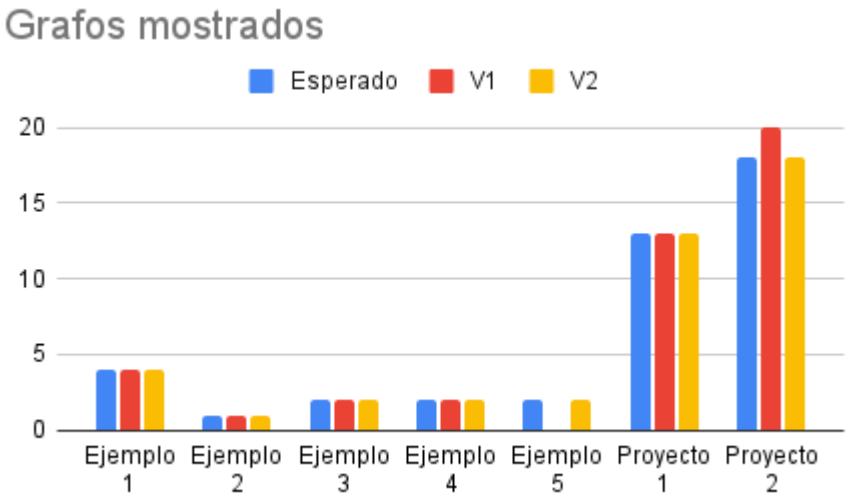

Figura 5.6: Comparación de cantidad de grafos mostrados por la herramienta en ambas versiones con lo esperado

En la Tabla 5.1 se puede observar una comparación de la Complejidad Ciclomática calculada por la herramienta en ambas versiones con lo esperado. Como fue mencionado en la Sección 4.3.1, la Complejidad Ciclomática fue calculada como $\pi + 1$, donde π es la cantidad de bloques condicionales (de decisión) presentes en el programa, contando sólo los que se encuentran dentro de secuencias de bloques válidas (más información en la Sección 4.3.2). De acuerdo a los datos expuestos en la tabla, se puede observar que el principal cambio es que la versión actual de la herramienta entrega valores por actor, mientras que la primera versión sólo entrega un valor general para todo el programa.

Tabla 5.1: Comparación valores de Complejidad Ciclomática calculados por ambas versiones de la herramienta con lo esperado

	Esperado	V1	V2
Ejemplo 1	Cat: 1 Dino: 2 Sun: 1 Basketball: 2	3	Cat: 1 Dino: 2 Sun: 1 Basketball: 2
Ejemplo 2	Monkey: 3	3	Monkey: 3
Ejemplo 3	Ballerina: 1	1	Ballerina: 1
Ejemplo 4	Kid: 2	2	Kid: 2
Ejemplo 5	Sprite1: 1	1	Sprite1: 1
Proyecto 1	bulbasur: 1 hierbita: 1 ash: 2 Mankey: 1	2	bulbasur: 1 hierbita: 1 ash: 2 Mankey: 1
Proyecto 2	Stage: 1 Squirtle: 3 pokeball: 3 Pikachu: 2	6	Stage: 1 Squirtle: 3 pokeball: 3 Pikachu: 2

5.2. Test de usabilidad

Para realizar el test de usabilidad de la herramienta, se pidió a uno de los investigadores, Francisco Gutiérrez, que probara la aplicación y respondiera el test SUS [24], el cual consta de 10 preguntas en donde se presenta una afirmación y según ésta se debe indicar si se está de acuerdo o en desacuerdo en una escala de 1 a 5, donde 1 es “Muy en desacuerdo” y 5 es “Muy de acuerdo”. El fin de este test es medir la facilidad de uso percibida de un sistema [25], pero cabe señalar que investigaciones más recientes indican que también proporciona una medida global de la satisfacción del sistema y subescalas de usabilidad y capacidad de aprendizaje [26].

Los resultados de la aplicación del test SUS a la herramienta en su versión actual se pueden observar en la Tabla 5.2, en donde también se muestran los puntajes normalizados de cada ítem. La normalización de los puntajes consiste en lo siguiente: para ítems impares se resta 1 al puntaje y para ítems pares se resta el puntaje a 5.

Tabla 5.2: Afirmaciones y sus respectivos puntajes asignados (y normalizados) en el test de usabilidad

Afirmación	**Puntaje**	**Puntaje normalizado**
(1) Creo que me gustaría utilizar este sistema con frecuencia	5	4
(2) El sistema me pareció innecesariamente complejo	2	3
(3) Pensé que el sistema fue fácil de usar	4	3
(4) Creo que necesitaría el apoyo de un técnico para poder utilizar este sistema	1	4
(5) Encontré que las diversas funciones de este sistema estaban bien integradas	5	4
(6) Pensé que había demasiada inconsistencia en este sistema	1	4
(7) Me imagino que la mayoría de la gente aprendería a utilizar este sistema muy rápidamente	4	3
(8) El sistema me pareció muy complicado de utilizar	1	4
(9) Me sentí muy seguro al usar el sistema	4	3
(10) Necesité aprender muchas cosas antes de poder empezar a utilizar este sistema	1	4

Habiendo normalizado los puntajes, éstos valores deben sumarse y el total ser multiplicado por 2.5, con lo que se obtiene un valor dentro de la escala $0 - 100$, el cual corresponde a la calificación. Realizando esta operación se obtiene una calificación de 90.

Además de la asignación de puntajes, el investigador proporcionó las siguientes observaciones:

- *La aplicación está bien. Las funcionalidades están contextualizadas y compactas en un solo sistema. Es relativamente intuitiva de usar para una persona con conocimientos de ingeniería de software y terminologías técnicas empleadas en la herramienta.*
- *Se puede evaluar un proyecto sin necesidad de abrirlo en la plataforma o aplicación de escritorio de Scratch. Este tipo de evaluación in situ puede ahorrar recursos y simplificar la tarea de procesamiento de datos, pues los condensa en las rúbricas y tablas de frecuencia.*
- *Se detectaron detalles en el layout, se recomienda separar más las tablas de métricas y usar colores distintos en botones cercanos para evitar confusiones.*

- *No se requiere ayuda para usar la aplicación, pero sí para instalarla, hay que ver si se puede hacer algo al respecto a futuro.*
- *Es deseable intentar unificar y comparar los datos que entrega la herramienta con los que otorga Dr. Scratch, para así tener información más completa para la evaluación de los proyectos.*

5.3. Evaluación del código

Como se menciona en la Sección 4.2, se realizó una refactorización del algoritmo de generación de CFGs, en parte porque la función era sumamente complicada y también porque presentaba nombres de variables y funciones que no correspondían con lo que representaban realmente, dificultando aún más la legibilidad del código. En la nueva versión de la herramienta, pensando en que son necesarias más iteraciones que serán realizadas por otros desarrolladores, se optó por dejar comentarios en los pasos importantes de las funciones con el algoritmo escrito en palabras, con el fin de simplificar la fase de exploración y familiarización con el código en futuras iteraciones.

A continuación se realizarán comparaciones de extractos de código que cumplen la misma función, extraídos de la primera versión y la versión actual de la herramienta.

En los Códigos 5.1 y 5.2 se puede observar el inicio de la función encargada de obtener los bloques anidados y crear los arcos correspondientes para la generación de los CFGs. La principal diferencia es el nombre de la función, el cual fue cambiado puesto que el anterior no reflejaba lo que realmente hace. Otra diferencia son los comentarios, donde en la versión actual se describen los pasos importantes con el algoritmo en palabras para mejorar la comprensión de lo que hace la función.

Código 5.1: views.py, primer extracto de función add_proyect para creación de arcos (primera versión)

```
# Recursive function to access the blocks generated by a conditional block that are
# block: recursive block
# next_prim_block: block to come back to linear secuence

def get_conditional_blocks(block, graph):
    og_block = block
    nested_blocks = Block.objects.filter(...)

    # Explorations indicate in every query 1.- Condition 2.- substack1 3.- substack2
    if len(nested_blocks) >= 2:
        condition_block = nested_blocks[0]
    else:
        condition_block = None

    while (len(nested_blocks) > 0):
        ...
```

Código 5.2: views.py, primer extracto de función add_proyect para creación de arcos (versión actual)

```
# creates arches for nested blocks
def get_nested_blocks(parent_block, graph):
    # get all blocks that have parent_block as a parent
    nested_blocks = Block.objects.filter(...)

    # for each nested block
    while(len(nested_blocks) > 0):
        ...
```

En los Códigos 5.3 y 5.4 se muestra otra parte de la misma función. La principal diferencia entre estos dos extractos es la cantidad de comentarios, pues en la primera versión sólo presenta uno, mientras que en la versión actual hay cuatro. También se puede observar que los comentarios de la versión actual describen mejor qué hace la función.

Código 5.3: views.py, segundo extracto de función add_proyect para creación de arcos (primera versión)

```
prev = next_block
next = next_block.next
while (next):
    n_block = Block.objects.get(...)
    n_block.primary = True
    n_block.graph = graph
    n_block.save()
    if n_block == condition_block or n_block.type == 'operator' or prev.type == 'operator
    ↪ ':
        arch = Arch(...)
    else:
        arch = Arch(...)
    arch.save()
    # if block is type control it means there is more blocks

    if n_block.type == 'control' or n_block.type == 'operator':
        get_conditional_blocks(n_block, graph=graph)
    prev = n_block
    next = n_block.next
```

Código 5.4: views.py, segundo extracto de función add_proyect para creación de arcos (versión actual)

```
curr = block
next = block.next

# while there is a next node
while(next):
    # add the next node to the graph and create an arch from the current node to the next
    ↪  node
    next_block = Block.objects.get(...)
    next_block.primary = True
    next_block.graph = graph
    next_block.save()

```

```
    arch = Arch(...)
    arch.save()

    # if the next node is a control node and it is not a wait block, there is a recursive
    ↪ call
    if next_block.type=='control' and 'wait' not in next_block.action:
        get_nested_blocks(next_block, graph)

    # update current and next
    curr = next_block
    next = next_block.next
```

Para complementar, se realizó un análisis estático de software [27] en el código `views.py`, empleando la herramienta Pylint [28]. Los resultados se pueden observar en el Código 5.5, donde se muestran los *warnings*, *errors* y *refactors* propuestos para el código analizado, asignando un puntaje al final (línea 53). Se removieron las advertencias relacionadas a convenciones (como nombres de variables y líneas en blanco).

Código 5.5: Resultado análisis estático del código usando Pylint

```
************* Module core.views
views.py:7:0: E0401: Unable to import 'django.contrib.auth.models' (import-error)
views.py:8:0: E0401: Unable to import 'django.core.files.storage' (import-error)
views.py:9:0: E0401: Unable to import 'rest_framework' (import-error)
views.py:10:0: E0401: Unable to import 'rest_framework' (import-error)
views.py:11:0: E0401: Unable to import 'rest_framework.decorators' (import-error)
views.py:12:0: E0401: Unable to import 'rest_framework.response' (import-error)
views.py:21:0: R0903: Too few public methods (0/2) (too-few-public-methods)
views.py:39:32: W0613: Unused argument 'pk' (unused-argument)
views.py:62:13: R1732: Consider using 'with' for resource-allocating operations (consider-using-
    ↪ with)
views.py:47:24: W0613: Unused argument 'request' (unused-argument)
views.py:47:33: W0613: Unused argument 'pk' (unused-argument)
views.py:66:12: W0612: Unused variable 'fdir' (unused-variable)
views.py:78:29: W0613: Unused argument 'request' (unused-argument)
views.py:78:38: W0613: Unused argument 'pk' (unused-argument)
views.py:81:8: W0612: Unused variable 'data' (unused-variable)
views.py:132:30: W0613: Unused argument 'pk' (unused-argument)
views.py:139:19: W0613: Unused argument 'request' (unused-argument)
views.py:139:28: W0613: Unused argument 'pk' (unused-argument)
views.py:146:21: W0613: Unused argument 'request' (unused-argument)
views.py:146:30: W0613: Unused argument 'pk' (unused-argument)
views.py:165:0: R0903: Too few public methods (0/2) (too-few-public-methods)
views.py:174:0: R0914: Too many local variables (34/15) (too-many-locals)
views.py:191:43: W0622: Redefining built-in 'zip' (redefined-builtin)
views.py:258:24: W0622: Redefining built-in 'type' (redefined-builtin)
views.py:259:24: W0621: Redefining name 'action' from outer scope (line 11) (redefined-outer-name)
views.py:397:24: W0622: Redefining built-in 'next' (redefined-builtin)
views.py:193:37: R1732: Consider using 'with' for resource-allocating operations (consider-using-
    ↪ with)
views.py:201:28: W1309: Using an f-string that does not have any interpolated variables (f-string-
    ↪ without-interpolation)
views.py:207:16: W0702: No exception type(s) specified (bare-except)
views.py:211:16: W0105: String statement has no effect (pointless-string-statement)
views.py:232:16: W0105: String statement has no effect (pointless-string-statement)
views.py:255:29: W0123: Use of eval (eval-used)
views.py:284:16: W0105: String statement has no effect (pointless-string-statement)
views.py:317:28: W0622: Redefining built-in 'next' (redefined-builtin)
views.py:292:58: W0640: Cell variable project defined in loop (cell-var-from-loop)
views.py:314:32: W0640: Cell variable get_nested_blocks defined in loop (cell-var-from-loop)
views.py:323:95: W0640: Cell variable project defined in loop (cell-var-from-loop)
views.py:336:36: W0640: Cell variable get_nested_blocks defined in loop (cell-var-from-loop)
```

```
views.py:354:124: W0640: Cell variable project defined in loop (cell-var-from-loop)
views.py:360:106: W0640: Cell variable project defined in loop (cell-var-from-loop)
views.py:289:16: R1711: Useless return at end of function or method (useless-return)
views.py:369:16: W0105: String statement has no effect (pointless-string-statement)
views.py:174:0: R0912: Too many branches (26/12) (too-many-branches)
views.py:174:0: R0915: Too many statements (123/50) (too-many-statements)
views.py:431:0: R1710: Either all return statements in a function should return an expression, or
    ↪ none of them should. (inconsistent-return-statements)
views.py:560:20: W0622: Redefining built-in 'next' (redefined-builtin)
views.py:564:44: E1120: No value for argument 'count' in function call (no-value-for-parameter)
views.py:577:16: W0622: Redefining built-in 'next' (redefined-builtin)
views.py:505:0: R0915: Too many statements (77/50) (too-many-statements)

-------------------------------------------------------------------
Your code has been rated at 7.72/10
```

5.4. Discusión

Con respecto a la funcionalidad de la herramienta, se tiene que los cambios en el procesamiento de los programas resultaron en una mejora en el cálculo de métricas, haciendo que los valores que entrega la herramienta se asemejen aún más a lo que se espera de una revisión manual e individual de los programas sin necesidad de emplear la misma cantidad de tiempo y recursos. De todas formas hay aspectos que requieren atención, como la discrepancia entre grafos generados y mostrados en la herramienta en algunos casos, como se puede observar en la Figura 5.4 (gráfico en la esquina superior izquierda) y 5.6, más precisamente en el Proyecto 2.

En términos de la Complejidad Ciclomática se plantea como deseable el aplicar la fórmula $E - N + 2P$ [19] usando los CFGs generados y comparar los valores obtenidos con $\pi + 1$, que son los que la herramienta muestra actualmente. Se propone comparar el resultado de ambas fórmulas y no reemplazar uno con el otro para así poder dar más información a los investigadores para que discutan y determinen cuál de los valores representa mejor la complejidad de los programas, dando la posibilidad de mantener ambas fórmulas o elegir sólo una para mostrar en la herramienta en futuras iteraciones.

En el cálculo de interacciones se propone redefinir lo que es una interacción en el programa, puesto que actualmente sólo considera los bloques tipo Sensing que representan una acción, pero también debería considerarse como una interacción los bloques tipo Event que reciben *input* del usuario, como presionar una tecla (ver Figura 5.7) o hacer *click* en un actor (ver Figura 5.8), pues por definición éstos corresponden a interacciones con el usuario [18].

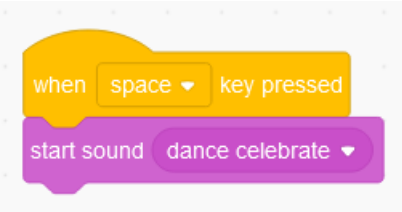

Figura 5.7: Secuencia de bloques que presenta interacción con bloque Event (presionar tecla)

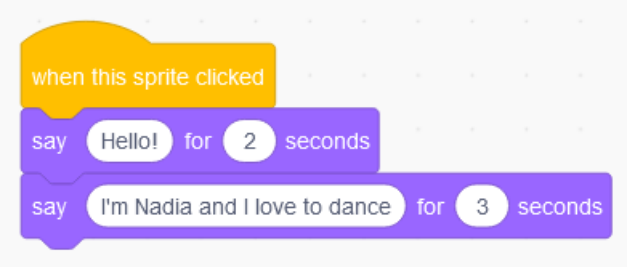

Figura 5.8: Secuencia de bloques que presenta interacción con bloque Event (hacer *click* en actor)

Con respecto al test de usabilidad, considerando que las calificaciones por sobre 68 se clasifican por sobre el promedio, el que la herramienta haya obtenido 90 indica que la facilidad de uso percibida es alta y se encuentra por sobre el promedio, lo que también señala un resultado favorable con respecto a la usabilidad y capacidad de aprendizaje de la herramienta [26], lo cual se condice con las observaciones entregadas por el investigador que realizó el test.

En términos de legibilidad del código, los cambios realizados ayudan con respecto a su lectura pero no son suficientes para lograr una comprensión y familiarización más inmediata, pues lo recomendable sería modularizar el código, es decir, dividir las funciones grandes en varias funciones más pequeñas tal que cada una se encargue de algo particular, haciendo el código más fácil de leer, mantener, hacer pruebas, encontrar *bugs* y reutilizar código [29]. Además, es necesario revisar y corregir los *refactors*, errores y advertencias detectados por Pylint en el Código 5.5.

Capítulo 6

Conclusión

El presente trabajo de memoria corresponde a la extensión de Scratch Analyzer, una herramienta que nace de la necesidad de unificar y automatizar la revisión y análisis de proyectos desarrollados en el lenguaje de programación basado en bloques llamado Scratch. Los proyectos que la herramienta debe procesar provienen de los cursos de programación en Scratch para niños y niñas, impartidos por investigadores del DCC. Para evaluar el desempeño de los y las estudiantes que asisten al curso éstos deben presentar un proyecto final, el cual debe ser revisado por el equipo de investigadores para luego utilizar los datos extraídos de estas evaluaciones en la investigación, la cual busca determinar si el curso, incluyendo los temas y metodologías empleadas, es efectivo en la educación temprana de niños y niñas, y si también éste motiva a los mismos a interesarse en el área de la programación. Actualmente, los investigadores deben revisar los proyectos individualmente, dividiendo la evaluación en dos partes: el análisis proporcionado por la plataforma de análisis de proyectos Dr. Scratch (a la cual sólo se puede cargar un programa a la vez) y la aplicación de la rúbrica de evaluación diseñada por los investigadores, por lo que la creación de una herramienta que realice ambas tareas en conjunto ahorraría bastantes horas de trabajo, acelerando la recopilación de datos para la investigación.

Scratch Analyzer permite la carga masiva de proyectos, los cuales los organiza además según el curso al que pertenecen. La herramienta procesa los programas, traduciéndolos a una estructura de datos manejable en Python, para luego aplicar los análisis y cálculos relevantes para un proyecto de software, como la complejidad ciclomática e interacciones con el usuario. Además, genera Grafos de Control de Flujo (CFGs) para cada secuencia de bloques válida dentro del programa, mostrando así de una manera más gráfica el comportamiento del programa sin necesidad de cargarlo y correrlo en la plataforma de Scratch.

La primera versión de Scratch Analyzer, sin embargo, presentaba problemas y limitaciones que impedían que se pudieran cargar y analizar algunos proyectos, además de no ser capaz de generar todos los CFGs correspondientes en casos con bloques personalizados, por lo que en esta iteración se estudiaron los casos de borde y *bugs* presentes en la herramienta y se propusieron e implementaron cambios que fueron desde el procesamiento y análisis de los proyectos, hasta ajustes en la interfaz web para complementar esos cambios. Cabe señalar que los cambios propuestos no interfieren con la estructura del modelo de datos definido en la primera iteración de la herramienta.

Los cambios implementados fueron divididos en 4 partes: correcciones en el *parser* (intérprete), definición de un nuevo algoritmo de generación de CFGs y ajustes en la representación gráfica de los mismos, cambios en los análisis automatizados (cálculo de métricas) y ajustes en la interfaz web de la herramienta. Las correcciones en el *parser* abarcan desde qué bloques se almacenan efectivamente en la base de datos, incluyendo los bloques asociados a escenarios, hasta el cambio en la condición de búsqueda de bloques anidados. El algoritmo de generación de CFGs fue redefinido e implementado tanto para bloques secuenciales como anidados; la definición del nuevo algoritmo se hizo por ingeniería inversa, estudiando programas de ejemplo y los CFGs que se espera sean generados a partir del código correspondiente. Para la representación gráfica de los CFGs, se incorporaron los bloques personalizados (My Blocks) y se descartaron los bloques que no representaban una acción en el código (como los Operators). En los cambios en los análisis automatizados se consideraron cuatro métricas: complejidad ciclomática, bloques inalcanzables, grafos inalcanzables e interacciones, en los cuales se realizaron ajustes en los cálculos para lograr que los resultados entregados por la herramienta se alinearan mejor con los valores esperados de una evaluación manual.

Para evaluar si los cambios realizados otorgan un valor a la herramienta, se realizó una validación funcional y una evaluación de legibilidad de código empleando herramienta de análisis estático de software (Pylint). Con respecto a la funcionalidad de la herramienta, se tiene que los cambios en el procesamiento de los programas resultaron en una mejora en el cálculo de métricas para el análisis de los proyectos, logrando que los valores que entrega la herramienta se asemejen aún más a lo que se espera de una revisión manual e individual de los programas sin necesidad de emplear la misma cantidad de tiempo que el sistema de evaluación actual. Sin embargo, se detectó una discrepancia entre la cantidad de grafos generados y los que son efectivamente mostrados por la herramienta en algunos casos particulares; también se plantea la recomendación de revisitar el cálculo de las métricas complejidad ciclomática e interacciones con el usuario en futuras iteraciones.

Con respecto a los resultados obtenidos en el test de usabilidad empleando el SUS, se tiene que la herramienta presenta una alta facilidad de uso percibida para una persona familiarizada con los conceptos y aplicaciones de la ingeniería de software, que corresponde a lo que se espera del usuario objetivo para esta herramienta.

A partir de las observaciones proporcionadas por el investigador que aplicó el test SUS a la herramienta se puede afirmar que ésta sí proporciona valor y cumple con la función para la que fue creada, es decir, unificar y simplificar la evaluación de proyectos desarrollados en Scratch. Cabe señalar que de ellas también surgen requerimientos y ajustes que deben ser tomados en consideración para las futuras iteraciones.

De la evaluación de legibilidad de código se desprende que aún queda trabajo por hacer para mejorar la facilidad de comprensión del código. Entre las soluciones propuestas para mejorar este punto está el modularizar el código y refactorizar nombres de funciones y variables, así como también redactar en el `README` un resumen de las funciones y estructuras importantes en el código, indicando sus respectivas ubicaciones en el proyecto, permitiendo así un fácil acceso y comprensión de las distintas funcionalidades de la herramienta y así sim-

plificar y disminuir el tiempo requerido para la familiarización con del código para futuras iteraciones de esta herramienta y así permitir a los investigadores reemplazar el sistema de evaluación actual de manera más rápida y sencilla.

Así, considerando el objetivo general de este trabajo de memoria, se concluye que éste sí se cumplió, pues el parseo, análisis y representación gráfica por medio de CFGs de los proyectos desarrollados en Scratch es ahora más robusto y completo que en la versión anterior de la herramienta, pues considera casos relevantes que no se habían tomado en cuenta en la primera versión (como los bloques en Stages, por ejemplo) y realiza cálculos más acertados de las métricas para el análisis de proyectos.

Si bien la herramienta en su estado actual cumple con el objetivo principal por el que ésta fue creada, se recomienda continuar realizando iteraciones para abordar los casos borde que no fueron considerados en este trabajo de memoria (como la discrepancia entre la cantidad de grafos generados y mostrados) e incorporar los cambios planteados a partir de las observaciones recibidas en la validación con usuarios finales (como la integración de los resultados de la herramienta con los de Dr. Scratch y mejorar el *layout* de las tablas de frecuencia y métricas en la vista de proyectos). Una posible extensión de funcionalidades que resultaría útil es añadir la capacidad de comparar proyectos y entregar métricas de similitud, ya sea entre proyectos entregados por los niños y niñas o con los proyectos de ejemplo que éstos utilizaron como inspiración para sus programas. Otra funcionalidad deseable es la autenticación de usuarios, junto con pasar la aplicación a producción para que los evaluadores puedan acceder a ella desde sus propios dispositivos sin necesidad de realizar el proceso de instalación y ejecución de la herramienta, los cuales pueden resultar engorrosos (de acuerdo a la experiencia del investigador que realizó el test de usabilidad). Por último, y tal como se menciona unos párrafos atrás, se recomienda modificar el `README`, agregando un resumen de las funciones y estructuras más importantes en el código, indicando sus respectivas ubicaciones en el proyecto. Con esto, Scratch Analyzer podrá reemplazar de forma definitiva al sistema de evaluación actual y así facilitar el trabajo de los evaluadores y la recopilación de datos para determinar si el exponer a niños y niñas desde temprana edad a cursos de programación y pensamiento computacional tiene un impacto significativo en ellos.

Bibliografía

[1] Martin, D. (2020). *The Best Age for a Child to Start Coding*. [Blog] https://www.codemonkey.com/blog/the-best-age-for-a-child-to-start-coding Accessed 22/08/2022.

[2] Pérez Angulo, J. (2019). *El pensamiento computacional en la vida cotidiana*. Revista Scientific, [online] (vol. 4, núm. 13), pp. 293-306. https://www.redalyc.org/journal/5636/563659492016/html/ Accessed: 22/08/2022

[3] Hermans, F; Aivaloglou, E. (2017). *Teaching Software Engineering Principles to K-12 Students: A MOOC on Scratch*. IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET), 2017, pp. 13-22. https://doi.org/10.1109/ICSE-SEET.2017.13

[4] Pell, S. (2021). *What is Block-Based Coding?*. [Blog] Robotical. https://robotical.io/blog/what-is-block-based-coding/ Accessed: 22/08/2022

[5] Simmonds, J.; Gutierrez, F.; Casanova, C.; Sotomayor, C.; Hitschfeld, N. (2018). *Coding or Hacking?: Exploring Inaccurate Views on Computing and Computer Scientists among K-6 Learners in Chile*. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18). Association for Computing Machinery, New York, NY, USA, pp. 993–998. https://doi.org/10.1145/3159450.3159598

[6] Gutierrez, F.; Simmonds, J.; Hitschfeld, N; Casanova, C.; Sotomayor, C; Peña-Araya, V. (2018). *Assessing software development skills among K-6 learners in a project-based workshop with Scratch*. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '18). Association for Computing Machinery, New York, NY, USA, pp. 98–107. https://doi.org/10.1145/3183377.3183396

[7] Simmonds, J.; Gutierrez, F.; Casanova, C.; Sotomayor, C.; Hitschfeld, N. (2019). *A Teacher Workshop for Introducing Computational Thinking in Rural and Vulnerable Environments*. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). Association for Computing Machinery, New York, NY, USA, pp. 1143–1149. https://doi.org/10.1145/3287324.3287456

[8] Berger, M. (2021). Universidad de Chile. *Herramienta para análisis de programas desarrollados en Scratch*

[9] Dr. Scratch Blog. November, 2015.
http://www.drscratch.org/
Accessed: 11/10/2022

[10] Cooper, K.; Torczon, L. (2012). *Engineering a Compiler (Second Edition), Chapter 5: Intermediate Representations*. Morgan Kaufmann, pp. 221-268.
https://doi.org/10.1016/B978-0-12-088478-0.00005-0
Recovered from: https://www.sciencedirect.com/topics/computer-science/control-flow-graph
Accessed: 23/08/2022

[11] React JS
https://es.reactjs.org/docs/getting-started.html
Accessed: 23/09/2022

[12] Django REST Framework
https://es.reactjs.org/docs/getting-started.html
Accessed: 23/09/2022

[13] Visual Studio Code
https://code.visualstudio.com/
Accessed: 23/09/2022

[14] Ubuntu
https://ubuntu.com/desktop
Accessed: 28/10/2022

[15] Scratch. *Acerca de Scratch*
https://scratch.mit.edu/about
Accessed: 22/08/2022

[16] Moreno-León, J.; Robles, G. (2015). *Dr. Scratch: Análisis Automático de Proyectos Scratch para Evaluar y Fomentar el Pensamiento Computacional*. RED-Revista de Educación a Distancia, Número monográfico sobre "Pensamiento computacional" 46(10), pp. 3-6. September 2015
https://doi.org/10.6018/red/46/10

[17] Dr. Scratch.
https://drscratchblog.wordpress.com/tag/dr-scratch/
Accessed: 19/10/2022

[18] Endrass, Birgit, et al (2011). *"Towards Culturally-Aware Virtual Agent Systems."Handbook of Research on Culturally-Aware Information Technology: Perspectives and Models*, edited by Emmanuel G. Blanchard and Danièle Allard, IGI Global, 2011, pp. 412-430.
https://doi.org/10.4018/978-1-61520-883-8.ch018

[19] McCabe, T. (1976) *A Complexity Measure*. IEEE Transactions on Software Engineering, vol. se-2, no.4, pp. 308-320. December 1976.
http://www.literateprogramming.com/mccabe.pdf

[20] Jana, S. (2006). *Data flow analysis in Compiler*. Columbia.
http://www.cs.columbia.edu/~suman/secure_sw_devel/Basic_Program_Analysis_DF.pdf
Accessed: 23/08/2022

[21] Gosling, J.; Joy, B.; Steele, G.; Bracha, G.; Buckley, A. (2014). *The Java Language Specification, Java SE 8 Edition (1st. ed.)*. Addison-Wesley Professional.
https://docs.oracle.com/javase/specs/jls/se9/html/jls-16.html

[22] Wegman, M.; Zadeck, F. (1991). *Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), 181–210.*
https://doi.org/10.1145/103135.103136

[23] Norman, D.; Nielsen, J. *The Definition of User Experience (UX).*
https://www.nngroup.com/articles/definition-user-experience/
Accessed: 27/10/2022

[24] John Brooke (1996). *SUS – a quick and dirty usability scale*. Book Usability Evaluation in Industry (pp.189-194) (January 1996)
https://www.researchgate.net/publication/319394819_SUS_--_a_quick_and_dirty_usability_scale
Accessed: 05/07/2023

[25] Yi He; Qimei Chen; Kitkuakul, S. (2018). *Regulatory focus and technology acceptance: Perceived ease of use and usefulness as efficacy*. Cogent Business & Management, 5:1.
https://doi.org/10.1080/23311975.2018.1459006

[26] Lewis, J.; Sauro, J. (2009). *The Factor Structure of the System Usability Scale*. Proceedings of the 1st International Conference on Human Centered Design: Held as Part of HCI International. 5619. 94-103.
https://doi.org/10.1007/978-3-642-02806-9_12
Recovered from: https://measuringu.com/wp-content/uploads/2017/07/Lewis_Sauro_HCII2009.pdf

[27] Wichmann, B.; Canning, A.; Clutterbuck, D.; Winsbarrow, L.; Ward, N.; Marsh, D. (1995). *Industrial Perspective on Static Analysis*. Software Engineering Journal. 10 (2): 69–75.
https://doi.org/10.1049/sej.1995.0010

[28] PyPI. *Pylint 2.17.4*
https://pypi.org/project/pylint/
Accessed: 18/07/2023

[29] Macdonald, M. (2023) *Modular programming: beyond the spaghetti mess*, Tiny Blog. Tiny Technologies Inc., 28 June.
Available at: https://www.tiny.cloud/blog/modular-programming-principle/
Accessed: 14/07/2023.

Anexos

Anexo A

Scratch

A.1. Capturas interfaz web Scratch

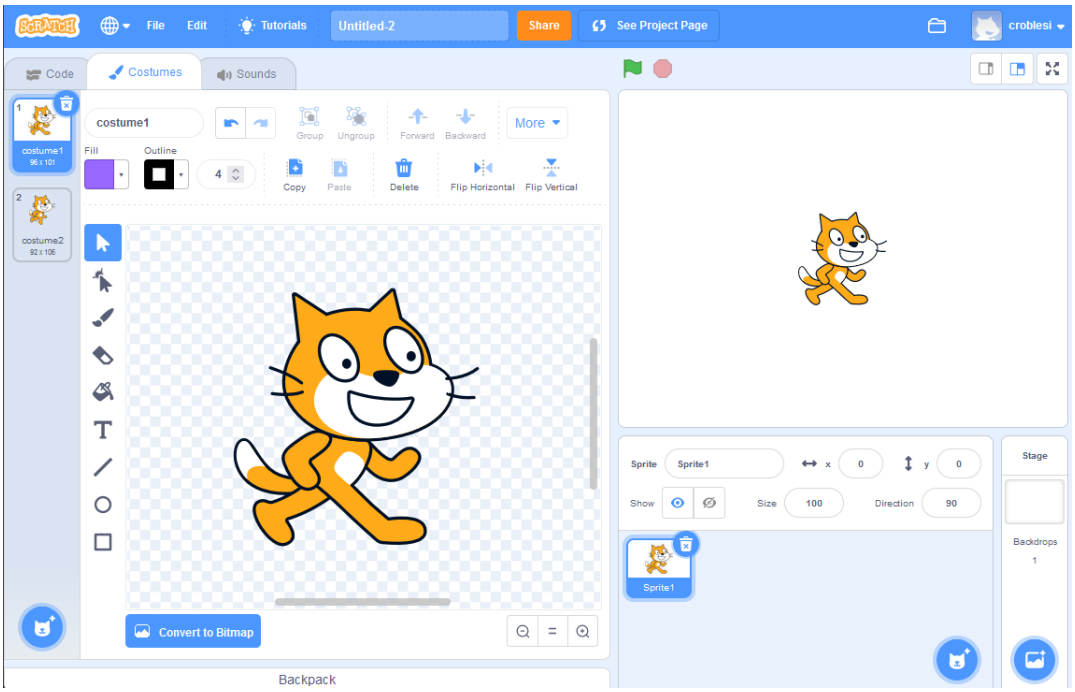

Figura A.1.1: Interfaz web de Scratch - pestaña Costumes

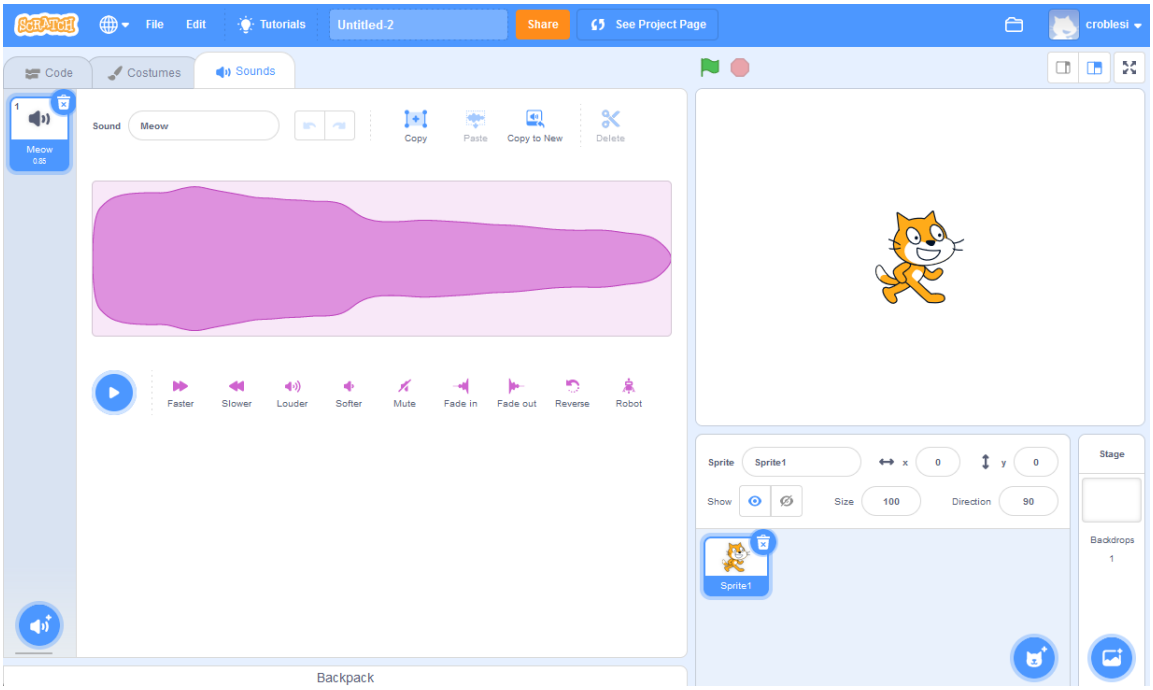

Figura A.1.2: Interfaz web de Scratch - pestaña Sounds

Anexo B

Modelo de datos

B.1. Diagrama del modelo de datos

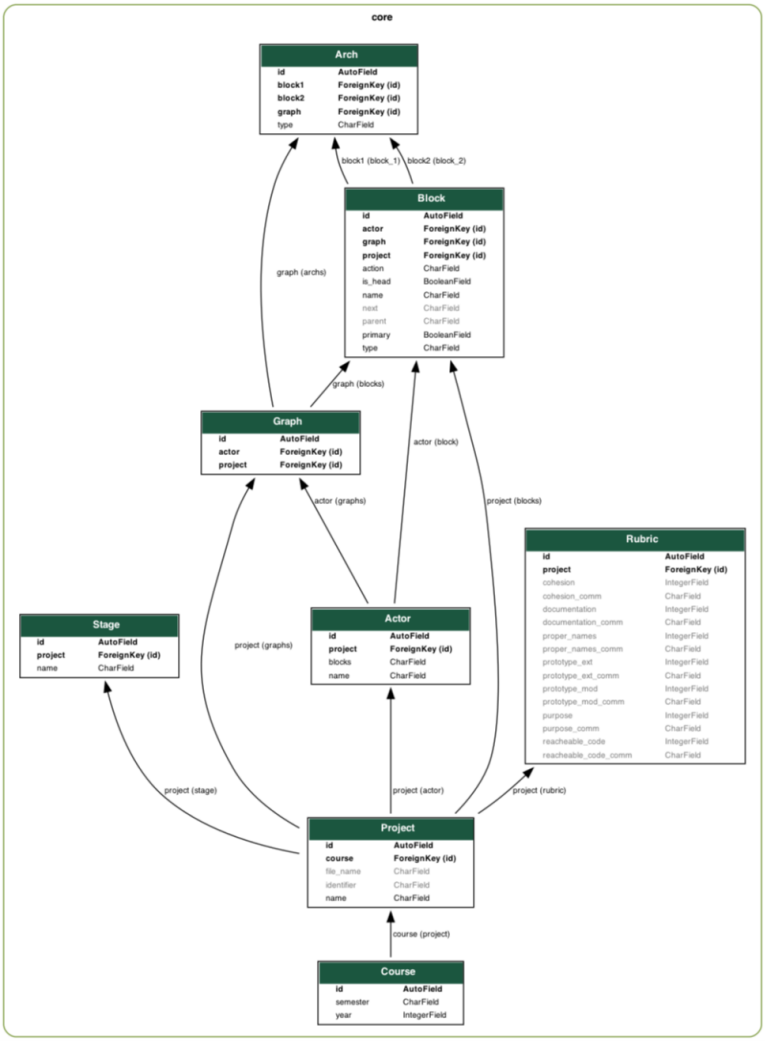

Figura B.1.1: Modelo de datos de Scratch Analyzer, definido por María José Berger [8]

Anexo C

Validación

C.1. Tabla cantidad de bloques totales

Tabla C.1.1: Comparación cantidad total de bloques por proyecto

	Scratch	**JSON**	**CGF V1**	**CFG V2**
Ejemplo 1	28	33	26	27
Ejemplo 2	28	41	29	28
Ejemplo 3	13	17	13	13
Ejemplo 4	22	32	23	22
Ejemplo 5	10	15	0	9
Proyecto 1	50	52	50	50
Proyecto 2	70	83	66	67