FACULTAD DE INFORMATICA

UNIVERSIDAD COMPLUTENSE DE MADRID

PROYECTO DE SISTEMAS INFORMATICOS

ProsymbABS
Perfilador simbólico para objetos concurrentes

Nícolas Paolo Bueno Cavero
Javier Gómez Edo

Curso 2013-2014

Directora:

Elvira Albert Albiol

FACULTAD DE INFORMATICA

UNIVERSIDAD COMPLUTENSE DE MADRID

PROYECTO DE SISTEMAS INFORMATICOS

ProsymbABS
Perfilador simbólico para objetos concurrentes

Nícolas Paolo Bueno Cavero
Javier Gómez Edo

Curso 2013-2014

Directora:

Elvira Albert Albiol

Los alumnos abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar solo con fines académicos, no comerciales, mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y/o el prototipo desarrollado.

Madrid, 23 de Junio de 2014.

Nícolas Paolo Bueno Cavero   Javier Gómez Edo

Agradecimientos

En primer lugar quisiéramos agradecer a nuestra tutora Elvira Albert Albiol por la guía y la ayuda prestada durante el desarrollo del proyecto.

A mi familia por todo el apoyo que me han dado durante toda la carrera y que me han ayudado a ser la persona que soy hoy en día. A mis amigos, tanto las nuevas amistades que me han ayudado a sobrellevar la carrera de la mejor manera posible como las amistades de toda la vida que siempre han estado ahí conmigo.

Javier

A todos mis seres queridos que siempre han estado conmigo, en especial a mi madre que siempre me ha soportado y que sin ella no habría sido esto posible. A mis hermanos y amigos, que me han brindado su apoyo.

Nícolas

A todos ellos muchas gracias.

Resumen

ProsymbABS es una aplicación desarrollada para computar el consumo de recursos simbólicos obtenidos tras ejecutar programas ABS, los cuales están basados en el paradigma de los objetos concurrentes.

Es simbólico puesto que ofrece una representación simbólica del consumo de recursos, contando, para ello, el número de objetos creados (incluyendo sus tipos), el número de llamadas a métodos y el número de instrucciones ejecutadas.

La aplicación recibe el árbol sintáctico del código ABS compilado y trabaja sobre dicho árbol para calcular los distintos recursos consumidos por la ejecución del programa, como el número de objetos creados, los métodos ejecutados por cada objeto, el número de instrucciones ejecutadas y el tiempo de ejecución.

Para facilitar el manejo de la herramienta hemos desarrollado un entorno basado en el banco de trabajo de *"Eclipse"* donde el programador puede escribir el código ABS y ver los resultados de la ejecución de ProsymbABS sobre ese código fácilmente.

Palabras clave:

- Concurrencia.
- Compilación.
- Consumo de recursos.
- Banco de trabajo.
- ABS.

Abstract

ProsymbABS is an application developed to compute the symbolic resources consumed by the execution of ABS programs, based on the paradigm of concurrent objects.

It is symbolic because it provides a symbolic representation of the resource consumption, by counting the number of objects created (including their types), the number of method calls and the number of instructions executed.

The application receives the ABS syntax tree, from the compiled code, and executes it to calculate the different resources consumed by the program execution, as the number of objects created, the methods executed by each object, the number of instructions executed and the execution time.

To facilitate the use of the tool, we have developed an execution environment based on “*Eclipse’s*” workbench where the programmer can write the ABS code and see the result of the execution of that code with the ProsymbABS tool.

Keywords:

- Concurrence.
- Compiling.
- Resource consumption.
- Workbench.
- ABS.

Tabla de figuras

Figura 1.1: Sintaxis del nivel de ABS para objetos concurrentes 10
Figura 2.1: Ejemplo código MainClass 13
Figura 2.2: Ejemplo código ObjetoConcurrente 14
Figura 2.3: Ejemplo código Contexto 15
Figura 2.4: Diagrama de clases del paquete Main 15
Figura 2.5: Ejemplo código Interfaz 16
Figura 2.6: Diagrama de clases del paquete Interfaz 17
Figura 2.7: Ejemplo de código ComandoAwait 18
Figura 2.8: Diagrama de clases del paquete Comandos 18
Figura 2.9: Ejemplo código Booleano 19
Figura 2.10: Ejemplo código Futuro 20
Figura 2.11: Diagrama de clases del paquete TipoValue 20
Figura 2.12: Ejemplo código AsyncCall 22
Figura 2.13: Ejemplo código ConstObjConcurrente 23
Figura 2.14: Ejemplo código ConstInt 23
Figura 2.15: Ejemplo código ExprGet 24
Figura 2.16: Diagrama de clases del paquete Expresiones 24
Figura 3.1: Programa de fibonacci ABS 25
Figura 3.2: Diagrama secuencial de ejecución ABS 26
Figura 3.3: Resultado ejecución total en ProsymbABS 27
Figura 3.4: Ejecución parcial en ProsymbABS 29
Figura 3.5: Resultado ejecución parcial en ProsymbABS 30
Figura 4.1: Interfaz de la aplicación 31
Figura 4.2: Pestaña Archivo 32
Figura 4.3: Selección de archivo ABS 32
Figura 4.4: Interfaz con código compilado 33
Figura 4.5: Error en el código ABS 33
Figura 4.6: Panel de planificaciones 34

Figura 4.7: Interfaz de la aplicación con código 35
Figura 4.8: Botón *play*.. 36
Figura 4.9: Panel de clases.. 37
Figura 4.10: Panel de método 38
Figura 4.11: Consola de la interfaz 39

INDICE

Agradecimientos III

Resumen IV

Abstract V

Tabla de figuras VI

1. Introducción 1

2. Estado del arte 3

  2.1. Profiling 3

  2.2. Concurrencia 3

    2.2.1. Beneficios de la programación concurrente 4

    2.2.2. Problemas inherentes a la programación concurrente 5

  2.3. Lenguaje ABS 6

    2.3.1. Abstract Behavioral Specification 7

    2.3.2. Diseño y estructura de ABS 7

    2.3.3. Modelo de concurrencia de ABS 8

    2.3.4. Nivel de objetos concurrentes del núcleo ABS 8

      2.3.4.1. Suspend 9

      2.3.4.2. Await 9

      2.3.4.3. Llamadas síncronas y asíncronas 9

      2.3.4.4. Operaciones para variables futuras 9

  2.4. Motivación 10

3. Implementación 12

  3.1. Organización de ProsymbABS 12

    3.1.1. Main 13

    3.1.2. Interfaz 15

    3.1.3. Comandos 17

    3.1.4. TipoValues 18

    3.1.5. Expresiones 21

4. Resultados obtenidos 25
5. Manual del usuario 31
5.1. Inicio 31
5.2. Edición 32
5.3. Errores 33
5.4. Planificación 34
5.5. Ejecución 35
5.5.1. Ejecución total del programa ABS 35
5.5.2. Ejecución parcial del código ABS 36
5.6. Resultados 38
6. Conclusiones 40
6.1. Trabajos futuros 41
7. Referencias y bibliografía 42
8. Apéndice 44

1. Introducción

Durante los primeros años del desarrollo de los ordenadores, el hardware representaba el coste dominante de los sistemas y debido a ello los estudios de rendimiento se concentraban en el hardware. Actualmente, según la tendencia apreciable, el software representa una porción cada vez mayor de los presupuestos informáticos, incluye el sistema operativo de multiprogramación/multiproceso, sistemas de comunicaciones de datos, sistemas de administración de bases de datos, sistemas de apoyo a varias aplicaciones. Por lo tanto, la necesidad de evaluación de las prestaciones es una consecuencia natural del aumento de la potencia y de la complejidad de los sistemas.

En los primeros tiempos los ordenadores eran concebidos para que fuesen utilizados en su totalidad por el programador (prácticamente no existía el software), y los elementos fundamentales para la medición eran la longitud de la palabra del ordenador, el conjunto de instrucciones y su implementación, el ciclo de base de la CPU, el tiempo de ejecución de una instrucción característica (Por ejemplo la instrucción sumar).

La aparición del software, de los periféricos cada vez más sofisticados, y de las unidades centrales más complejas (multiprocesadores, pipelines, memorias cache, etc.) con sistemas de interrupciones muy sofisticados, el aumento de la dimensión de las memorias, todo ello ha hecho que la evaluación del comportamiento se haya convertido en un cuerpo de doctrina en el que no sólo se ha de considerar el hardware, sino también las facilidades proporcionadas por el software al acercar la máquina a los usuarios, provocando entonces la aparición del "*overhead*" (es decir, de los gastos generales de la máquina para repartir los recursos entre los distintos usuarios) que lleva asociado todo software.

Todas estas consideraciones hacen comprender que la evaluación del comportamiento no es tarea sencilla, ya que ha de tener en cuenta muchos y variados aspectos del hardware, del software y de las aplicaciones que se han de llevar a cabo en el sistema informático. En consecuencia, evaluamos un sistema para comprobar que su funcionamiento es el correcto, es decir, el esperado.

El análisis de software es el proceso automatizado de analizar el comportamiento del software. Existen dos tipos principales de análisis, el análisis estático de software y el análisis dinámico de software. Estas técnicas de análisis intentan encontrar y mejorar en un software cuestiones de correctitud, optimización y seguridad.

- **Análisis estático de software:** se realiza sin ejecutar el programa. En la mayoría de los casos, el análisis se realiza en alguna versión del código fuente y en otros casos se realiza en el código objeto. El término se aplica generalmente a los análisis realizados por una herramienta automática, el análisis realizado por un humano es llamado comprensión de programas como también revisión de código.

- **Análisis dinámico de software:** supone la ejecución del programa y observación de su comportamiento. Para que el análisis dinámico resulte efectivo el programa a ser analizado se debe ejecutar con los suficientes casos de prueba como para producir un comportamiento interesante, se pueden usar varias estrategias de pruebas de software para lograr esto tales como cobertura de código o simplemente programas conocidos como "*fuzzers*" que ayudan a asegurar que una porción adecuada del conjunto de posibles comportamientos del programa ha sido observada.

2. Estado del arte

2.1. Profiling

El análisis de rendimiento, comúnmente llamado ***profiling***, es la investigación del comportamiento de un programa de computadora usando información reunida desde el análisis dinámico del mismo. El objetivo es averiguar el consumo de recursos obtenidos tras la ejecución de diferentes partes del programa para detectar los puntos problemáticos y las áreas donde sea posible llevar a cabo una optimización del rendimiento (ya sea en velocidad o en consumo de recursos). Un *profiler* puede proporcionar distintas salidas, como una traza de ejecución o un resumen estadístico de los eventos observados.

Normalmente el *profiling* es utilizado durante el desarrollo de software como método para la depuración y optimización de los algoritmos, esta práctica vista de esta manera es buena, pero es vista más como una actividad interna que suele carecer de información cuando no es evaluado por personal realmente especializado y en el entorno adecuado para ello.

El *profiling* se puede llevar a cabo en el código fuente o sobre un binario ejecutable mediante una herramienta llamada "*profiler*".

En 1994, Amitabh Srivastava y Alan Eustace de Digital Equipment Corporation publican un artículo describiendo ATOM. ATOM es una plataforma para convertir un programa en su propio profiler. Es decir, en tiempo de compilación, inserta el código en el programa a ser analizado. Ese código introducido produce salidas de datos de análisis.

> *"Las herramientas de análisis de programas son extremadamente importantes para entender el comportamiento de los programas. Los arquitectos informáticos necesitan esas herramientas para evaluar cómo rendirán los programas en nuevas arquitecturas. Los desarrolladores de software necesitan herramientas para analizar sus programas e identificar secciones críticas de código. Los desarrolladores de compiladores a menudo utilizan herramientas para saber cómo funcionará el algoritmo de predicción de saltos o la planificación de las instrucciones."*
>
> \- Amitabh Srivastava y Alan Eustace – ATOM, 1994

2.2. Concurrencia

El paradigma de la concurrencia consiste en dos o más procesos que entrelazan su ejecución. No tienen por qué ejecutarse exactamente al mismo tiempo, simplemente

es suficiente con el hecho de que existe un intercalado entre la ejecución de sus instrucciones. Si se ejecutan al mismo tiempo los dos procesos, entonces tenemos una situación de programación paralela.

Un programa, al ponerse en ejecución, puede dar lugar a más de un proceso, cada uno de ellos ejecutando una parte del programa. Podemos definir un proceso como una actividad asíncrona susceptible de ser asignada a un procesador. Cuando varios procesos se ejecutan concurrentemente puede haber procesos que colaboren para un determinado fin, mientras que puede haber otros que compitan por los recursos del sistema. Para llevar a cabo estas tareas de colaboración y competencia por los recursos se hace necesaria la introducción de mecanismos de comunicación y sincronización entre procesos.

2.2.1. Beneficios de la programación concurrente

Existen diversos motivos por los que la programación concurrente es útil. Por ejemplo:

- **Velocidad de ejecución:** Cuando la ejecución de un programa conlleva la creación de varios procesos y el sistema consta de más de un procesador, existe la posibilidad de asignar un proceso a cada procesador de tal forma que el programa se ejecuta de una forma más rápida.

- **Solución de problemas inherentemente concurrentes:** Hay problemas cuya solución es más fácil de abordar mediante el uso de programación concurrente. Destacamos los que se nombran a continuación:

    - *1. Sistemas de control:* Los sistemas de control son aquellos sistemas en los que hay una captura de datos. La recolección de datos se puede estar haciendo de diversas entidades físicas como por ejemplo en edificios o en estancias dentro de un edificio. No sería tolerable un sistema secuencial que vaya capturando los datos uno a uno de las distintas estancias. Podría haber un incendio en el edificio y si los datos se tomarán de forma secuencial, mientras que se capturan los datos de la última estancia, podría haberse quemado la primera. Por ello, tanto la captura de los datos como su tratamiento y posterior actuación son candidatos a ser procesos distintos y de naturaleza concurrente.

    - *2. Tecnologías Web:* La mayoría de los programas relacionados con la web son concurrentes. Servidores web que son capaces de atender concurrentemente múltiples conexiones de usuarios, programas de chat que permiten mantener la conversación de varios usuarios, servidores de correo que permiten que múltiples usuarios puedan mandar y recibir mensajes al mismo tiempo, navegadores que permiten que un usuario pueda estar haciendo una descarga mientras navega por otras páginas, etc.

    - *3. Aplicaciones basadas en interfaces de usuarios:* La concurrencia en este tipo de aplicaciones va a permitir que el usuario pueda interaccionar con la aplicación aunque ésta esté realizando alguna tarea que consume mucho tiempo de procesador. Un proceso controla la

interfaz mientras otro hace la tarea que requiere un uso intensivo de la CPU. Esto facilitará que tareas largas puedan ser abortadas a mitad de ejecución.

- *4. Simulación:* Los programas secuenciales encuentran problemas al simular sistemas en los que existen objetos físicos que tienen un comportamiento autónomo independiente. La programación concurrente permitirá modelar esos objetos físicos y ponerlos en ejecución de forma independiente y concurrente, sincronizándolos de la forma apropiada.

- *5. SGBD:* En Sistemas Gestores de Bases de Datos la concurrencia juega un papel muy importante cuando se va a permitir a varios usuarios interactuar con el sistema. Cada usuario puede ser visto como un proceso. Obviamente hay que implementar la política adecuada para evitar situaciones en las que dos usuarios modifican al mismo tiempo un registro. Sin embargo, a varios usuarios que quieran acceder a un mismo registro para consultarlo y no modificarlo, debe permitírseles un acceso concurrente.

- **Mejor rendimiento.**

2.2.2. Problemas inherentes a la programación concurrente

Hay tres problemas inherentes a la programación concurrente que son:

- **EXCLUSIÓN MUTUA**

Cuando dos procesos P1 y P2 se ejecutan concurrentemente están accediendo al mismo tiempo a una variable compartida entre los dos para actualizarla. Nos interesa que todas las líneas de código de un proceso se ejecuten en un solo paso sin ningún tipo de intercalado con las otras líneas del otro proceso. A la porción de código que queremos que se ejecute de forma indivisible se le denomina **sección crítica**. Nos interesa asegurarnos que las secciones críticas se ejecuten en **exclusión mutua**, es decir, sólo uno de los procesos debe estar en la sección crítica en un instante dado.

La programación concurrente debe proporcionarnos mecanismos para especificar que partes del código han de ejecutarse en exclusión mutua con otras partes. La exclusión mutua es un problema derivado de la abstracción en los lenguajes de alto nivel. Una instrucción de alto nivel se convierte en un conjunto de instrucciones máquina. Estas instrucciones máquina son las que realmente se ejecutan concurrentemente. En el paradigma secuencial este hecho carece de importancia pero en ejecuciones concurrentes de instrucciones el resultado puede ser incorrecto.

- **CONDICIÓN DE SINCRONIZACIÓN**

Hay situaciones en las que el recurso compartido por varios procesos, se encuentra en un estado en el que un proceso no puede hacer una determinada acción con él hasta que no cambie su estado. A esto se le denomina condición de sincronización. La programación concurrente ha de proporcionarnos mecanismos para bloquear procesos que no puedan hacer algo en un momento determinado a la espera de algún evento, pero también permita desbloquearlos cuando ese evento haya ocurrido.

- **VERIFICACION**

En los últimos años, la demanda de software fiable y de calidad ha crecido mucho más rápido que la tecnología necesaria para su producción. La complejidad y el tamaño de los programas actuales exigen el uso de herramientas formales ágiles que aseguren la construcción de software correcto. Asimismo, con el crecimiento explosivo de la “tecnología de la información”, cobran enorme relevancia los análisis y demostraciones de propiedades de seguridad, que puedan ayudar a evitar, por ejemplo, graves errores en protocolos de comunicación y servicios web. Los lenguajes utilizados en este ámbito (XML, HTML, etc.) plantean el reto de adaptar convenientemente las técnicas clásicas de verificación, depuración, para abordar este tipo de problemas.

Las técnicas de verificación de programas permiten demostrar que un programa satisface sus especificaciones. Las técnicas de depuración de programas permiten detectar las causas que introducen discrepancias entre el comportamiento real y el comportamiento que se espera de un programa ayudando a corregir, en su caso, las disfunciones encontradas. Puesto que tanto la verificación como la depuración de programas necesitan considerar algún tipo de representación computable de la semántica de los sistemas considerados, la interpretación abstracta, como marco general que permite diseñar, aproximar y comparar semánticas de programas que expresan varios tipos de propiedades observables aporta una componente esencial que añadir a las técnicas de verificación y depuración.

El propósito de este proyecto se va a centrar en resolver los problemas derivados del último punto. Es interesante observar el comportamiento de los programas a la hora de repartir el consumo de recursos entre los distintos procesadores ya que puede haber procesadores que reciban una alta carga de trabajo y otros que no estén trabajando. Se puede conseguir una gran optimización en el código de los programas concurrentes conociendo su consumo de recursos.

ProsymbABS es un perfilador dedicado al estudio del consumo de recursos de aplicaciones concurrentes, basados en el lenguaje ABS.

2.3. Lenguaje ABS

ABS (“*Abstract behavioral specification*”) es un lenguaje para los el modelado y desarrollo de sistemas distribuidos y concurrentes. ABS permite una especificación de alto nivel de un sistema, incluyendo su concurrencia y mecanismos de sincronización, así como las actualizaciones de estados locales. Los modelos de ABS capturan el flujo

de control concurrente abstrayendo varios detalles de implementación que resultaría ser ignorado a nivel de modelado, tales como la representación concreta de estructuras de datos internas, la planificación de las activaciones de método y las propiedades de los entornos de comunicación.

ABS está dirigido a sistemas distribuidos. En un entorno distribuido, los detalles de implementación de los demás objetos del sistema no son necesariamente conocidos. En vez de ello, ABS utiliza interfaces como tipos para los objetos, abstrayendo el tipo de sistema de las clases que implementan la funcionalidad de los distintos objetos.

ABS soporta llamadas asíncronas a métodos, las cuales pueden mandar tareas a otros objetos sin transferir el control de la llamada, usando para ello los tipos **futuros**. Una característica de este modelo de concurrencia es la planificación cooperativa de activaciones de métodos para controlar de forma explícita el intercalado interno de las tareas dentro de los objetos concurrentes.

2.3.1. Abstract Behavioral Specification

Los lenguajes de especificación se pueden dividir en tres categorías:

- **Lenguajes orientados al diseño**: se centran en los aspectos estructurales de los sistemas, así como en la relación entre clases y el control de mensajes entre objetos. Ejemplos de lenguajes orientados al diseño son UML/OCL, FDL y lenguajes de descripción arquitectural.

- **Lenguajes declarativos**: se basan es aspectos funcionales como concurrencia e interacción, identificando un pequeño conjunto de primitivas y su semántica formal. Ejemplos de lenguajes declarativos son los modelos de autómatas.

- **Lenguajes orientados a la implementación**: se centran en las propiedades del comportamiento de sistemas implementados. Ejemplos de lenguajes orientados a la implementación son JML y Spec#.

ABS se sitúa en esas tres categorías. Comparte con los lenguajes orientados a la implementación que está diseñado para que se acerque a la manera de pensar de los programadores, es decir, es un lenguaje de alto nivel, ya que mantiene una sintaxis parecida a la de Java. Por otro lado, el lenguaje tiene unas semánticas formalmente definidas, al estilo de los lenguajes declarativos, que permiten al modelador abstraer detalles de implementación innecesarios debido a definiciones hechas por el usuario de tipos de datos y funciones.

2.3.2. Diseño y estructura de ABS

El modelo de concurrencia de ABS se divide en dos niveles y separa la comunicación local, síncrona y memoria compartida en el nivel inferior, de la comunicación asíncrona sólo con paso de mensaje en el nivel superior.

El nivel inferior está basado en JCoBox [1] el cual generaliza el modelo de concurrencia de Creol [2] sobre los grupos de objetos concurrentes llamados *cogs*. Los *cogs* se pueden interpretar como componentes de ejecución basados en objetos con sus

propios *heaps*. El comportamiento del *cog* se basa en una multitarea de peticiones externas y activaciones de métodos internos. Únicamente las llamadas asíncronas pueden ocurrir entre diferentes *cogs*, los cuales no comparten el mismo *heap*.

Aparte del lenguaje de objetos concurrentes, ABS soporta tipos definidos por el usuario con funciones de primer orden. Se utilizan los tipos funcionales para simplificar significativamente la especificación y verificación de programas ABS.

2.3.3. Modelo de concurrencia de ABS

Conceptualmente, cada *cog* tiene un procesador dedicado y está ubicado en un entorno distribuido con comunicación asíncrona y desordenada.

El conjunto de objetos está ubicado dentro del *cog*. En el nivel superior del modelo de concurrencia de ABS toda la comunicación se realiza entre los objetos creados, los cuales están tipificados por interfaces, mediante llamadas asíncronas de métodos. Las llamadas son asíncronas hasta que el objeto llamante decide en tiempo de ejecución cuando sincronizar la respuesta de tal llamada. Las llamadas asíncronas de métodos se pueden ver como disparadores de actividad concurrente, creando nuevas tareas (también llamados procesos) en el objeto llamado.

ABS mezcla un comportamiento activo, caracterizado por un método principal, con un comportamiento pasivo, caracterizado por llamadas asíncronas de métodos. Por consiguiente, un objeto con un conjunto de procesos a ejecutar, originado por varias llamadas asíncronas. Entre todos esos procesos al menos uno de los procesos de entre todos los objetos del *cog* está activo y el resto están en suspensión en un inventario de procesos.

El nivel de concurrencia de un programa ABS se refleja directamente en el número de *cogs* introducidos en el programa. Un modelo Creol de concurrencia corresponde a un programa ABS donde cada objeto tiene su propio *cog*.

2.3.4. Nivel de objetos concurrentes del núcleo ABS

La sintaxis del nivel de objetos concurrentes del núcleo ABS se puede observar a continuación:

Syntactic categories	**Definitions**
C, *I*, m in Names	$P ::= \overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x}; s\}$
g in Guard	$IF ::= \textbf{interface}\ I\ \{\ \overline{Sg}\ \}$
s in Statement	$CL ::= \textbf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\textbf{implements}\ \overline{I}\ ]\ \{\overline{T}\ \overline{x};\ \overline{M}\ \}$
	$Sg ::= T\ m\ (\overline{T}\ \overline{x})$
	$M ::= Sg\ \{\overline{T}\ \overline{x}; s\}$
	$g ::= \mathrm{b} \mid e? \mid g \wedge g$
	$s ::= s;\ s \mid x = rhs \mid \textbf{suspend} \mid \textbf{await}\ g \mid \textbf{skip}$
	$\mid \textbf{if}\ b\ \{s\}\ [\textbf{else}\ \{s\}] \mid \textbf{while}\ b\ \{s\} \mid \textbf{return}\ e$
	$rhs ::= e \mid \textbf{new}\ [\textbf{cog}]\ C[(\overline{e})] \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.\textbf{get}$

El bloque principal de un programa equivale a la especificación del cuerpo de un método ({$\overline{T}\ \overline{x}$; s}). Las expresiones de ABS incluyen la creación de objetos en el mismo *cog* (escrito "*new C* ($\overline{e}$)"), creación de objetos en nuevos *cogs* (escrito "*new cog C*($\overline{e}$)"), llamadas a métodos, expresiones de asignación (booleanas, enteras, cadenas).

2.3.4.1. Suspend

La primitiva ***suspend*** incondicionalmente libera al procesador suspendiendo el proceso activo.

2.3.4.2. Await

En la expresión "***await*** *g*" la guarda *g* controla la liberación del proceso. Consiste en una condición booleana que cuando se evalúa a falso, el procesador se libera y el proceso activo se suspende. Cuando el procesador se encuentra libre, un proceso disponible en el conjunto de procesos suspendidos se ejecutará.

2.3.4.3. Llamadas síncronas y asíncronas

La comunicación de ABS se basa en llamadas asíncronas, denotadas por la expresión "*o!m(e)*" y en llamadas síncronas, denotadas por la expresión "*o.m(e)*", donde "*o*" corresponde al objeto llamante, el signo "*!*" denota una llamada asíncrona y el signo "*.*" denota una llamada asíncrona, "*m*" corresponde al nombre del método que se quiere llamar y "*e*" corresponde a los parámetros de entrada del método. Cualquier método se puede llamar tanto síncrona como asíncronamente.

Tras realizar la llamada asíncrona "*x = o!m(e)*", el procesador llamante seguirá con su ejecución sin parar para realizar la llamada. Aquí "*x*" corresponde a una variable **futura**, es decir, una variable que corresponde al retorno de un valor el cual aún no se ha calculado.

2.3.4.4. Operaciones para variables futuras

Existen dos tipos de operaciones que se realizan en variables futuras que controlan explícitamente la sincronización externa de ABS.

La primera operación corresponde a un test de retorno que se ejecuta mediante el comando "***?***". Siendo "*e*" una expresión que denota una variable futura, la expresión "*e**?***" se evaluará a falso hasta que se responda a la llamada que devuelva el valor calculado de "*e*" (estos test de retorno se utilizan principalmente en las guardas (comando ***await***)).

La segunda operación recupera el valor de retorno de la variable futura mediante la expresión, en este caso, "*e.**get***", la cual bloquea la ejecución del procesador hasta que el valor de retorno esté disponible.

En la página de HATS (http://tools.hats-project.eu/) [5], podemos encontrar un manual [6] más detallado de la sintaxis de ABS.

Figura 1.1: Sintaxis del nivel de ABS para objetos concurrentes

2.4. Motivación

El hardware ha seguido durante años la Ley de Moore. Las velocidades de reloj se incrementaron y el software se escribía para explotar este crecimiento incesante en el rendimiento, a menudo por delante de la curva de los equipos físicos. Esa relación simbiótica entre hardware y software continuó sin cesar hasta hace muy poco. La Ley de Moore sigue vigente, pero se ha obviado la ley no escrita que dice que las velocidades de reloj seguirán aumentando proporcionalmente siempre.

Las mejoras en el rendimiento de la memoria se quedan cada vez más por detrás de la productividad del procesador, haciendo que el número de ciclos necesario de la CPU para acceder a la memoria principal no pare de crecer continuamente. Esto es lo que se conoce como "Pared de memoria" (*Memory Wall*).

Los ingenieros de hardware han mejorado el rendimiento del software secuencial haciendo que las instrucciones se ejecuten antes de que se conozcan los resultados de las instrucciones anteriores, una técnica conocida como Paralelismo a nivel de instrucción (ILP son sus siglas en inglés). Las novedades en ILP son difíciles de predecir, y su complejidad aumenta el consumo de energía. Como resultado de ello, las mejoras en ILP también se han estancado, lo que resulta en la llamada "Pared ILP" (*Wall ILP*).

Hemos llegado, pues, a un punto de inflexión. El ecosistema de software debe evolucionar para apoyar mejor y adaptarse correctamente a los sistemas de varios núcleos, y esta evolución llevará tiempo. Para beneficiarse de la rápida mejora en el rendimiento de los equipos deberá ejecutarse cada vez más rápido en cada nuevo hardware. La comunidad de desarrolladores debe aprender a construir aplicaciones simultáneas multi-hilo. La importancia de la arquitectura y el diseño limpio es fundamental para la reducción de la complejidad del software, y mejora la capacidad y facilidad de mantenerlo. Hay que poner énfasis en la comprensión, no sólo de las capacidades de la plataforma, sino también de las mejores prácticas emergentes.

Para el desarrollo de ProsymbABS se ha tenido en cuenta todos los aspectos relacionados con el *profiling* en sistemas concurrentes. Puesto que el desarrollo del hardware está llegando a su límite físico, se postula como necesidad la optimización del software para mejorar el rendimiento de los programas. En concreto hay que tener en cuenta la importancia de la concurrencia en el desarrollo software debido a que una óptima implementación del código puede aprovechar todos los recursos proporcionados por el ordenador.

En Java existen perfiladores como JProfiler [3] o el JBoss [4], pero el análisis de los programas concurrentes nos llamó bastante la atención para desarrollar desde el principio un perfilador con las características y la estructura de datos más cómoda para nosotros de obtener y guardar la información a estudiar. Con ello se pueden ampliar los campos que abarca ProsymbABS.

ProsymbABS se va a centrar en el análisis del rendimiento (o consumo de recursos) del software. Se tratará de un análisis dinámico ya que necesitamos ejecutar el programa para observar el comportamiento de su ejecución para calcular los recursos consumidos. Como el proyecto consiste en un perfilador simbólico, guardaremos la información que recojamos como información simbólica, como el número de los objetos creados (y sus respectivos tipos), el número de llamadas a métodos y el número de instrucciones ejecutadas.

3. Implementación

Para desarrollar nuestro proyecto partimos del *parser* de ABS, el cual nos devuelve:

- **Detalle de las Interfaces, clases y métodos**: A partir de esta información se pueden crear las clases de las que se compone el programa ABS. Con esta información obtenemos:
    - Nombre de la clase.
    - La interfaz que implementa.
    - Atributos de clase.
    - Sus método, de los método obtenemos:
        - Nombre del método.
        - Parámetros de entrada.
        - Atributos de método.
        - Tipo y atributo de retorno.
        - Instrucciones que representamos cada una como comandos.
- **Detalle del bloque principal (método Main)**: Aquí se recoge la información de las instrucciones que se ejecutan en el bloque principal, o método Main. Para ello se crea un Objeto Concurrente que reúne todas las instrucciones y las variables que se declaren en este método (incluidos los nuevos Objetos Concurrentes que se declaren).
- **Los errores Sintácticos y Semánticos del programa**: Se obtienen los errores que el *parser* de ABS encuentra en el código. Devuelve la línea donde se produce el error y un comentario de error.
- **Librerías**: Al igual que en Java, ABS puede importar distintas librerías. Sin embargo, no hemos diseñado esta funcionalidad para nuestra aplicación.

ProsymbABS está diseñado en lenguaje Java y se compone de varias clases en las que organizamos toda la información que nos devuelve el *parser* de ABS.

3.1. Organización de ProsymbABS

La primera tarea de la aplicación es interpretar la información obtenida del *parser* por lo que, básicamente, la aplicación se podría definir como un intérprete del árbol sintáctico de ABS. Para ello, el código del proyecto se divide en 5 paquetes principales, uno de ellos correspondiente a la interfaz de ProsymbABS y los demás se dedican a la organización de la información recogida y a la funcionalidad de la aplicación. La ventaja de este *parser* es que gracias a que hace la compilación del código se sabe si hay errores en el código ABS, por lo que no se ejecutará el programa hasta que se hayan corregido.

3.1.1. Main

En este paquete se encuentran las clases principales de la aplicación. La clase **MainClass** realiza la interpretación del *parser* ABS, obteniendo los errores, si hay, del programa ABS, las diferentes clases con sus métodos, los objetos y objetos concurrentes que se van creando. Los objetos concurrentes se guardan en una lista que será la que se ejecute cuando se ejecute la aplicación.

```
public MainClass(String args) {
    Main absCompiler = new Main();
     //String[] file = {"Suma.abs"};
     String[] file = {args};

     try {
         Model model = absCompiler.parse(file);
         Block b = model.getMainBlock();
         CompilationUnit c = model.getCompilationUnit(1);
         if (model.hasErrors() || model.hasParserErrors() || model.hasTypeErrors())
             System.out.println("Hay Errores");
         parsearErrores(model.getErrors(), model.getParserErrors(), model.getTypeErrors());
         System.out.println(c.toString());
         List<Decl> listaDecl = c.getModuleDeclList().getChild(0).getDeclList();
         int n = listaDecl.getNumChild();
         for (int i = 0;i<n;i++) {
             Decl d = listaDecl.getChild(i);
             String tipoClase = d.getClass().toString();
             if (tipoClase.contains("ClassDecl")) {
                 ClassDecl cd = (ClassDecl) d;
                 parsearClase(cd);
             }
         }
         //id 0 indica objetoConcurrente main
         Clase claseMain = new Clase();
         claseMain.setNombreClase("Main");
         tablaClases.put("Main", claseMain);
         ObjetoConcurrente main = new ObjetoConcurrente(0,"Main", "Main", new HashMap<String, TipoValue>());

         Metodo metodoMain = new Metodo();
         metodoMain.setNombreClase("Main");
         String nombreMetodo = "metodoMain";
         metodoMain.setNombreMetodo(nombreMetodo);
         metodoMain.setRetorno(null);
         metodoMain.setTablaAtr(new HashMap<String,TipoValue>());
         metodoMain.setOrdenAtr(new ArrayList<String>());
         Queue<Comando> q = parsearBlock(b);
         metodoMain.setcolaCom(q);
         claseMain.getTablaMetodos().put(nombreMetodo, new Metodo(metodoMain));
```

Figura 2.1: Ejemplo código MainClass

Los **Objetos Concurrentes** son las piezas en torno a las que gira la aplicación. Cada uno de estos objetos guarda la información de la clase a la que pertenecen y una cola de los métodos que van a ejecutar en el programa ABS. Estos métodos los hemos identificado como la clase **Tarea**, donde se guarda el método a ejecutar y sus parámetros de entrada. El método a ejecutar es una clase de tipo **Metodo** el cual guarda la información sobre la clase a la que pertenece, los parámetros de entrada del método y la cola de comandos de la que se compone dicho método. Las Tareas ejecutan los comandos que contiene el método, los cuales pueden crear nuevos

Objetos Concurrentes, crear tareas asíncronas en los otros objetos concurrentes o en el mismo que lo invoca.

```
public void ejecutar(Contexto context) {
    context.setCamposClase(campos);
    boolean await = false;
    while (siguienteTarea(context)) {
        //context.setTareaActual(procesoActivo);
        procesoActivo.ejecutar(context);
        if (context.getPararAwait() && colaTareas.size() == 1) {
            context.setPararAwait(false);
            await = true;
            break;
        }
        context.setPararAwait(false);
    }
    List<ObjetoConcurrente> l = context.getListaObjEjecucion();
    if (context.getPararGet() || await) {
        ObjetoConcurrente o = l.get(0);
        l.remove(this);
        l.add(o);
    } else {
        context.getTablaObj().remove(id);
        context.getTablaObjInactivos().put(id, this);
        l.remove(this);
        System.out.println("Numero de llamadas Asincronas de la clase" + nombreClase + " : " + asynCallObj);
    }
}
```

Figura 2.2: Ejemplo código ObjetoConcurrente

Clase guarda la información de las clases declaradas en el programa ABS. Cada clase define sus propios métodos, por lo que nuestra estructura de datos guarda en **Clase** una tabla con los métodos y los atributos de las clases que el usuario declara en el programa ABS.

Por último, este paquete se compone de la clase **Contexto**, el cual es la principal característica del perfilador. **Contexto** es una estructura de datos que guarda toda la información que la clase **MainClass** va interpretando del *parser* ABS. **Contexto** se desarrollo para guardar esta información de forma que después del compilado del código ABS fuera relativamente sencilla la ejecución del programa y la recolección de los datos del consumo de recursos de ABS por parte del perfilador. En esta clase guardamos todas las clases creadas, todos los objetos concurrentes creados y gran variedad de variables para controlar el hilo de ejecución definido por la planificación que se haya definido antes de la ejecución de la aplicación.

```java
public class Contexto {
    int contID = 1;
    Map<Integer,ObjetoConcurrente> tablaObj = new HashMap<Integer,ObjetoConcurrente>();
    Map<Integer,ObjetoConcurrente> tablaObjInactivos = new HashMap<Integer,ObjetoConcurrente>();
    List<ObjetoConcurrente> listaObjEjecucion = new ArrayList<ObjetoConcurrente>();
    ObjetoConcurrente ObjConcurrenteActivo = null;
    Metodo mActual = null;
    Map<String,Clase> tablaClases = new HashMap<String,Clase>();
    Map<String, TipoValue> camposClase = new HashMap<String,TipoValue>();
    Map<String, TipoValue> camposMetodo = new HashMap<String,TipoValue>();
    TipoValue retorno = null;
    boolean pararGet = false;
    boolean pararAwait = false;
    int instruccionesTotales = 0;
    int newCogs = 0;
    int newObj = 0;
    int numLlamadasAsync = 0;

    public Contexto(Map<Integer,ObjetoConcurrente> tablaObj,List<ObjetoConcurrente> listaObjEjecucion,Map<String,Clase> tablaClases) {
        this.tablaObj = tablaObj;
        this.listaObjEjecucion = listaObjEjecucion;
        this.tablaClases = tablaClases;
    }
```

Figura 2.3: Ejemplo código Contexto

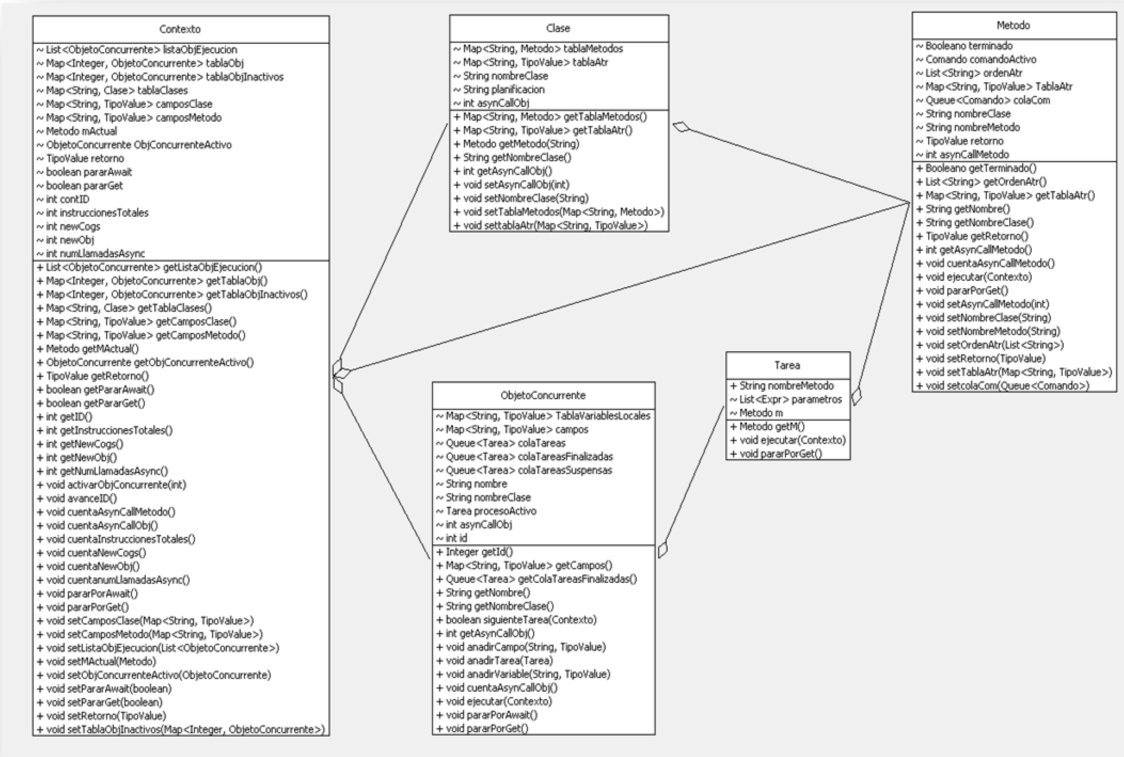

Figura 2.4: Diagrama de clases del paquete Main

3.1.2. Interfaz

Este paquete se compone de las clases que se encargan de diseñar la interfaz de la aplicación.

Las clases **LinePainter** y **LinePainterError** son básicamente el mismo código, la diferencia es que el primero se encarga en colorear la línea donde está el cursor apuntando, y el segundo se encarga de colorear las líneas donde el *parser* ABS ha encontrado errores.

La clase **Interfaz** tiene como atributo un objeto de tipo **MainClass**, el cuál es el intérprete que estará compilando el código con cada cambio que se haga en el panel central, donde aparece el código ABS. El usuario podrá ejecutar el código compilado en el botón con la imagen de *play* siempre y cuando la compilación no haya producido errores, o ejecutando un método en particular, utilizando el panel de la derecha, y el resultado de la información del consumo de recursos se mostrará en el panel inferior.

```
JTabbedPane tabbedPane = new JTabbedPane(JTabbedPane.TOP);
tabbedPane.setBounds(205, 509, 1151, 205);
contentPane.add(tabbedPane);

JScrollPane scrollPane_2 = new JScrollPane();
tabbedPane.addTab("Consola", null, scrollPane_2, null);

textArea_1 = new JTextArea();
textArea_1.setEditable(false);
scrollPane_2.setViewportView(textArea_1);

JTabbedPane tabbedPane_1 = new JTabbedPane(JTabbedPane.TOP);
tabbedPane_1.setBounds(205, 35, 966, 460);
contentPane.add(tabbedPane_1);

JScrollPane scrollPane = new JScrollPane();
tabbedPane_1.addTab("ABS", null, scrollPane, null);

textPane = new JTextPane();
textPane.getDocument().addDocumentListener(docListener);
scrollPane.setViewportView(textPane);
LinePainter painter = new LinePainter(textPane);

tabbedPane_2 = new JTabbedPane(JTabbedPane.TOP);
tabbedPane_2.setBounds(10, 35, 185, 679);
contentPane.add(tabbedPane_2);

scrollPane_1 = new JScrollPane();
scrollPanePlan = new JScrollPane();
tabbedPane_2.addTab("Clases", null, scrollPane_1, null);
```

Figura 2.5: Ejemplo código Interfaz

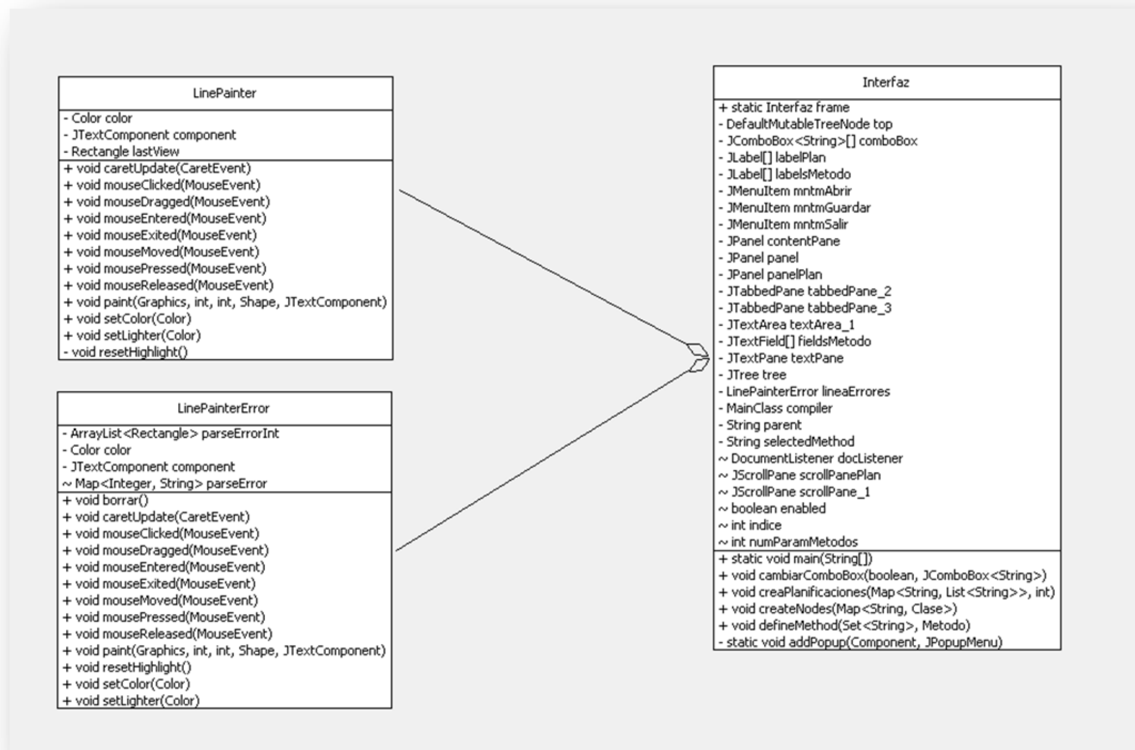

Figura 2.6: Diagrama de clases del paquete Interfaz

3.1.3. Comandos

Cada una de las instrucciones que se ejecutan en el programa ABS y que el *parser* de ABS nos devuelve, se clasifica como comando dependiendo del tipo de comando que sea (asignación, llamada asíncrona, if, while, etc).

El paquete se compone de una clase abstracta **Comando** con un método *ejecutar* que definirán todas las clases que hereden de la misma. De entre todos los tipos de comandos que se han diseñado para el proyecto (**Asignación, ComandoAsyncCall, ComandoAwait, ComandoIf, ComandoSyncCall, ComandoWhile, Return**) cabe destacar los siguientes:

- **ComandoAsyncCall**: se encarga de hacer las llamadas asíncronas que realizan los objetos concurrentes. Para ello crea una nueva tarea en la cola de tareas del objeto concurrente que realice la llamada.
- **ComandoAwait**: realiza las mismas funciones que la primitiva *await* de ABS. Consta de una expresión booleana que para la ejecución del objeto concurrente que lance este comando hasta que la expresión se evalúe a cierto.
- **ComandoSyncCall**: se encarga de realizar una llamada síncrona a un método, pero no realiza ninguna espera ni lo añade como tarea al objeto concurrente.

```java
import java.util.Iterator;

public class ComandoAwait extends Comando{
    Expr e;

    public ComandoAwait(Expr e) {
        this.e = e;
    }

    public void ejecutar(Contexto context) {
        context.cuentaInstruccionesTotales();
        Booleano b = (Booleano)e.eval(context);
        if (!b.getB()){
            context.pararPorAwait();
            context.setPararAwait(true);
        }
    }

}
```

Figura 2.7: Ejemplo de código ComandoAwait

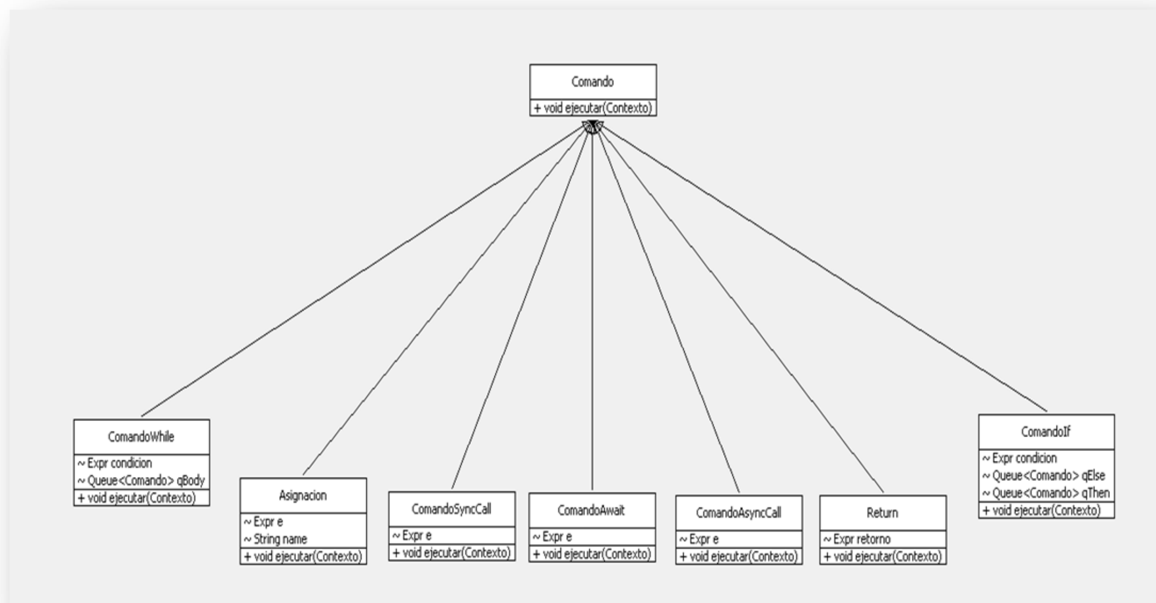

Figura 2.8: Diagrama de clases del paquete Comandos

3.1.4. TipoValues

Todos los tipos que la aplicación va a evaluar se encuentran en este paquete. La clase abstracta **TipoValue** es el padre de las demás clases que aparecen en el paquete. De entre todos los tipos especificados (**Booleano, Cadena, Concurrente, Entero, Futuro, Objeto, Operador**) cabe destacar los siguientes:

- **Booleano, Cadena, Entero**: son los tipos más comunes que aparecen en los programas ABS. Heredan de **TipoValue** el método ejecutar donde, dependiendo del operador que se le pase al método como parámetro, realiza una función u otra, como por ejemplo, la suma o la resta, la concatenación de cadenas, las operaciones *or* y *and*.

- **Futuro**: este tipo se encarga de tratar los tipos futuros de ABS. Constan de dos atributos, el valor y un booleano de acabado, de forma que cuando valor se rellene, el booleano pasará a ser cierto por lo que se podrá obtener su valor con la instrucción *get*.

- **Operador**: se encarga de guardar el operador de las operaciones unarias y binarias que se ejecutan en el código ABS.

```
public TipoValue ejecutar(String operacion, TipoValue v) {
    if (operacion.equals("&&")) {
        Booleano e = new Booleano();
        e.boolAnd(this,(Booleano)v);
        return e;
    } else if (operacion.equals("||")) {
        Booleano e = new Booleano();
        e.boolOr(this,(Booleano)v);
        return e;
    } else if (operacion.equals("not")) {
        Booleano e = new Booleano();
        e.boolNot(this);
        return e;
    }
    return this;
}
```

Figura 2.9: Ejemplo código Booleano

```java
public class Futuro extends TipoValue {
    TipoValue valor;
    Booleano terminado = new Booleano(false);

    public Futuro() {valor = null;}
    public Futuro(TipoValue v, Booleano terminado) {
        valor = v;
        this.terminado = terminado;
    }

    public Futuro (Futuro f) {valor = f.valor;}

    public TipoValue get(){
        if (terminado.getB()){return valor;}
        else return null;
    }
    public TipoValue ejecutar(String operacion, TipoValue v) {
        return null;
    }
    public TipoValue getValor() {
        return valor;
    }
    public Booleano getTerminado() {
        return terminado;
    }
    public void setTerminado(boolean b) {
        terminado.setB(false);
    }


}
```

Figura 2.10: Ejemplo código Futuro

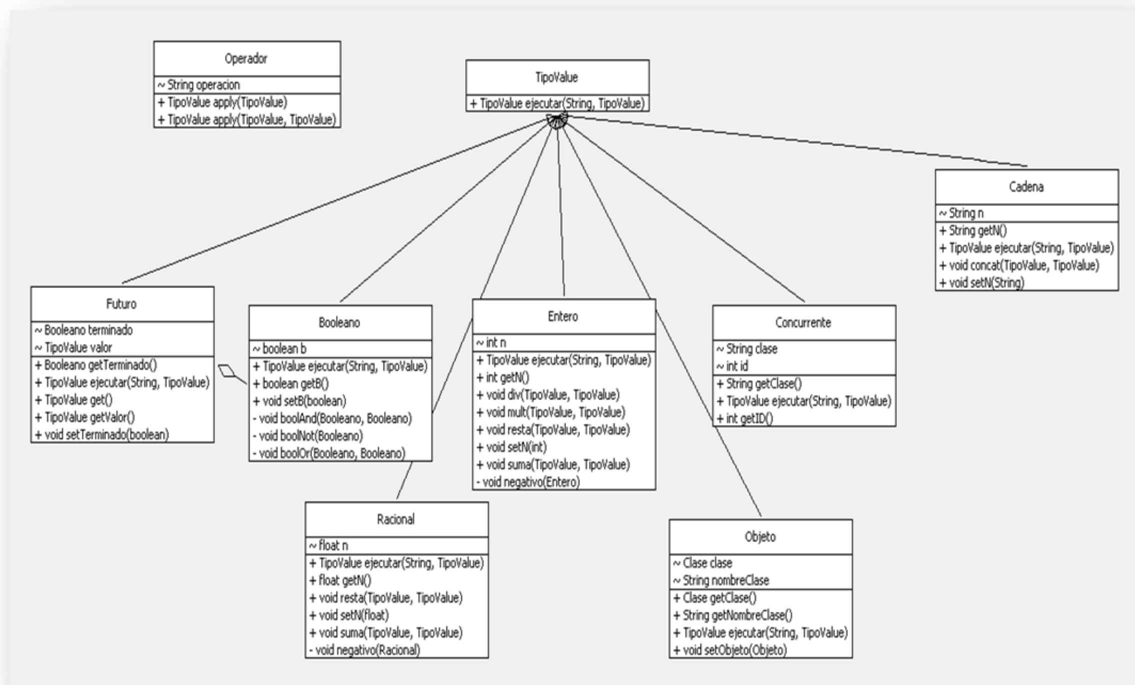

Figura 2.11: Diagrama de clases del paquete TipoValue

3.1.5. Expresiones

El último paquete de la implementación corresponde al de **Expresiones**. Al igual que en **Comandos** y **TipoValue** existe una clase padre **Expr** de la que heredan el resto de clases. Las expresiones implementan el método *eval* en vez del *ejecutar* de los otros paquetes ya que se evalúa la expresión y se devuelve un objeto de tipo **TipoValue** tras la evaluación. De todas ellas (**AsyncCall, BinOp, ConstBool, ConstInt, ConstObjConcurrente, ConstObjeto, ConstString, ExprAwait, ExprGet, SyncCall, UniOp, Var**) cabe destacar los siguientes:

- **AsyncCall**: se encarga de recoger la información de una llamada asíncrona. Tras evaluarse la expresión asíncrona se devolverá un objeto de tipo **Futuro** por ser una llamada asíncrona.
- **BinOp**: se encarga de evaluar las expresiones de operaciones binarias para devolver una expresión que se asignará a una variable.
- **ConstBool, ConstInt, ConstString**: evalúan las expresiones de tipo **Booleano**, **Entero** y **Cadena**.
- **ConstObjConcurrente**: evalúa la expresión "*new cog*" para asignar a una variable un objeto de tipo **ObjetoConcurrente**.
- **ExprGet**: es el encargado de determinar si un objeto de tipo **Futuro** ha conseguido ya su valor o no. Si no tiene valor el objeto, **ExprGet** mandará una petición de parada al objeto llamante para que ceda el control de flujo a otro objeto concurrente.

```
public TipoValue eval(Contexto context) {
    context.cuentaAsynCallObj();
    context.cuentaAsynCallMetodo();
    Clase c = null;
    List<Expr> parametrosTmp = new ArrayList<Expr>();
    int n = parametros.size();
    for (int i = 0; i < n; i++){
        Expr e = parametros.get(i);
        TipoValue t = e.eval(context);
        Expr eTmp;
        if (t.getClass().toString().contains("Entero")){
            eTmp = new ConstInt(((Entero)t).getN());
        }else if (t.getClass().toString().contains("Booleano")){
            eTmp = new ConstBool(((Booleano)t).getB());
        }else {
            eTmp = new ConstObjeto(((Objeto)t).getNombreClase());
        }
        parametrosTmp.add(eTmp);
    }

    Metodo m = null;
    //SI NO SABEMOS LA CLASE DEL OBJETO QUE HACE LA LLAMADA SYNCRONA LA BUSCAMOS EN LA TABLA DE ATRIBUTOS DEL METODO
    if (clase.equals("")){
        Expr e = new Var(id);
        TipoValue ob = e.eval(context);
        //SI EL TIPOVALUE ES OBJETO (NO ESTA CREADO CON NEW COG)
        if (ob.getClass().toString().contains("Objeto")){

            //FALLO!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            Objeto objeto = (Objeto) ob;
            c = context.getTablaClases().get(objeto.getClase());

        }else {
        //SI EL TIPOVALUE ES OBJETO CONCURRENTE (ESTA CREADO CON NEW COG)
            Concurrente objeto = (Concurrente) ob;
            c = context.getTablaClases().get(objeto.getClase());
            if (context.getTablaObjInactivos().containsKey(objeto.getID())){
                context.activarObjConcurrente(objeto.getID());
            }
            ObjetoConcurrente objCon = context.getTablaObj().get(objeto.getID());
            m = new Metodo(c.getMetodo(metodo));
            Tarea tAsyncCall = new Tarea(m.getNombre(), parametrosTmp, m);
            objCon.anadirTarea(tAsyncCall);
        }
    }else{
        c = context.getTablaClases().get(clase);
        m = new Metodo(c.getMetodo(metodo));
        Tarea tAsyncCall = new Tarea(m.getNombre(), parametrosTmp, m);
        ObjetoConcurrente objCon = context.getObjConcurrenteActivo();
        objCon.anadirTarea(tAsyncCall);
    }

    return new Futuro(m.getRetorno(), m.getTerminado());
}
```

Figura 2.12: Ejemplo código AsyncCall

```
public class ConstObjConcurrente extends Expr{
    String clase;
    String nombre;

    public ConstObjConcurrente(String clase) {
        this.clase = clase;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String name) {
        this.nombre = name;
    }

    public TipoValue eval(Contexto context){
        context.cuentaNewCogs();
        Map<String,Clase> tablaClases = context.getTablaClases();
        if (tablaClases.containsKey(clase)) {
            Clase c = tablaClases.get(clase);
            Map<String,TipoValue> campos = c.getTablaAtr();
            int id = context.getID();
            ObjetoConcurrente oC = new ObjetoConcurrente(id, clase, nombre, campos);
            context.getTablaObj().put(id,oC);
            context.getListaObjEjecucion().add(oC);
            context.avanceID();
            return new Concurrente(id,clase);
        }
        return null;
    }

}
```

Figura 2.13: Ejemplo código ConstObjConcurrente

```
public class ConstInt extends Expr{
    int n;
    public TipoValue eval(Contexto context){
        return new Entero(n);
    }

    public ConstInt(int n) {
        this.n = n;
    }
}
```

Figura 2.14: Ejemplo código ConstInt

```java
public class ExprGet extends Expr{
    String name;

    public ExprGet(String n) {
        name = n;
    }

    public TipoValue eval(Contexto context) {
        Futuro f = null;
        if (context.getCamposMetodo().containsKey(name)) {
            f = (Futuro) context.getCamposMetodo().get(name);
        } else if (context.getCamposClase().containsKey(name)) {
            f = (Futuro) context.getCamposClase().get(name);
        }
        if (f.getTerminado().getB()) {
            System.out.println("EXPRGET:: " + ((Entero)f.getValor()).getN());
            return f.getValor();
        } else {
            context.pararPorGet();
            context.setPararGet(true);
        }
        return null;
    }
}
```

Figura 2.15: Ejemplo código ExprGet

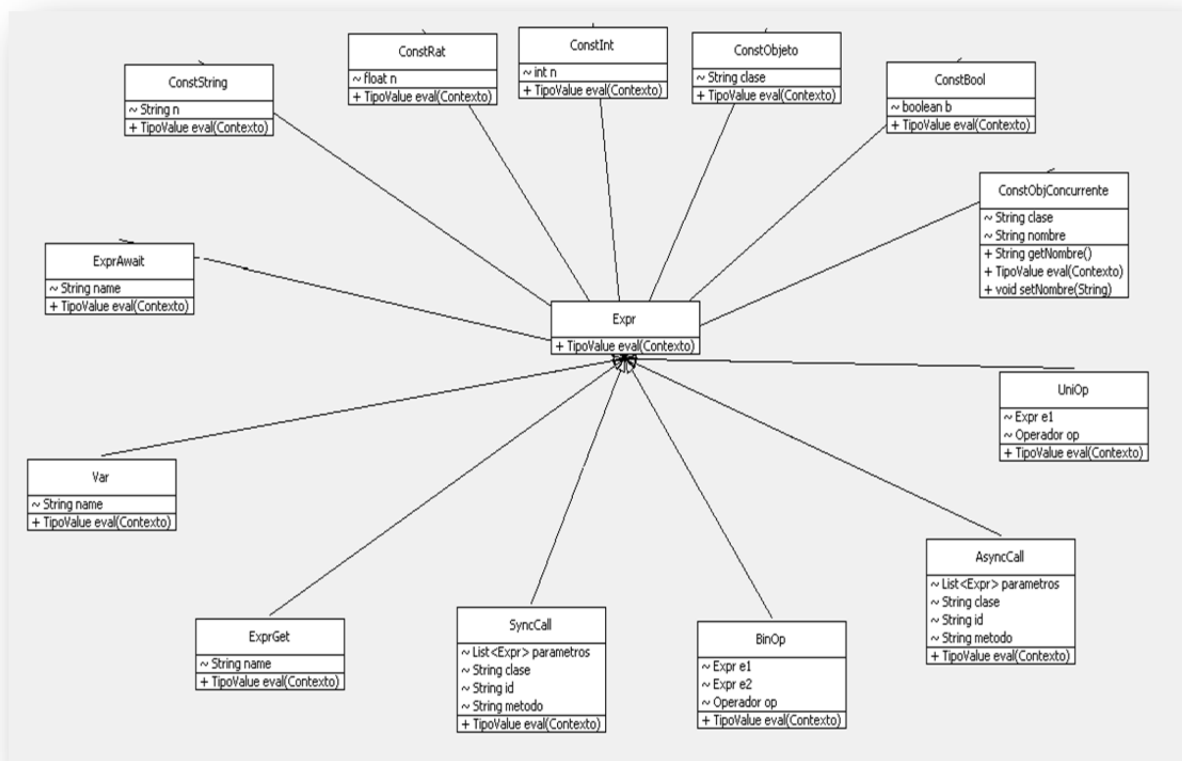

Figura 2.16: Diagrama de clases del paquete Expresiones

4. Resultados obtenidos

Para probar el funcionamiento de ProsymbABS realizamos una prueba para el algoritmo de Fibonacci en el que realizamos llamadas asíncronas por cada llamada recursiva que realiza el programa. De este modo repartimos las tareas entre varios procesadores.

Cargando el programa ABS en la interfaz de ProsymbABS podemos ver lo siguiente:

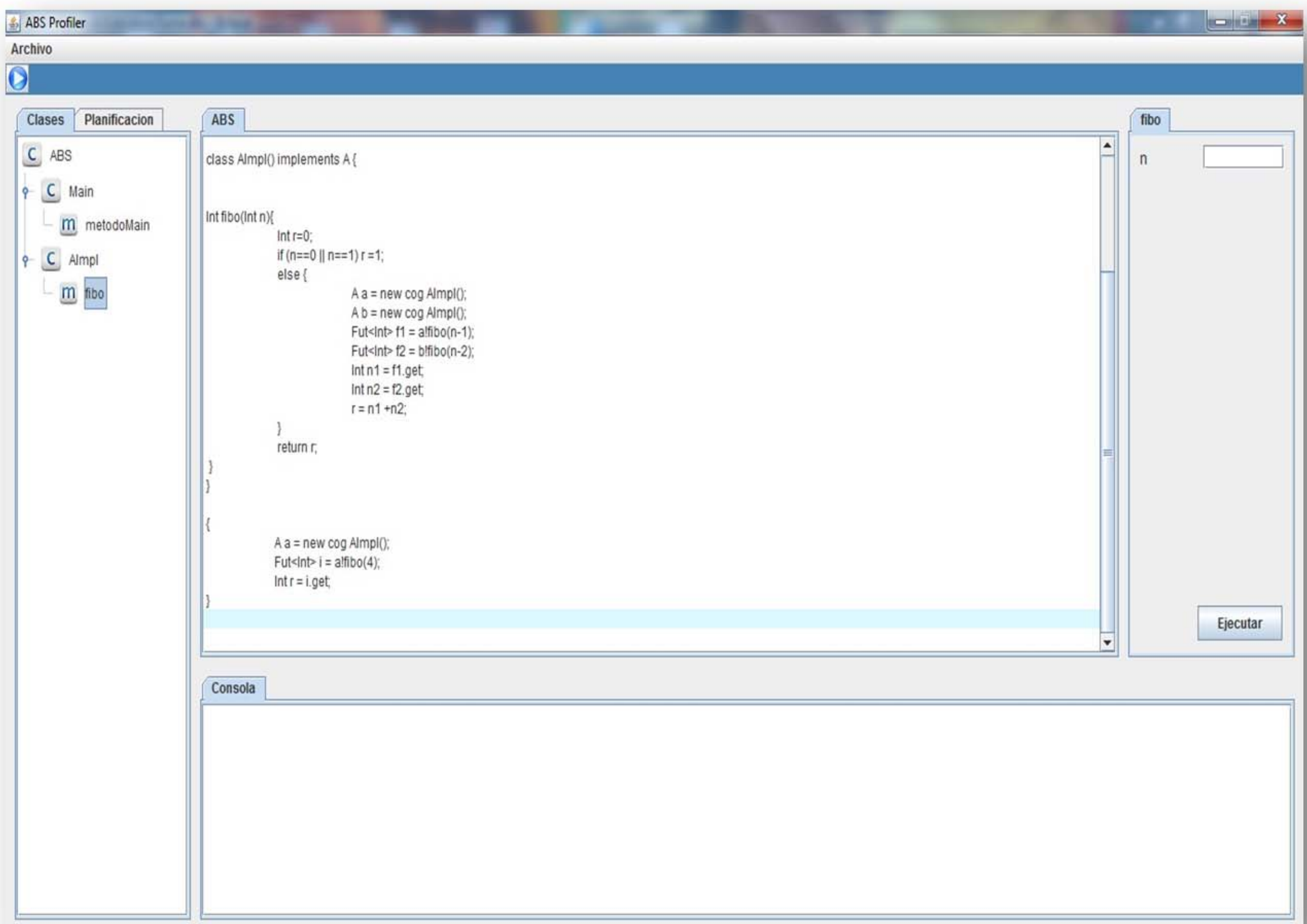

Figura 3.1: Programa de fibonacci ABS

Como ABS no tiene ningún perfilador desarrollado por el momento, probamos la ejecución del programa en ABS puro que nos proporciona el *plugin* de ABS Tools. Este *plugin* cuenta con la posibilidad de ejecutar el código dibujando un diagrama de secuencias, con lo que se puede ver cómo ha funcionado la ejecución.

Figura 3.2: Diagrama secuencial de ejecución ABS

El programa ABS realiza una ejecución del algoritmo de Fibonacci para el número cuatro. Debido a la recurrencia y a que, como hemos explicado antes, cada llamada recursiva corresponde a una llamada asíncrona, se crean un total de nueve objetos concurrentes de tipo **AImpl**, hecho que se puede ver en la figura anterior.

Una vez cargado nuestro código en el panel central, automáticamente ProsymbABS se encargará de llamar al *parser* de ABS para que compile el código y, si no hay errores, devolverá el árbol sintáctico resultante. Una vez compilado el código se realiza el parseado del árbol sintáctico para guardar toda la información en la variable de **Contexto** de la aplicación como las diferentes clases definidas, la lista de comandos de cada método, preparar la lista de ejecución de los objetos concurrentes (en este caso al principio sólo contamos con el objeto concurrente "Main"). Con ello ya tenemos todo listo para iniciar la ejecución del programa y realizar el análisis de consumo de recursos sobre el mismo.

La ejecución del código se puede realizar de dos maneras. La primera consiste en realizar la ejecución completa del programa, es decir, partiremos desde el bloque principal del programa ABS y obtendremos la siguiente salida por consola:

```
Consola
Numero de cogs creados para la clase Main : 1
Numero de cogs creados para la clase AImpl : 9
1 llamada al metodo metodoMain desde el objeto 'Main-metodoMain' con identificador '0' de la clase 'Main'
                    Numero instrucciones ejecutadas: 7
                    Numero de llamadas asincronas realizadas: 1
1 llamada al metodo fibo desde el objeto 'a' con identificador '1' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 13
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'a' con identificador '2' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 12
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'b' con identificador '3' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 11
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'a' con identificador '4' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 11
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'b' con identificador '5' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'a' con identificador '6' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'b' con identificador '7' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'a' con identificador '8' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'b' con identificador '9' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
Instrucciones totales: 74
New Cogs totales: 10
New Objetos totales: 0
33 ms
FIN
```

Figura 3.3: Resultado ejecución total en ProsymbABS

ProsymbABS muestra la información simbólica recogida de la ejecución del programa ABS. Debido a que ejecutamos el programa ABS desde el método principal, creamos una clase "Main" donde empezará a ejecutarse el código del programa ABS. Desde esta clase se realiza una llamada asíncrona al algoritmo de fibonacci mediante la creación de un *cog* nuevo "*a*" previamente. Al tratarse de una llamada asíncrona, se crea una variable futura para reservar el valor del resultado que devolverá la llamada a fibonacci. Dentro de su *cog* el objeto "*a*" ejecutará el código del algoritmo de fibonacci. Como el parámetro que recibe es mayor que uno (cuatro), toma el camino por la parte recursiva del algoritmo y realiza dos llamadas asíncronas para que continúen la ejecución.

Por lo tanto se seguirán creando nuevos *cogs* hasta que el valor del parámetro sea 0 o 1, por lo que habremos llegado al caso base del algoritmo y la llamada retornará un valor de resultado. En el algoritmo de fibonacci crea ocho *cogs* nuevos para esta ejecución, más la llamada del bloque principal se crean un total de nueve *cogs* que coincide con el número de *cogs* creados en la ejecución realizada anteriormente únicamente del código ABS.

Cada cog que se crea se le asigna un identificador para diferenciar los distintos cogs, ya que se puede dar el caso de que se creen cogs con el mismo nombre. Mostramos, además, las diferentes llamadas a métodos provenientes de los cogs creados. De cada llamada a método se muestra información referente al número de instrucciones ejecutadas en cada método y el número de llamadas asíncronas que se realizan.

Aparte de mostrar los datos correspondientes a cada método ejecutado en el programa, se muestra, además, la cuenta total del número de instrucciones que se han ejecutado en el programa ABS (asignaciones, creación de objetos, operaciones binarias, llamadas asíncronas, operaciones de retorno). En total, como se puede observar en la imagen anterior, se ejecutan 74 instrucciones (comandos) en la aplicación.

Por último calculamos el tiempo que ha tardado ProsymbABS en realizar esta ejecución, para ello marcamos con una variable el punto de inicio y el punto final, dentro del código de nuestra aplicación, con la que se calculará el tiempo transcurrido entre ambos puntos. Comenzamos a contar justo antes de lanzar la ejecución del programa ABS y paramos de contar cuando termina de ejecutarse el último objeto concurrente que queda en la lista de objetos concurrentes.

El segundo tipo de ejecución con el que se puede trabajar en ProsymbABS es el de ejecutar únicamente el método deseado, para ello se selecciona el método en el diagrama de la izquierda. Con ello se actualiza el panel derecho con los parámetros que tiene el método para definirles su valor inicial:

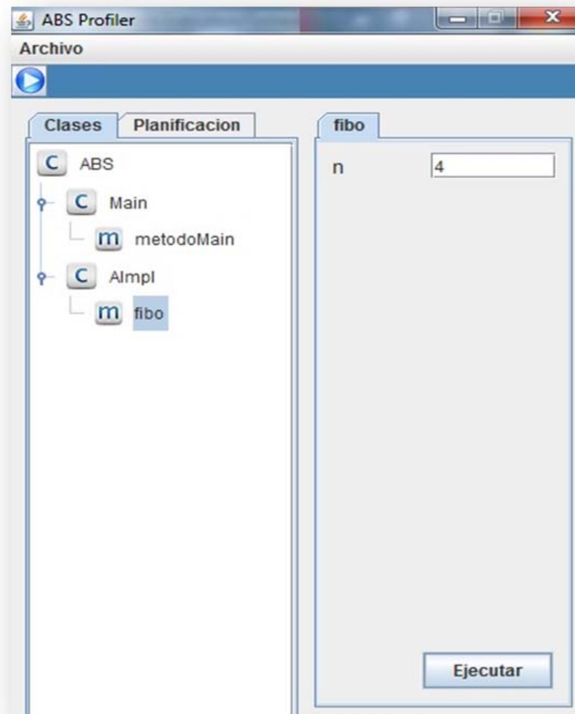

Figura 3.4: Ejecución parcial en ProsymbABS

Una vez rellenados los campos, al hacer click en el botón Ejecutar obtendremos el siguiente resultado por pantalla:

```
Consola
Numero de cogs creados para la clase Main : 0
Numero de cogs creados para la clase AImpl : 9
1 llamada al metodo fibo desde el objeto 'AImpl-fibo' con identificador '0' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 13
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'a' con identificador '1' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 12
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'b' con identificador '2' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 11
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'a' con identificador '3' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 11
                    Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'b' con identificador '4' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'a' con identificador '5' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'b' con identificador '6' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'a' con identificador '7' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'b' con identificador '8' de la clase 'AImpl'
                    Numero instrucciones ejecutadas: 4
                    Numero de llamadas asincronas realizadas: 0
Instrucciones totales: 67
New Cogs totales: 9
New Objetos totales: 0
29 ms
FIN
```

Figura 3.5: Resultado ejecución parcial en ProsymbABS

Se observan unos pequeños cambios con respecto a la ejecución anterior. En este caso la ejecución no comienza desde el bloque principal por lo que no hay creado un objeto concurrente “Main”. En vez de ello, se crea directamente como primer objeto concurrente el de la clase que implementa el método elegido.

Por lo tanto se crean un total de ocho *cogs* que corresponde a la ejecución del método de fibonacci, y no nueve como ocurre en la ejecución anterior debido a que no se ha creado ese objeto concurrente “Main”.

Por la misma razón, el tiempo de ejecución y el número de instrucciones ejecutadas ha bajado levemente con respecto a la ejecución anterior. El bloque principal ejecuta un total de siete instrucciones las cuales son:

- Creación del objeto.
- Asignación.
- Llamada asíncrona.
- Asignación.
- Método get (variable de terminado evaluada a falso).
- Método get (variable de terminado evaluada a cierto).
- Asignación.

Se ejecutan, por lo tanto 67 instrucciones frente a las 74 de la ejecución anterior.

5. Manual del usuario

A continuación presentamos el manual de la aplicación. Como requisito previo a la ejecución es necesario tener instalado la última versión de Java en el ordenador para que la aplicación funcione correctamente con todos los *plugins* con los que se ha implementado.

5.1. Inicio

Tras iniciar la aplicación nos aparecerá la siguiente pantalla, correspondiente a la interfaz de la aplicación:

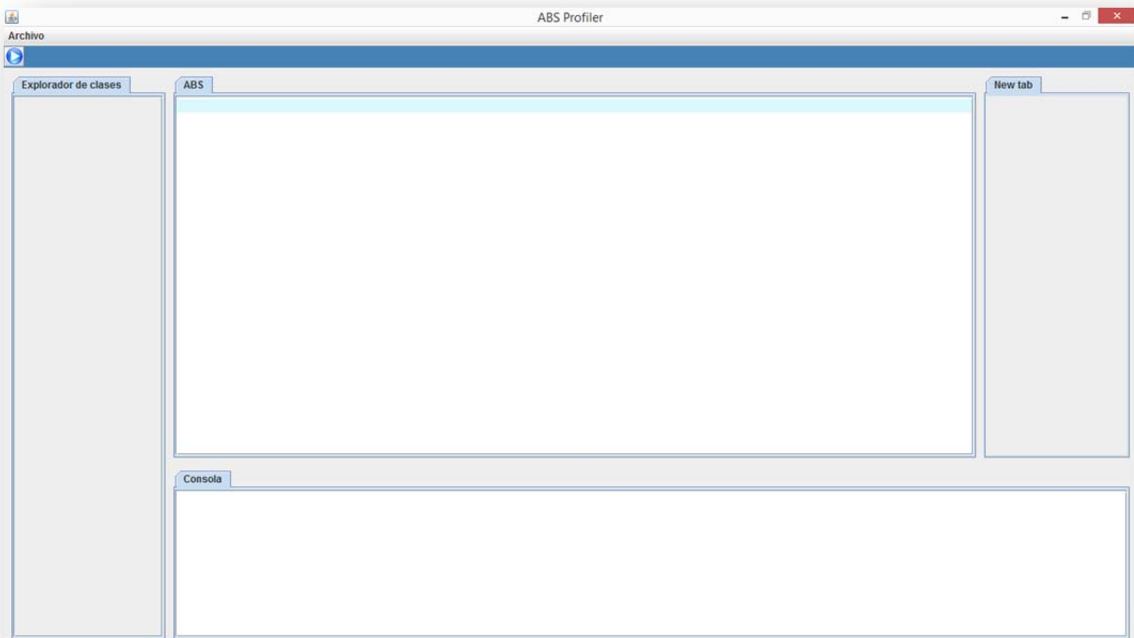

Figura 4.1: Interfaz de la aplicación

La interfaz consta de varios paneles con distintas funcionalidades. El panel izquierdo mostrará un diagrama de clases del programa ABS que se ha compilado y las distintas planificaciones que se le asignan a cada clase. El panel central contiene el código ABS del cual se realizará la compilación y posterior ejecución del mismo. El panel derecho nos permite ejecutar un método concreto pasándole los valores a sus parámetros de entrada. Por último, el panel inferior corresponde a la consola de la aplicación donde se muestra los resultados obtenidos por el perfilador (métrica, número de objetos creados, número de llamadas asíncronas, tiempo de ejecución).

5.2. Edición

Podemos empezar a editar nuestro código directamente en el área de texto del panel central, o bien, podemos abrir un archivo en formato ABS. Para elloo le damos a Archivo (barra superior) y luego a Abrir.

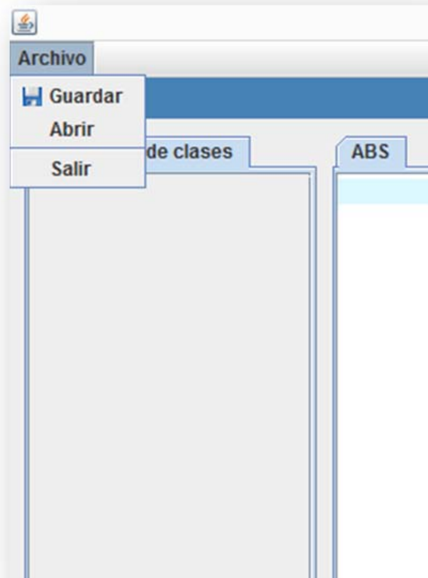

Figura 4.2: Pestaña Archivo

Una vez le hayamos dado a abrir aparecerá un explorador de archivos de nuestro ordenador desde donde podemos seleccionar uno para abrir.

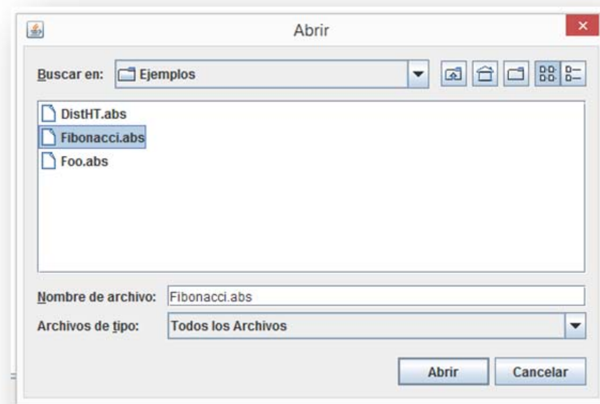

Figura 4.3: Selección de archivo ABS

Después de abrir tendremos el código ABS en el panel central, desde donde podemos editar o ejecutar la aplicación.

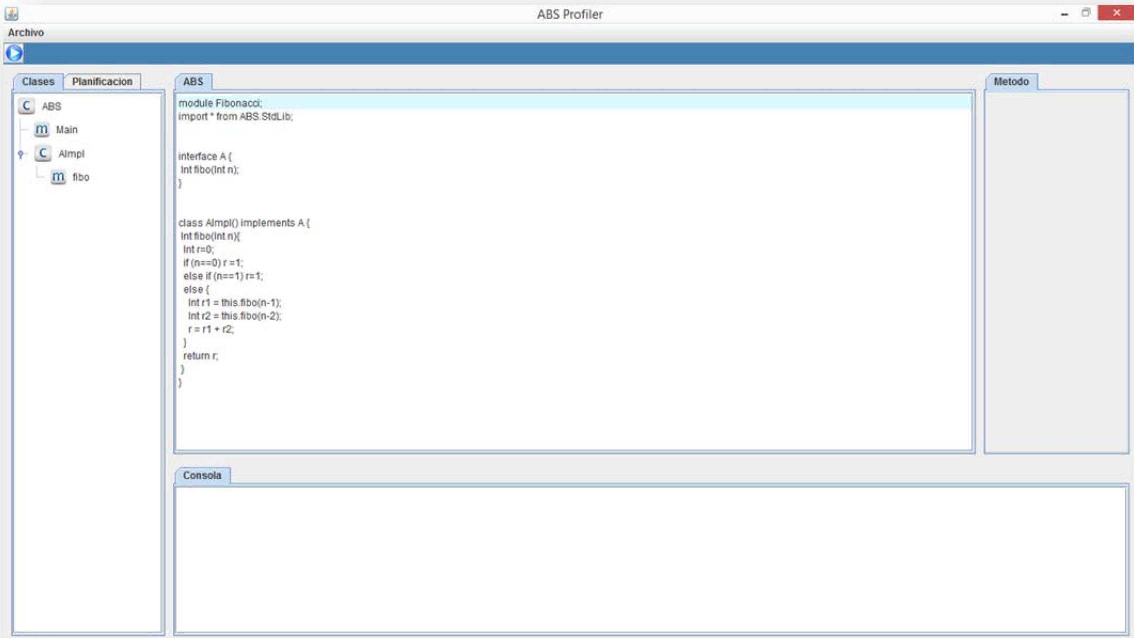

Figura 4.4: Interfaz con código compilado

5.3. Errores

El código del panel central se podrá ir editando y, al igual que Java, el compilador seguirá funcionando con cada cambio que registre el código. Podremos ver los errores que aparecerán marcados con una línea roja facilitándonos su depuración. Una vez corregido el error la línea roja desaparecerá.

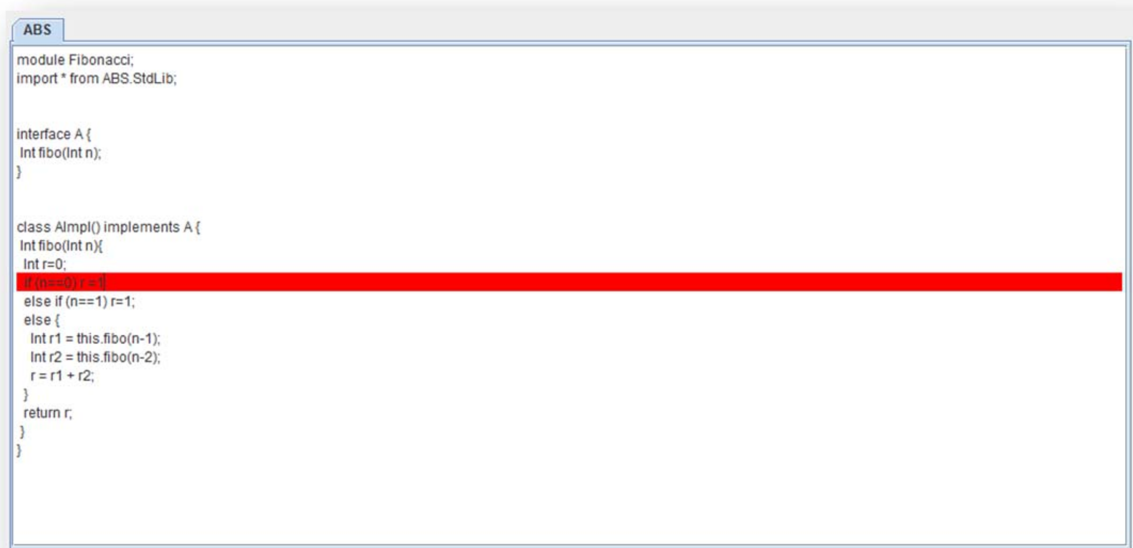

Figura 4.5: Error en el código ABS

5.4. Planificación

En la pestaña planificación (en el panel izquierdo), es donde podremos elegir las planificaciones que queremos para nuestras clases:

- **Azar**: se selecciona una Tarea de la lista de Tareas
- **FIFO**: se seleccionan las Tareas por orden de llegada
- **LIFO**: se seleccionan primero la tarea que se ha añadido recientemente
- **Round Robin**: ejecuta *n* Tareas elegidas al azar y pasa al siguiente Objeto Concurente.
- **Prioridades**: Para este caso tendremos que añadir prioridades a los métodos de las clases, para así saber que tarea será la siguiente.

Estas planificaciones serán las encargadas de elegir el orden de ejecución de las tareas en los distintos Objetos Concurrentes.

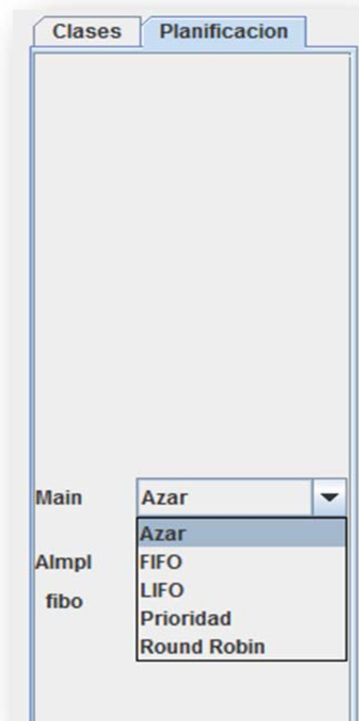

Figura 4.6: Panel de planificaciones

5.5. Ejecución

Una vez que nuestro programa está compilado correctamente, libre de errores y tras haber elegido una planificación para cada clase podremos ejecutar nuestro código de las siguientes maneras:

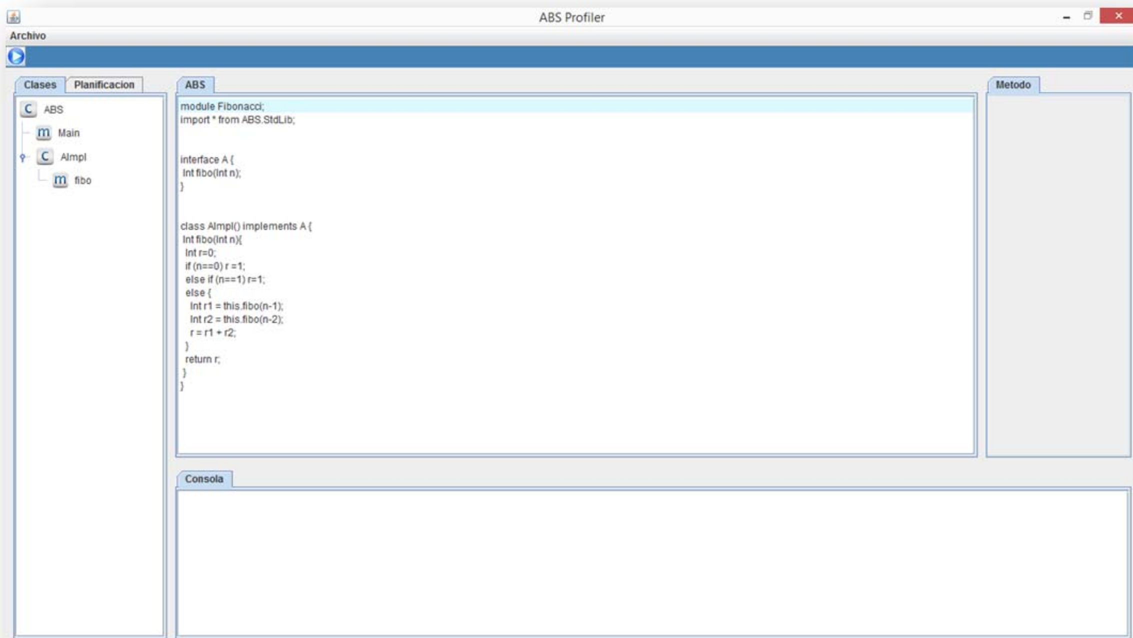

Figura 4.7: Interfaz de la aplicación con código

5.5.1. Ejecución total del programa ABS

Esta opción es la encargada de ejecutar nuestro código ABS de manera convencional, esto quiero decir que empecerá la ejecución desde el bloque principal del programa ABS, creando un proceso principal. Para realizar esta ejecución tendremos que darle al símbolo de *play* que aparece en la barra superior.

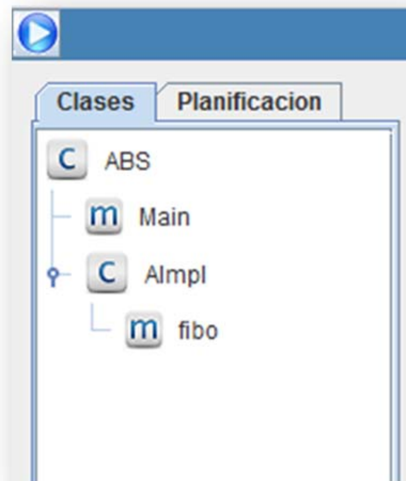

Figura 4.8: Botón *play*

5.5.2. Ejecución parcial del código ABS

Otra posibilidad para ejecutar ProsymbABS es seleccionando un solo método pasándole los valores de los parámetros de entrada del método seleccionado.

Desde el panel izquierdo podremos explorar las clases y los métodos que tenemos en nuestro código. Este panel se actualizará a la vez que vamos editamos nuestro código. Desde este panel podemos seleccionar el método que queremos ejecutar.

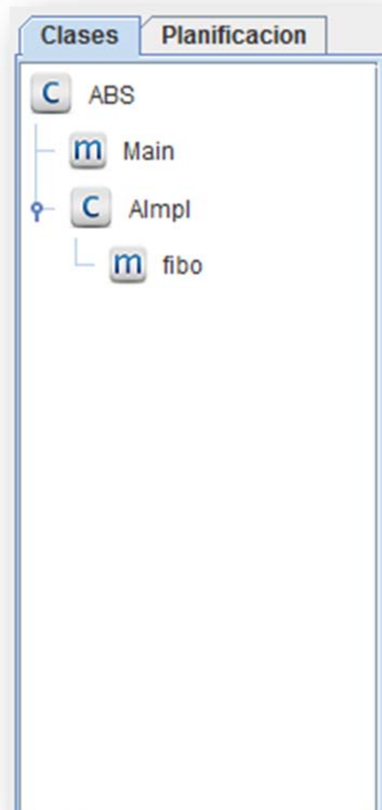

Figura 4.9: Panel de clases

Una vez seleccionado el método se refrescará el panel que tenemos a la derecha, desde donde podemos inicializar los parámetros de entrada. Si no recordamos el tipo de los atributo, podemos saberlo poniendo el cursor encima del nombre del atributo, donde aparecerá un *tooltip* con el tipo del parámetro seleccionado. Una vez inicializados los parámetros, solo tenemos que pulsar el botón Ejecutar.

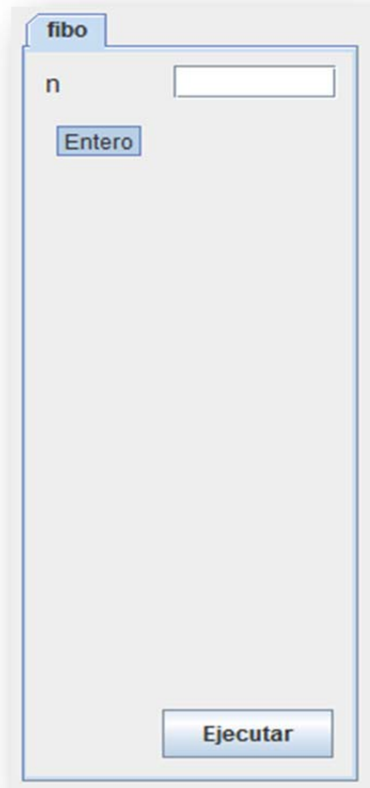

Figura 4.10: Panel de método

5.6. Resultados

Una vez ejecutado el código (tanto total como parcialmente), se podrá observar los resultados que devuelve nuestro perfilador sobre la ejecución del programa ABS en el panel inferior de la interfaz, correspondiente a la consola.

```
Consola
Numero de cogs creados para la clase Main : 1
Numero de cogs creados para la clase AImpl : 9
1 llamada al metodo metodoMain desde el objeto 'Main-metodoMain' con identificador '0' de la clase 'Main'
                Numero instrucciones ejecutadas: 7
                Numero de llamadas asincronas realizadas: 1
1 llamada al metodo fibo desde el objeto 'a' con identificador '1' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 13
                Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'a' con identificador '2' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 12
                Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'b' con identificador '3' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 11
                Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'a' con identificador '4' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 11
                Numero de llamadas asincronas realizadas: 2
1 llamada al metodo fibo desde el objeto 'b' con identificador '5' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 4
                Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'a' con identificador '6' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 4
                Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'b' con identificador '7' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 4
                Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'a' con identificador '8' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 4
                Numero de llamadas asincronas realizadas: 0
1 llamada al metodo fibo desde el objeto 'b' con identificador '9' de la clase 'AImpl'
                Numero instrucciones ejecutadas: 4
                Numero de llamadas asincronas realizadas: 0
Instrucciones totales: 74
New Cogs totales: 10
New Objetos totales: 0
33 ms
FIN
```

Figura 4.11: Consola de la interfaz

6. Conclusiones

Hemos conseguido ejecutar el ABS de forma secuencial, de modo que siempre se ejecuta un solo Objeto Concurrente o proceso a la vez, lo que nos hecho más fácil contabilizar los datos del perfilador, ya que no tenemos que tener en cuenta la concurrencia.

Cuando un objeto concurrente ejecuta todas las tareas que tiene pendiente, pasa a ejecutar otro Objeto Concurrente, la selección del siguiente Objeto Concurrente en nuestro caso es al azar. El objeto concurrente que finaliza sus tareas pasa a la lista de Objetos Concurrente inactivos, a la espera de que otro objeto concurrente le añada una nueva tarea con una llamada asíncrona.

Las tareas que tiene un objeto concurrente se ejecutan según la planificación que hayamos elegido, la cual puede ser una de las siguientes:

- **Azar**: se selecciona una Tarea de la lista de Tareas
- **FIFO**: se seleccionan las Tareas por orden de llegada
- **LIFO**: se seleccionan primero la tarea que se ha añadido recientemente
- **Round Robin**: ejecuta *n* Tareas elegidas al azar y pasa al siguiente Objeto Concurente.
- **Prioridades**: Para este caso tendremos que añadir prioridades a los métodos de las clases, para así saber qué tarea será la siguiente.

Aunque esta planificación no es parte del lenguaje ABS, desarrollamos esta parte como una ayuda para la depuración de programas ABS ya que vimos como prioritario la comparación de la ejecución de un mismo código con distintas planificaciones para entender de mejor manera su comportamiento.

El método *get* en ABS puro hace una espera activa en el Proceso del Objeto Concurrente mientras la variable futura no tenga su valor, pero en nuestra aplicación no podemos hacer una espera ya que sólo utilizamos un proceso (las instrucciones *get*). Por eso nuestro intérprete puede contabilizar más de una vez esta instrucción ya que, al volver a ejecutar el Objeto Concurrente donde está declarado, tendrá que volver a comprobar si el método *get* ya devuelve el valor de la variable futura.

El método *await*, al igual que el método *get*, puede ser contabilizado más de una vez, aunque en este caso la espera solo le corresponde a la Tarea, ya que el objeto Concurrente pasa a ejecutar otra tarea.

Cuando ejecutamos un solo método se crea un objeto concurrente principal que lo nombramos de la siguiente manera: 'nombre de la clase' - 'nombre del método' (p.e. AImpl - fibo) y se le pasará los parámetros que hayamos especificado.

En ejecuciones muy largas y sobre todo donde se crean muchos Objetos Concurrentes, hemos pensado que no es necesario escribir todos los datos del número de llamadas asíncronas de cada uno, ya que será imposible su visualización en el panel de la Consola, por lo que limitamos el número de objetos concurrentes que se visualizarán. No obstante, estos objetos seguirán contando para los datos generales de la ejecución (número de instrucciones, número de objetos concurrentes, número de llamadas asíncronas).

6.1. Trabajos futuros

Mucha parte de nuestro código está comentada de manera que pueda ser más sencillo su entendimiento. También, como se indica en la sección de implementación, tenemos nuestras clases divididas en paquetes para ayudar a facilitar la estructuración del proyecto y que sea fácil el acceso a las distintas clases.

Creemos que nuestro intérprete de ABS podría tener un *plugin* para eclipse, para no tener que estar ejecutando cada vez la aplicación, pero manteniendo muchas de las características que tiene actualmente: errores que se van indicando mientras editas el código, ejecución de un solo método. También se podría añadir coloreado especial de palabras claves de ABS para mejorar la visualización del código

Otra cosa pendiente es la implementación de la parte funcional de ABS. Para dicha implementación se podría empezar diseñando las clases que se necesitarán, ya que no debería de interferir con el diseño de las clases que usamos actualmente. Ademas, habría que añadir métodos en clase **MainClass** para que reciban la información del *parser* de ABS y la interpreten a estas nuevas clases. Al igual que con la parte secuencial, se puede calcular el consumo de recursos de la parte funcional de ABS.

Otra opción que se podría añadir sería opción gráfica donde se podría explicar con gráficos la recopilación de datos que realiza el perfilador, para facilitar la interpretación de los mismos.

Como la ejecución de los Objetos concurrentes se hace de manera secuencial, se podría añadir que cada Objeto Concurrente sea un *thread* distinto, con esto ahorraríamos bastante tiempo en la ejecución, sobre todo cuando hay muchos objetos concurrentes creados. La concurrencia entre los objetos implicaría que la recopilación de la información como numero de instrucciones totales, número de objetos concurrentes creados o número de llamadas asíncronas tenga que hacerse de manera atómica para proteger su acceso mientras estamos contabilizándolas.

7. Referencias y bibliografía

[1] JCoBox, disponible en https://softech.informatik.uni-kl.de/Homepage/JCoBox

[2] E. Broch Johnsen, O. Owe, I. Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems, http://www.sciencedirect.com/science/article/pii/S0304397506004804

[3] JProfiler, disponible en https://www.ej-technologies.com/products/jprofiler/features.html

[4] JBoss, disponible en http://www.jboss.org/technology/

[5] Página de herramientas ABS del proyecto HATS, http://tools.hats-project.eu/

[6] Manual de la sintaxis de ABS, http://tools.hats-project.eu/download/absrefmanual.pdf

Profiling:

- http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/Analisis%20Comparativo%20del%20Rendimiento.pdf
- http://es.wikipedia.org/wiki/An%C3%A1lisis_de_rendimiento_de_software
- http://es.wikipedia.org/wiki/An%C3%A1lisis_din%C3%A1mico_de_software
- http://es.wikipedia.org/wiki/An%C3%A1lisis_est%C3%A1tico_de_software

Concurrencia:

- http://cala.unex.es/cala/epistemowikia/index.php?title=Verificaci%C3%B3n_de_Programas_Concurrentes_(versi%C3%B3n_1)/Introducci%C3%B3n_a_la_Concurrencia

- http://www.teknoplof.com/2013/10/10/la-programacion-concurrente-es-el-futuro-inmediato-o-deberia-serlo/

Lenguaje ABS:

- http://tools.hats-project.eu/
- http://einarj.at.ifi.uio.no/Papers/johnsen10fmco.pdf

8. Apéndice

En este apartado hemos querido adjuntar un artículo que explica la semántica de ABS y que nos ha sido bastante útil para entender el funcionamiento de la ejecución de ABS y, por tanto, ha facilitado la realización del proyecto.

2.3 Operational Semantics

An object is of the form $\mathtt{ob}(o, C, h, \langle tv, s\rangle, \mathcal{Q})$, where o is the *object identifier*, C is its class name, h is its local *heap*, $\langle tv, s\rangle$ is the *execution context* of the current task, being tv the *table of local variables* and s the sequence of instructions to be executed by the current task, and $\mathcal{Q}$ is the set of pending tasks, being each of them an execution context. In the following we use ϵ to denote either an empty sequence of instructions or an empty execution context. A heap h maps *field names* (declared in C) to $\mathbb{V} = \mathbb{Z} \cup \{\mathsf{null}\} \cup \mathit{Objects}$, where *Objects* denotes the set of object identifiers. A table of variables tv maps local variables to $\mathbb{V}$. It contains the special entry $\mathtt{r}$ to associate the return variable of a method to the corresponding future variable. Future events have the form $\mathtt{fut}(\mathtt{fn}, v)$ where $v \in \mathbb{V} \cup \bot$. The symbol $\bot$ indicates that $\mathtt{fn}$ does not have a value yet. For simplicity, we assume that all methods return a value. An execution state (or *configuration* $\mathcal{S}$) has the form $\{\!\!\{a_1, \ldots, a_n\}\!\!\}$, where a_i can be either an *object* or a *future* event. Execution states are in fact represented as multisets of objects and future events. In the following, we use the notation $\{\!\!\{a|\mathcal{S}\}\!\!\}$ to stand for the new multiset $\{\!\!\{a\}\!\!\} \cup \mathcal{S}$.

The *operational semantics* is given in a rewriting-based style, where, at each step, a subset of the state is rewritten according to the rules in Fig. 3. Let us intuitively explain the semantics. Function $eval_e$ evaluates an expression e with respect to a heap h and a table of variables in the standard way. Note that the heap is required to evaluate expressions of the form $this.f$, what returns (rule 1) $h(f)$ as result. Function $eval_{guard}(guard, tv)$ in rule 4 returns *true* iff $guard \equiv true$ or $guard \equiv x_1\ op_R\ x_2$ and $tv(x_1)\ op_R\ tv(x_2)$. Finally the evaluations of conditions in `await` instructions is done by function $eval_g$. In particular, for the case of future variables $x?$ in rules 9 and 10, this function behaves as follows: $eval_g(x?, tv, h, \mathcal{S}) = true$ iff $tv(x) = \mathtt{fn}$ and $\mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}$, $v \neq \bot$. The notation $tv[x \mapsto v]$ (resp. $h[f \mapsto v]$) stands for storing v in the local variable x (resp. field f).

Rules 1 and 2 operate in the expected way. In rule 3, it can be observed that the table of variables tv maps x to o_1. Function $newRef()$ is in charge of generating fresh object identifiers.

In rule 4, a call to a block is resolved by finding a matching rule and adding its body to the sequence of instructions to be executed. The notation $r \equiv p(this', \bar{x}', y') \leftarrow b_1, \ldots, b_n \ll P$ stands for a fresh renaming of a rule in P. $newenv(r)$ creates a new mapping for variables in r, where each variable is initialized to either 0 or null. When the execution of a block finishes (rule 6) the state is prepared to later apply rule 11 to select a new task from the queue.

Rule 5 deals with asynchronous method invocations. When an object o_1 calls to a method $p(\bar{x})$, the information required to execute the call is stored in the queue of the object represented by o_1. Observe that tv_2 has the special entry $\mathtt{r}$ to store the relation between the future variable $\mathtt{fn}$ where storing the result and the output parameter y'. This future variable is initially undefined, thus $\mathtt{fut}(\mathtt{fn}, \bot)$ is added to the state. When the method returns a value (rule

$$(1)\ \frac{v = eval_e(e, h, tv), x \in dom(tv)}{\{\mathtt{ob}(o, C, h, \langle tv, x := e \cdot s\rangle, \mathcal{Q})|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, \langle tv[x \mapsto v], s\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(2)\ \frac{v = tv(y)}{\{\mathtt{ob}(o, C, h, \langle tv, this.f := y \cdot s\rangle, \mathcal{Q})|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h[f \mapsto v], \langle tv, s\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(3)\ \frac{o_1 = newRef(),\ h_1 \text{ and } tv_1 \text{ are empty mappings}}{\begin{array}{c}\{\mathtt{ob}(o, C, h, \langle tv, x := \mathsf{new}\ D \cdot s\rangle, \mathcal{Q})|\mathcal{S}\} \leadsto \\ \{\mathtt{ob}(o, C, h, \langle tv[x \mapsto o_1], s\rangle, \mathcal{Q}), \mathtt{ob}(o_1, D, h_1, \langle tv_1, \epsilon\rangle, \emptyset)|\mathcal{S}\}\end{array}}$$

$$(4)\ \frac{\begin{array}{c} r \equiv p(o', \bar{x}', \bar{y}') \leftarrow guard', b'_1, \ldots, b'_n \ll P, newenv(r) = tv_1 \\ tv_1(o') = tv(o), tv_1(\bar{x}') = tv(\bar{x}), tv_1(\bar{y}') = tv(\bar{y}), eval_{guard}(guard, tv) = true\end{array}}{\{\mathtt{ob}(o, C, h, \langle tv, \mathtt{call}(\mathtt{b}, p(o, \bar{x}, \bar{y})) \cdot s\rangle, \mathcal{Q})|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, \langle tv \cup tv_1, b'_1 \cdots b'_n \cdot s\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(5)\ \frac{\begin{array}{c} r \equiv p(this', \bar{x}', y') \leftarrow b'_1, \ldots, b'_n \ll P, tv(o_1) = o_2, \mathtt{fn} \text{ is a fresh future name} \\ newenv(r) = tv_1, tv_2 = tv_1[this' \mapsto o_2, \bar{x}' \mapsto tv(\bar{x}), \mathtt{r} \mapsto (y', \mathtt{fn})]\end{array}}{\begin{array}{c}\{\mathtt{ob}(o, C, h, \langle tv, \mathtt{call}(\mathtt{m}, p(o_1, \bar{x}, y)) \cdot s\rangle, \mathcal{Q}), \mathtt{ob}(o_2, D, h_1, \langle tv_3, s_1\rangle, \mathcal{Q}')|\mathcal{S}\} \leadsto \\ \{\mathtt{ob}(o, C, h, \langle tv[y \mapsto \mathtt{fn}], s\rangle, \mathcal{Q}), \mathtt{ob}(o_2, D, h_1, \langle tv_3, s_1\rangle, \{\langle tv_2, b'_1 \cdots b'_n\rangle\} \cup \mathcal{Q}'), \mathtt{fut}(\mathtt{fn}, \bot)|\mathcal{S}\}\end{array}}$$

$$(6)\ \frac{\mathtt{r} \notin dom(tv)}{\{\mathtt{ob}(o, C, h, \langle tv, \epsilon\rangle, \mathcal{Q})|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, \epsilon, \mathcal{Q})|\mathcal{S}\}}$$

$$(7)\ \frac{\mathtt{r} \in dom(tv), (y, \mathtt{fn}) = tv(\mathtt{r}), v = tv(y)}{\{\mathtt{ob}(o, C, h, \langle tv, \epsilon\rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, \bot)|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, \epsilon, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v)|\mathcal{S}\}}$$

$$(8)\ \frac{\mathtt{fn} = tv(y),\ v \neq \bot}{\{\mathtt{ob}(o, C, h, \langle tv, x := y.\mathsf{get} \cdot s\rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v)|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, \langle tv[x \mapsto v], s\rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v)|\mathcal{S}\}}$$

$$(9)\ \frac{eval_g(g, h, tv, \mathcal{S}) = true}{\{\mathtt{ob}(o, C, h, \langle tv, \mathtt{await}\ g \cdot s\rangle, \mathcal{Q})|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, \langle tv, s\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(10)\ \frac{eval_g(g, h, tv, \mathcal{S}) = false}{\{\mathtt{ob}(o, C, h, \langle tv, \mathtt{await}\ g \cdot s\rangle, q)|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, \epsilon, \{\langle tv, \mathtt{await}\ g \cdot s\rangle\} \cup \mathcal{Q})|\mathcal{S}\}}$$

$$(11)\ \frac{s \in Q}{\{\mathtt{ob}(o, C, h, \epsilon, \mathcal{Q})|\mathcal{S}\} \leadsto \{\mathtt{ob}(o, C, h, s, \mathcal{Q} - \{s\})|\mathcal{S}\}}$$

Fig. 3 Operational Semantics

7), the entry `r` is used to look for the corresponding future variable in order to update the $\bot$ value with the one which is returned. Note that rule 9 checks if a future variable is ready. In such case the computation proceeds. Otherwise, in rule 10, the `await` task in introduced is the corresponding queue, and the processor is released.

The instruction get blocks the execution until the future variable has a value in 8. If the evaluation of the guard in an `await` instruction succeeds, the execution continues in rule 9. Otherwise, in rule 10, the `await` task in introduced is the corresponding queue, and the processor is released. In rule 11 another task is dequeued (because the current one terminated or released the processor). Note that this rule is applicable thanks to rules 6 (resp. 7) which corresponds to the complete execution of a block (resp. a method).

We assume that executions start from a main method. Thus, the *initial configuration* is of the form $\{\!|\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv, s\rangle, \emptyset)|\!\}$, where local variables in *tv* are initialized to their default values. The execution then proceeds non-deterministically by applying the execution steps in Fig. 3. The execution finishes in a *final configuration* in which all events are either future events or objects of the forms $\mathtt{ob}(o, C, h, \epsilon, \emptyset)$ or $\mathtt{ob}(o, C, h, \langle tv, \epsilon\rangle, \emptyset)$. Executions can be regarded as *traces* of the form $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}_n$ where $\mathcal{S}_n$ is a final configuration.

Example 3 Consider the main method of the running example (Figure 1). After executing the constructors we reach a configuration with three objects:

$$\{\!|\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}, bc\rangle, \emptyset), \mathtt{ob}(o_1, \mathit{FileIS}, h_{o_1}, \epsilon,), \mathtt{ob}(o_2, \mathit{FileIS}, h_{o_2}, \epsilon,)|\!\}$$

where *bc* corresponds to the sequence of instructions from (*) on. After processing both asynchronous calls (rule 5) consecutively, the new state takes the form:

$$\begin{array}{l}\{\!|\ \mathtt{ob}(o_1, \mathit{FileIS}, h_{o_1}, \epsilon, \{\langle tv_{o_1}, body_{o_1}\rangle\}), \mathtt{fut}(\mathtt{fn}_1, \bot),\\ \quad \mathtt{ob}(o_2, \mathit{FileIS}, h_{o_2}, \epsilon, \{\langle tv_{o_2}, body_{o_2}\rangle\}), \mathtt{fut}(\mathtt{fn}_2, \bot),\\ \quad \mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}[f_1 \mapsto \mathtt{fn}_1, f_2 \mapsto \mathtt{fn}_2], bc'\rangle, \emptyset)\ |\!\}\end{array}$$

where $body_{o_1}$ (resp. $body_{o_2}$) is the renamed body of method readOnce (resp. readBlock). Furthermore, tv_{o_1} (resp. tv_{o_2}) stores the assignment $tv_{o_1}(\mathtt{r}) = (f_1, \mathtt{fn}_1)$ (resp. $tv_{o_2}(\mathtt{r}) = (f_2, \mathtt{fn}_2)$). When the event $\langle tv_{o_1}, body_{o_1}\rangle$ is extracted from the queue of o_1 (rule 11), its complete processing will replace $\mathtt{fut}(\mathtt{fn}_1, \bot)$ by $\mathtt{fut}(\mathtt{fn}_1, v)$ (rule 7), where v is the value returned by the method readOnce. Note then that rule 9 can be used to process the instruction await f_1? of the object main. At this point the new state will take this form:

$$\begin{array}{l}\{\!|\ \mathtt{ob}(o_1, \mathit{FileIS}, h_{o_1}, \epsilon, \emptyset), \mathtt{fut}(\mathtt{fn}_1, v),\\ \quad \mathtt{ob}(o_2, \mathit{FileIS}, h_{o_2}, \epsilon, \{\langle tv_{o_2}, body_{o_2}\rangle\}), \mathtt{fut}(\mathtt{fn}_2, \bot),\\ \quad \mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}[f_1 \mapsto \mathtt{fn}_1, f_2 \mapsto \mathtt{fn}_2], bc''\rangle, \emptyset)\ |\!\}\end{array}$$

3 Cost and Cost Models for Concurrent Programs

We now define the notion of cost we aim at approximating by static analysis. An execution step is annotated as $\mathcal{S} \rightsquigarrow_o^b \mathcal{S}'$, which denotes that we move from a state $\mathcal{S}$ to a state $\mathcal{S}'$ by executing instruction b in object o. Note that from a given state there may be several possible execution steps that can be taken since we have no assumptions on task scheduling. In order to quantify the cost of an execution step, we use a generic cost model $\mathcal{M} : \mathit{Ins} \mapsto \mathbb{R}$ which maps instructions built using the grammar in Section 2.2 to real numbers. The cost of an execution step is defined as $\mathcal{M}(\mathcal{S} \rightsquigarrow_o^b \mathcal{S}') = \mathcal{M}(b)$.

In the execution of sequential programs, the *cumulative cost of a trace* is obtained by applying a given cost model to each step of the trace. In our setting, this has to be extended because, rather than considering a single machine in which all steps are performed, we have a potentially distributed setting, with