

---

**jSyX: UNA MÁQUINA DE EJECUCIÓN SIMBÓLICA PARA *JAVA*  
*BYTECODE* Y SU APLICACIÓN A LA GENERACIÓN DE TESTS**

---

**Eduardo Tarascón Quintas**

**José Ángel García Fernández**



**PROYECTO SISTEMAS INFORMÁTICOS**

Departamento de Sistemas Informáticos y Computación

Facultad de Informática

Universidad Complutense de Madrid

Madrid, 21 de junio de 2013

Director Miguel Gómez-Zamalloa Gil




## **Autorización de difusión y utilización**

---

**L**os abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y 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.

Eduardo Tarascón Quintas

José Ángel García Fernández

A handwritten signature in black ink, appearing to read 'José Ángel García Fernández', with a large, stylized initial 'J' and 'G'.

Madrid, 21 de junio de 2013

## Agradecimientos

---

Queremos agradecer a Jon Pearce de Saint José State University por facilitarnos parte del código de su implementación que nos sirvió como base sobre la cual desarrollar este proyecto.

## Índice General

---

<b>Índice de figuras .....</b>	<b>V</b>
<b>Resumen .....</b>	<b>VII</b>
<b>Abstract.....</b>	<b>IX</b>
<b>1. Introducción .....</b>	<b>1</b>
1.1. Trabajos relacionados .....	1
<b>2. Objetivos .....</b>	<b>2</b>
<b>3. Creando una máquina virtual <i>Java</i> .....</b>	<b>4</b>
3.1. Máquinas virtuales .....	5
3.2. Introducción a una <i>JVM</i> .....	6
3.3. Estructura .....	7
3.4. Ejecución .....	7
3.5. <i>BCEL</i> .....	8
3.6. Detalles sobre los archivos <i>Java Class</i> .....	9
3.7. Implementación .....	11
3.7.1. Explicaciones sobre paquetes .....	12
3.7.2. Explicaciones sobre clases.....	12
3.7.3. Listado de instrucciones admitidas .....	17
3.7.4. Problemas durante la implementación .....	17
<b>4. Creando la ejecución simbólica.....</b>	<b>18</b>
4.1. Programación Simbólica.....	19
4.2. Programación con restricciones.....	19
4.3. <i>Choco</i> .....	20

4.3.1. <i>Model</i> .....	21
4.3.2. <i>Solver</i> .....	21
4.4. Implementación .....	21
4.4.1. Problemas durante la implementación .....	23
4.5. Ejemplo .....	23
<b>5. Cliente .....</b>	<b>26</b>
5.1. Información sobre el cliente .....	27
5.2. <i>Commons CLI</i> .....	28
5.3. Detalles de implementación .....	29
5.4. Ejemplos de uso .....	30
5.4.1. Ejemplos de ejecución no simbólica .....	30
5.4.2. Ejemplos de ejecución simbólica .....	40
<b>6. Conclusiones .....</b>	<b>48</b>
6.1. Reflexión final .....	49
6.2. Trabajo futuro .....	49
<b>Bibliografía .....</b>	<b>51</b>
<b>Apéndices .....</b>	<b>53</b>
Apéndice A: Listado instrucciones <i>bytecode</i> admitidas .....	53

## Índice de figuras

---

<i>Figura 1. Ejemplo de máquina virtual de sistema .....</i>	<i>5</i>
<i>Figura 2. Ejemplo de máquina virtual de proceso .....</i>	<i>5</i>
<i>Figura 3. Esquema de trabajo de Java .....</i>	<i>6</i>
<i>Figura 4. Estructura de una JVM .....</i>	<i>7</i>
<i>Figura 5. Estructura del paquete Classfile .....</i>	<i>8</i>
<i>Figura 6. Estructura del paquete generic .....</i>	<i>9</i>
<i>Figura 7. Estructura general de un archivo .class.....</i>	<i>10</i>
<i>Figura 8. Tipos de constantes de la Constant Pool .....</i>	<i>11</i>
<i>Figura 9. Diagrama de paquetes de jSyX.....</i>	<i>12</i>
<i>Figura 10. Diagrama de clases base de la JVM .....</i>	<i>13</i>
<i>Figura 11. Diagrama de clases recubridoras de jSyX.....</i>	<i>15</i>
<i>Figura 12. Diagrama de la clase Natives para ejecución nativa de jSyX.....</i>	<i>16</i>
<i>Figura 13. Diagrama de clases con los tipos de jSyX .....</i>	<i>16</i>
<i>Figura 14: Arquitectura general de Choco. Existen dos partes claramente diferenciadas, una para el Modelado y otra para el Resolutor. ....</i>	<i>20</i>
<i>Figura 15: Árbol de ejecución foo con nomenclatura de símbolos.....</i>	<i>24</i>
<i>Figura 16 : Árbol de ejecución foo. ....</i>	<i>24</i>
<i>Figura 17. Mensaje de ayuda para uso de jSyX .....</i>	<i>27</i>

<i>Figura 18. Obtener versión de jSyX usando parámetros .....</i>	<i>28</i>
<i>Figura 19. Diagrama de clases para cliente .....</i>	<i>30</i>
<i>Figura 20. Prueba de ejecución no simbólica con -n -d -c.....</i>	<i>31</i>
<i>Figura 21. Prueba de ejecución no simbólica con -n -o -d -c.....</i>	<i>33</i>
<i>Figura 22. Prueba de ejecución no simbólica con -n -i -d -c y redirección.....</i>	<i>34</i>
<i>Figura 23. Prueba de ejecución no simbólica con -d -n -c de métodos nativos.....</i>	<i>35</i>
<i>Figura 24. Prueba de ejecución no simbólica con -n -d -c sobre statics.....</i>	<i>37</i>
<i>Figura 25. Prueba de ejecución no simbólica con -n -b -c para visionado de bytecode39</i>	
<i>Figura 26. Prueba de ejecución simbólica con -b -c y -m para visionado de bytecode de método.....</i>	<i>41</i>
<i>Figura 27. Prueba de ejecución simbólica con -d -m -c sobre abs.....</i>	<i>42</i>
<i>Figura 28. Prueba de ejecución simbólica con -l -m -c sobre abs.....</i>	<i>42</i>
<i>Figura 29. Prueba de ejecución simbólica con -l -d -m -c sobre objetos.....</i>	<i>45</i>
<i>Figura 30. Prueba de ejecución simbólica con -l -m -c -array-size sobre arrays .....</i>	<i>46</i>



La ejecución simbólica de programas consiste en una forma de ejecución donde se sustituyen los valores de entrada por símbolos formado por valores arbitrarios acotados. Esto dará como resultado múltiples caminos en los que los símbolos irán adquiriendo una serie de restricciones, dando lugar a un mapa completo de ejecución.

Ello supondrá una valiosa herramienta que se podrá utilizar tanto en verificación de programas, como en *testing*, ya que tendremos información de como funciona el programa en cada una de sus ramas de ejecución.

En este proyecto hemos diseñado y desarrollado una herramienta llamada *jSyX*, una máquina de ejecución simbólica capaz de cargar archivos *.class* propios de *Java* y recibiendo como parámetros una clase y su método, ejecutarlo, o bien, simbólicamente, o de manera normal. A parte se podrá obtener información sobre el *bytecode* del archivo *.class*

El objetivo del proyecto es una primera aproximación a la ejecución simbólica para programas escritos en *Java*.

## **Palabras Clave**

---

**Java**

**Máquina Virtual Java**

**Choco**

**BCEL**

**Programación simbólica**

**Programación con restricciones**

**Tests**

## Abstract

---

The symbolic execution of programs consists in a kind of execution where the inputs are replaced by symbols formed with arbitrary bounded values. This will result as multiple paths where the symbols will acquire a series of constraints; therefore a complete map of execution will be created.

It will suppose a valuable tool that could be use as program verification, or testing, in which we already have information of how the programs works in every single execution branch.

In this project we have design and develop a tool called *jSyX*, a symbolic execution machine capable of loading *Java .class* files, and receiving as parameters a class and a method, execute either symbolically or normal execution. Also it could obtain information about the *bytecode* of the *.class* file

The target of the project is a first approximation to the symbolic execution of programs written in *Java*.

## **Keywords**

---

**Java**

**Java Virtual Machine**

**Choco**

**BCEL**

**Symbolic programming**

**Constraints programming**

**Tests**

La ejecución simbólica ha cobrado mucho atención en los últimos años, a pesar de que la idea surgió hace tres décadas [1] . Sin embargo, hasta ahora no resultaba práctica. Gracias a avances en resolutores de restricciones como choco, la ejecución simbólica actualmente resulta factible y es una opción a considerar en el mundo del *testing* y verificación de programas entre otros.

Hoy en día la realización de pruebas o *testing* en software conlleva gran parte del tiempo y recursos de cualquier proyecto software. El coste en infraestructura software dedicada al *testing* es de \$59,5 mil millones al año solo en Estados Unidos [2] . Y se estima que el coste de validación de software supera el 50%.

Esto es debido en gran parte a la falta de automatización. La mayoría de los test son creados por los propios desarrolladores de software, que serán los encargados de crear el máximo posible de escenarios de ejecución. Esta tarea a parte de ser manual, no siempre consigue encontrar los casos de pruebas eficientes que abarquen todas las posibilidades siendo difícil saber cuando terminar de realizar todas las pruebas necesarias para el correcto funcionamiento de la aplicación. Por ello la gran mayoría del software que sale al mercado contiene en mayor o menor medida *bugs*.

La ejecución simbólica es una de las herramientas más importante para hacer razonamientos estáticos acerca del comportamiento de programas. Se puede encontrar en multitud de herramientas como generadores automáticos de pruebas, verificadores, visualizadores y depuradores de programa.

Herramientas tales como *Valgrind* o *Purify* basada en la ejecución dinámica para *debugging* con el objetivo de conseguir código más fiable, depende entradas concretas que se introduzcan para detectar errores. Así como JUnit, el usuario creará casos de *test* intentando abarcar la mayor parte de casos. Todo esto no es necesario en la



ejecución simbólica ya que la entrada se generará automáticamente y no está limitada a una cierta entrada para encontrar errores.

Este proyecto parte de la misma idea que un proyecto anterior denominado “PET, *Partial Evaluation-based Test Case Generator for Bytecode*”. Los principales investigadores son Elvira Albert, Miguel Gómez-Zamalloa, Germán Puebla y José Miguel Rojas desarrollado en 2009-2010. PET está escrito en PROLOG, sin embargo *jSyX* está escrito enteramente en Java por lo que se puede ejecutar en cualquier dispositivo con máquina virtual Java (JVM) lo que otorga una mayor portabilidad.

El hecho de hacerlo en Java no es debido únicamente a la portabilidad del programa, sino que también es un lenguaje ampliamente extendido, solo superado por C. Otra gran ventaja es la posibilidad de ejecutar código nativo fácilmente. Dado que la complejidad del programa en cuestión puede llegar a ser insostenible para la ejecución simbólica, ejecutar estos métodos nativos supone un ahorro de recursos bastante importante.

### 1.1. Trabajos relacionados

Como hemos mencionado anteriormente la ejecución simbólica ha adquirido protagonismo en los últimos años y podemos encontrar varios proyectos orientados al *testing*:

- *Pex - white box test generation for .NET* [3]: implementa la ejecución dinámica simbólica para generar entradas de *test* para *.NET*, soporta lenguaje como C#, Visual Basic y F#. También permite el uso de ejecución concreta de valores para simplificar las restricciones, que serán resueltas por el resolutor SMT.
- *EXE* [4]: es una herramienta de ejecución simbólica diseñada para C que está diseñada para lidiar con la complejidad de códigos de sistemas, EXE modela memoria a nivel de bit. Como resolutor de restricciones utiliza STP combinado con una serie de resolutores de alto nivel capaz de eliminar y capturar restricciones innecesarias.
- *DART - Directed Automated Random Testing* [5]: ejecuta un programa con valores aleatorios o valores concretos de las entradas y añade restricciones simbólicas a las entradas según se encuentre con instrucciones condicionales en su camino, y finalmente utiliza dichas restricciones para inferir variantes a las entradas previas. Este proceso es repetido automáticamente hasta encontrar todos los caminos posibles.

La elaboración de este proyecto tiene como objetivo fundamental la creación de una máquina virtual *Java* que permita ejecución simbólica, de manera que pueda ser utilizada para el desarrollo automático de Tests.

Esta máquina virtual deberá ser capaz de leer y *parsear* los archivos *.class* de *Java*, leyendo cada instrucción y realizando su ejecución de manera simbólica, se tomarán los datos de entrada del proceso como símbolos, a lo largo de la ejecución se tomarán uno u otro camino del árbol según restricciones establecidas por las instrucciones donde los símbolos podrán ir tomando valores. Una vez llegado a una hoja del árbol la salida se mostrará en función de dichas restricciones

Concretando esta especificación los objetivos del proyecto desarrollado serán los siguientes:

1. Creación de un intérprete simple para archivos *Java Class*: Primero trataremos de obtener una aplicación simple en la que podamos leer y *parsear* un archivo *.class* y realizar algunas operaciones con esta información.
2. Ampliación para obtener una *JVM* funcional: Una vez que seamos capaces de acceder a la información del *.class*, deberemos crear toda la estructura de una *JVM* para poder realizar la ejecución completa.
3. Modificación para permitir ejecución simbólica simple de enteros: En una primera aproximación trataremos de elaborar una *JVM* que permita una ejecución simbólica para programas simples que trabajen con números enteros, permitiendo la creación del árbol y el establecimiento de restricciones.
4. Ampliación para obtener ejecución simbólica para más tipos: Con la base obtenida, ampliaremos la ejecución para que permita otros tipos de datos, no solo simples, sino también sobre objetos.



5. Creación de un cliente de consola para facilitar el uso de la aplicación: Se elaborará una interfaz de cliente desde consola para que cualquiera pueda de manera sencilla utilizar las funcionalidades de *jSyX*.

Los dos primeros puntos, creación del intérprete y creación de la *JVM* completa van ampliamente ligados, todo este proceso será comentado en el capítulo 3, donde veremos la estructura y modelo de ejecución de la *JVM*, dando información de nuestra propia implementación.

Los puntos 3 y 4 serán tratados en el capítulo 4, dando información de cómo funciona la ejecución simbólica y como se ha conseguido.

El último punto sobre el cliente será explicado en el capítulo 5.

## CAPÍTULO 3

---

### 3. Creando una máquina virtual *Java*

**E**n este capítulo presentamos la realización de la máquina virtual de *Java*. Dando primero una introducción genérica a las máquinas virtuales, centrándonos luego en la máquina virtual de *Java*, aportando información sobre su estructura, modelo de ejecución así como librerías usadas en su implementación, de la que hablaremos al final del capítulo.

La creación de esta máquina virtual nos permitirá tener una base que usar en el posterior desarrollo de la ejecución simbólica y con restricciones que será al final *JSyX*.

#### 3.1. Máquinas virtuales

Una máquina virtual es un contenedor de software que simula a un ordenador [6]. Se pueden englobar en dos categorías:

- Máquinas virtuales de sistema
- Máquinas virtuales de proceso

Las primeras permiten a la máquina física sobre la que se ejecuta multiplicarse de manera que cada máquina virtual puede ejecutar su propio sistema operativo.

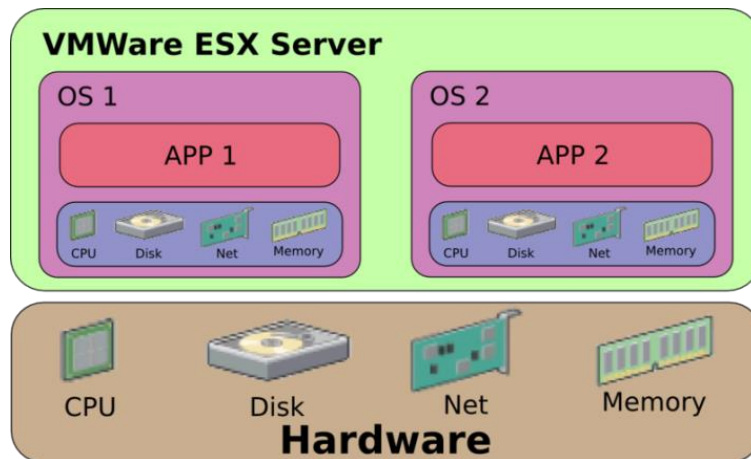


Figura 1. Ejemplo de máquina virtual de sistema

Las segundas y las que nos interesan para nuestro proyecto, proporcionan un entorno de ejecución independiente del hardware y del sistema operativo subyacente, de esta manera la ejecución del programa será siempre la misma, independientemente de donde se ejecute.

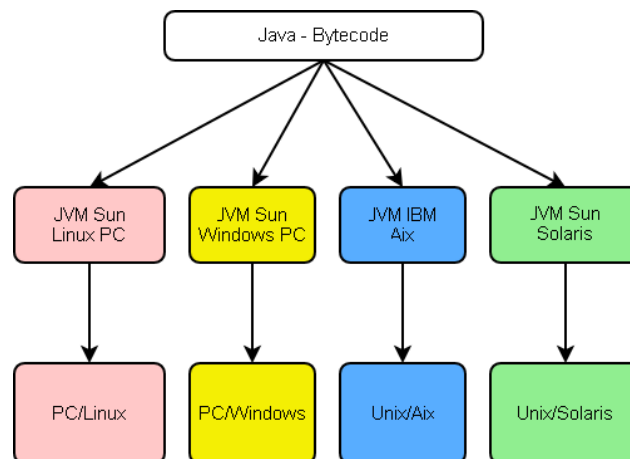


Figura 2. Ejemplo de máquina virtual de proceso

Como se podrá imaginar esto tiene sus problemas, el mayor de ellos es la pérdida de rendimiento debido a que el programa se está ejecutando sobre una capa de software extra que añade más complejidad al sistema, de manera que no se alcanzará el mismo rendimiento que si esta ejecución se realizara sobre el sistema anfitrión.

### 3.2. Introducción a una JVM

En términos de nuestro proyecto, la *JVM* (*Java Virtual Machine*) es una máquina virtual de proceso que permite que cualquier programa *Java* se pueda ejecutar sobre cualquier plataforma (siempre y cuando disponga de su *JVM*), soportando el famoso “*Write once, Run anywhere*” [7].

Los programas escritos en *Java* son traducidos a un código intermedio denominado *bytecode*, este código que trabaja al estilo de los lenguajes ensambladores es guardado en archivos *.class* (agrupados en bytes).

Típicamente un lenguaje ensamblador consiste en una serie de instrucciones simples que se corresponden con órdenes para el procesador sobre el que trabaja, estas instrucciones suelen estar formadas por un código de operación y cero o más operandos, este esquema es seguido por el *bytecode* de *Java*, con la ventaja de que al tratarse de un código intermedio, es más abstracto consiguiendo de esta manera disminuir la dependencia de la interpretación del mismo respecto del hardware específico.

Estos archivos *.class* contienen la información de cada clase además de las instrucciones *bytecode*. Estos archivos son cargados dinámicamente en un intérprete para su ejecución. [8]

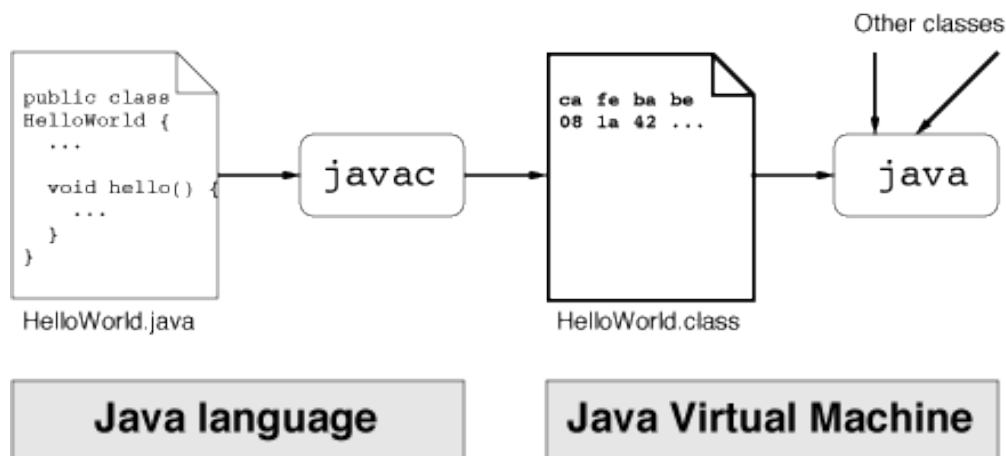


Figura 3. Esquema de trabajo de Java

Un intérprete en general permite la ejecución de programas de alto nivel, realizando la traducción de éste a un código máquina a medida que van leyendo instrucciones, a diferencia de los compiladores que realizan la traducción completa previa ejecución. Para el caso de *Java*, el intérprete permite la ejecución del *bytecode*, código intermedio previamente compilado desde el archivo fuente original, permitiendo su ejecución directa o la compilación conocida como *just-in-time (JIT)* a código máquina nativo del sistema para mejorar el rendimiento. [9]

Para el trabajo con los *.class*, usaremos la librería *BCEL*, más adelante profundizaremos en ella.

### 3.3. Estructura

Podemos ver a la *JVM* como una maquina con tres partes, una zona de memoria llamada *ClassArea*, otra llamada *Heap* y otra para *frames*. La primera guarda la información de clases y métodos, en la segunda se guarda la información de objetos dinámicos cada vez que se invoca a *new*, además cuando un objeto deja de ser útil es eliminado por el recolector de basura, la tercera, denominada globalmente como *Java stack*, guarda la información de cada *frame* (o *stack frame*), que son un espacio de datos que permite llevar un registro de las invocaciones a métodos que se han hecho y los datos asociados a dicho método, cada una de estas estructuras cuenta con una pila propia, un espacio para variables locales y parámetros, e información de control (*frame* padre, PC y método operando). [10]

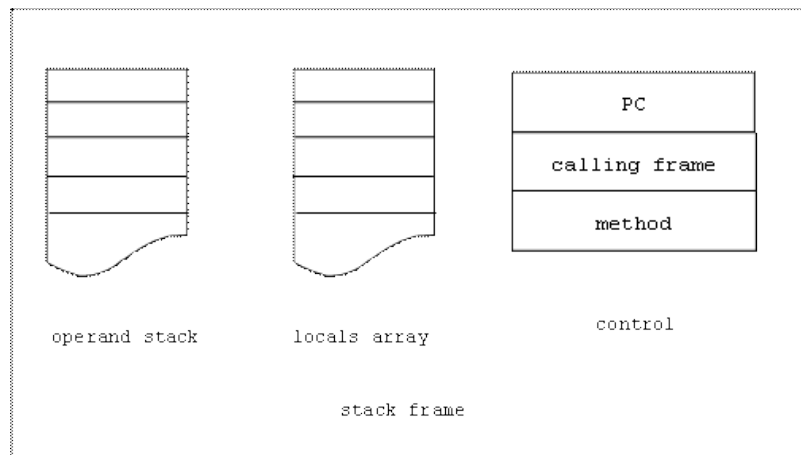


Figura 4. Estructura de una JVM

La *JVM* sigue el patrón de un intérprete basado en pila creando un *frame* por cada ejecución de método, cuando el método termina el control pasa al *frame* padre, de ahí la denominación de *Java Stack*.

### 3.4. Ejecución

La ejecución se hace a través de 212 instrucciones que se ocupan de distintas tareas, si existen para distintos tipos usan un prefijo (i para *integer*, d para *double*, etc).

- Operaciones con la pila: poner elementos (*bipush*), quitarlos (*pop*), duplicarlos (*dup*), etc.
- Aritméticas: sumas, restas, multiplicaciones de distintos tipos (*iadd*, *fadd*, *dmul*).
- Carga y guardado: cargar (*iload*) o guardar elementos desde memoria local (*istore*).
- Control de flujo: salto mediante comparación de valores (*if\_icmpeq*), a subrutina, por excepciones (*athrow*).
- Acceso a campos: para acceder a campos de clases (*getfield*, *putfield*, etc).
- Invocación de métodos: para invocar métodos ya sean estáticos (*invokestatic*), de interfaces (*invokeinterface*), etc.
- Objetos: para reservar memoria de objetos (*new*) o arrays (*newarray*).

- Conversión de tipos: cambio de un tipo (*f2i*) a otro o comprobación de tipos (*checkcast*).

En nuestra máquina virtual no hemos realizado la implementación de las 212 instrucciones, aunque sí de la mayoría de ellas que son más ampliamente utilizadas, por lo que una gran cantidad de programas pueden ser ejecutados sin problemas, aun así la inclusión del resto de instrucciones podría ser una posible y recomendable ampliación para la mejora del programa.

### 3.5. BCEL

La librería *BCEL* (*Byte Code Engineering Library*) es una librería para analizar, manipular y crear archivos *Java Class* [11]. En nuestro contexto es una ayuda para parsear los archivos *.class* y obtener toda la información que necesitamos para la ejecución del *bytecode* en nuestra máquina virtual de *Java jSyX*.

Hacemos uso de dos de los paquetes principales de *BCEL*.

El primero se encarga de la parte estática teniendo su clase principal en *JavaClass* (*org.apache.bcel.classfile*). Con ella podemos acceder a la información guardada en *.class* sin preocuparnos de lecturas de bajo nivel, aunque no es posible ningún tipo de modificación. Podemos ver un diagrama de su estructura en la figura 5.

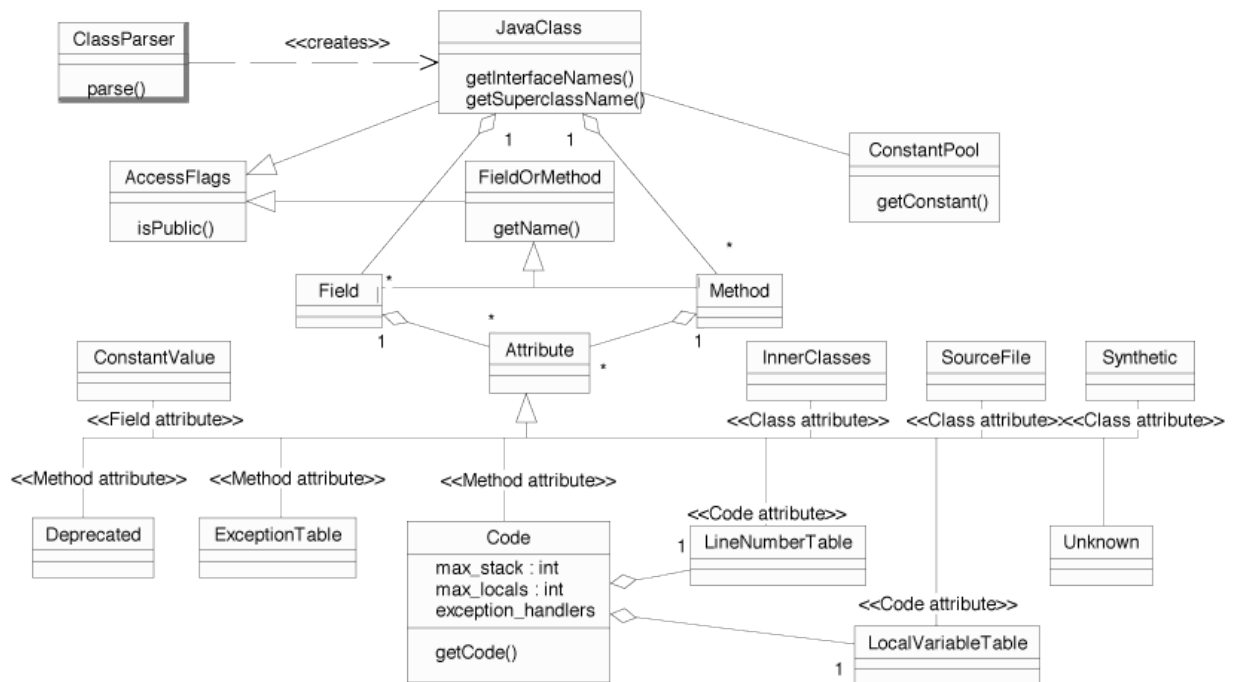
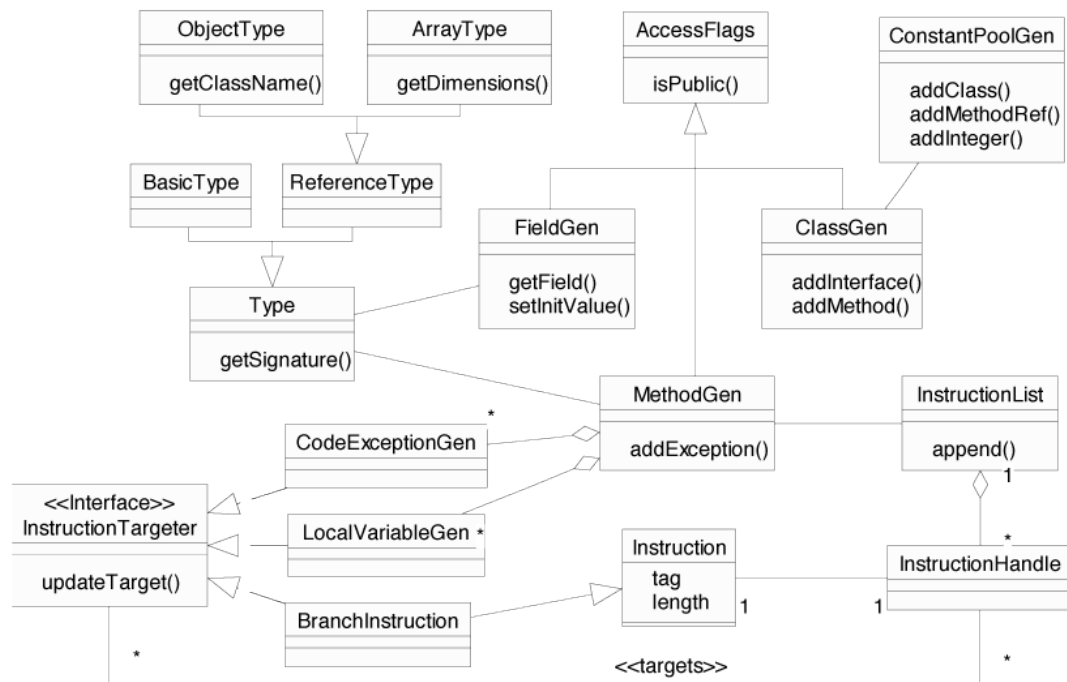


Figura 5. Estructura del paquete *Classfile*

Con la segunda *ClassGen* (*org.apache.bcel.generic*) podemos acceder a alguna de las opciones de la primera además de poder llevar a cabo modificaciones.



**Figura 6. Estructura del paquete generic**

En nuestra implementación hacemos uso de ambas aunque no nos es necesario hacer modificaciones en el código, debido a que ClassGen da acceso a algunos elementos que son necesarios a la hora de realizar la ejecución del *bytecode* de una manera más simple, como la lista de instrucciones encapsulada en InstructionHandle. A pesar de eso tenemos otra forma de ejecución que lee instrucciones directamente del *bytecode* a través del *method attribute code*.

Encapsulamos ambas estructuras en nuestro JClass, además tenemos generalmente para cada elemento de la librería una clase envoltorio que facilita el acceso a sus elementos.

### 3.6. Detalles sobre los archivos *Java Class*

Aunque al hacer uso de *BCEL* en nuestra aplicación podemos abstraernos de muchas de las características de bajo nivel de los archivos .class, comentaremos aquí algunas de sus características más importante.

Cualquiera que haya programado en Java sabrá que los archivos fuente son los `.java` y que al compilar se crea un archivo `.class`, como hemos dicho estos `.class` (agrupados en bytes), son los que la *JVM* usa para la ejecución, conteniendo las instrucciones *bytecode* además de la información de cada clase, almacenada en una estructura llamada *Constant Pool*.

En la siguiente imagen se puede ver una estructura general de este archivo.

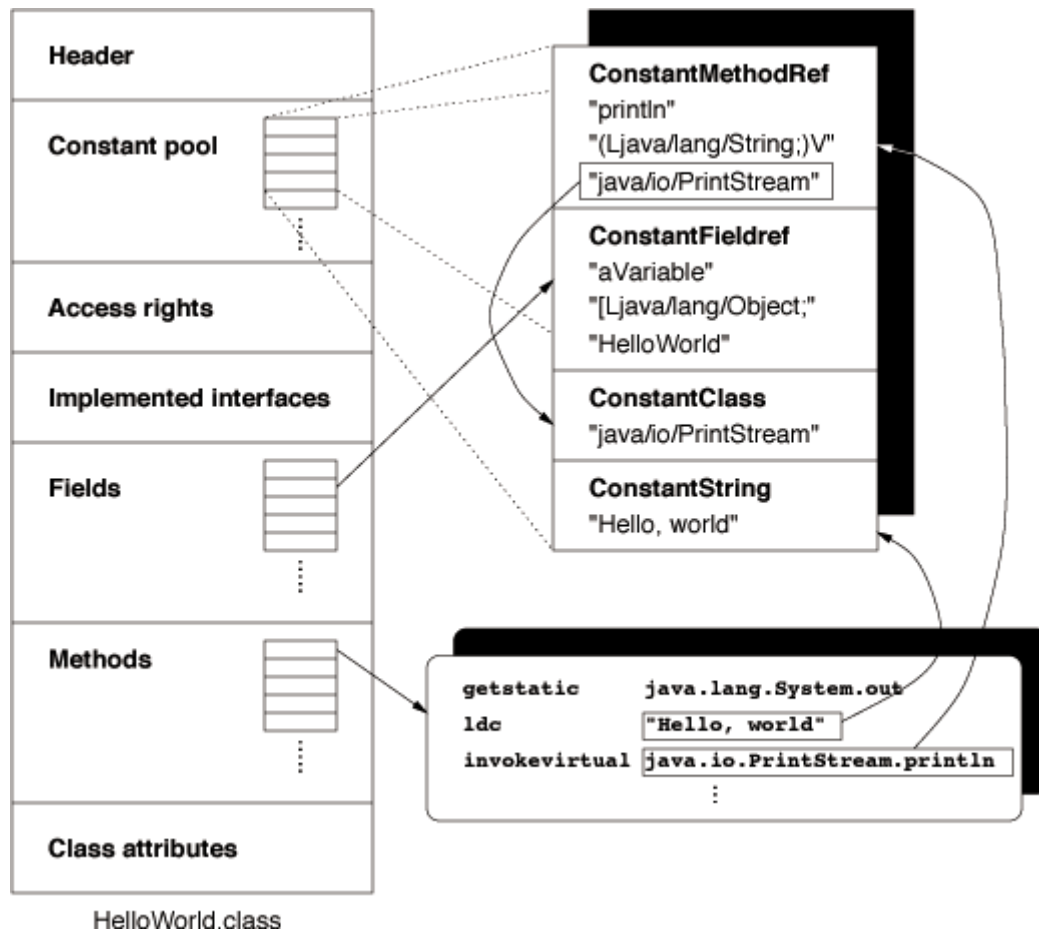


Figura 7. Estructura general de un archivo .class

Los archivos .class comienzan con una cabecera denominada *magic number* (0xCAFEFABE) [12], el número de versión y la *Constant Pool*.

Las instrucciones de la JVM no utilizan la información de ejecución de clases, interfaces, arrays, etc, sino que hace referencia a la información simbólica de la *Constant Pool*, así si una instrucción necesita saber sobre qué clase está ejecutando una instrucción *getstatic*, hará referencia a la *Constant Pool* y obtendrá esta información. [13] La siguiente imagen muestra los distintos tipos de constantes que podemos encontrar.



<i>Constant Type</i>
CONSTANT_Class
CONSTANT_Fieldref
CONSTANT_Methodref
CONSTANT_InterfaceMethodref
CONSTANT_String
CONSTANT_Integer
CONSTANT_Float
CONSTANT_Long
CONSTANT_Double
CONSTANT_NameAndType
CONSTANT_Utf8

Figura 8. Tipos de constantes de la Constant Pool

Después vienen los derechos de acceso de la clase codificados mediante una máscara de bits, a continuación las interfaces implementadas por la clase y una lista de los campos y métodos de la clase, la última parte se reserva para los atributos de clase como puede ser el nombre del archivo fuente que originó el `.class`.

Como mencionamos anteriormente, las instrucciones usan continuamente la estructura de la *Constant Pool*, en la figura 8 puede verse como para realizar la carga del String *"Hello, world"*, este se encuentra en la *Constant Pool* como un `ConstantString`. Normalmente ésta información está estructurada como una cadena de referencias, así para usar el método `println()` se dirige a un campo de tipo `ConstantMethodRef` que contiene su nombre y signatura y a su vez tiene una referencia para encontrar la clase que lo contiene `java.io.PrintStream` en un `ConstantClass`.

También mencionar la signatura del método, que contiene codificada la información de parámetros y retorno del mismo.

En el capítulo del cliente tendremos oportunidad de ver esto con más profundidad, haciendo uso del parámetro `-b` para obtener información del *bytecode* de los archivos *Java class*.

## 3.7. Implementación

Comentaremos los principales elementos para la implementación de la máquina virtual de *Java* sin ejecución simbólica [14]. Para desarrollar nuestra *JVM* hemos tomado como referencia la propia especificación oficial [8] y también una *JVM* desarrollada en C++ [15] que nos sirvió como referencia para concretar aspectos de la implementación.

### 3.7.1. Explicaciones sobre paquetes

Organizamos en 3 paquetes principales. En el paquete `com.bcel.Test` tenemos clases de prueba para la ejecución de la máquina virtual, la clase `Test` contiene distintos métodos para las pruebas, en la clase `A` tenemos algunos elementos para hacer pruebas sobre objetos.

El paquete `com.jsyx` es el principal de la máquina virtual, donde se encuentran las clases principales, tenemos otros dos subpaquetes, `com.jsyx.Test` para clases relacionadas con la ejecución y `com.jsyx.Test.tipos` donde tenemos la estructura de tipos de la máquina virtual, siendo la principal `JValue`, clase abstracta de la que heredan el resto de tipos.

Por último en el paquete `com.bcel.log` tenemos clases para la generación del archivo de log.

En la siguiente imagen mostramos un diagrama de paquetes donde puede verse esta estructura.

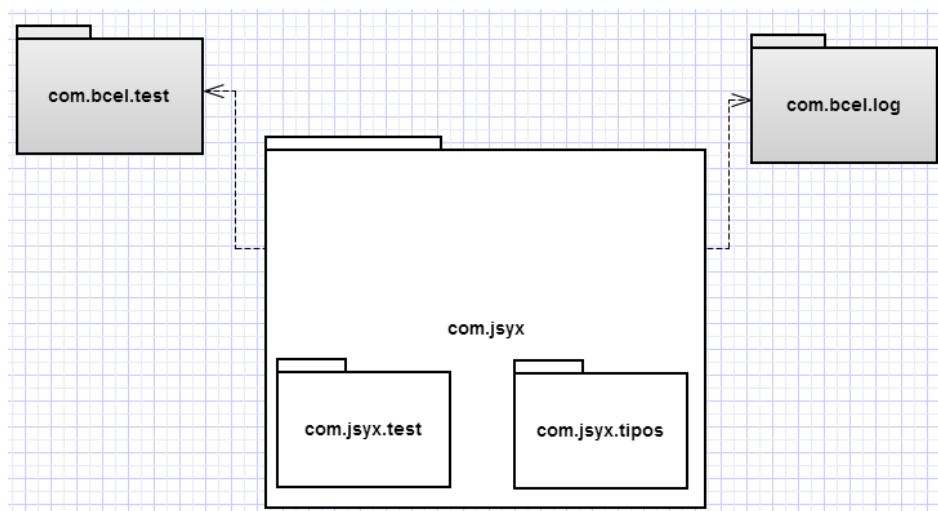


Figura 9. Diagrama de paquetes de jSyX

### 3.7.2. Explicaciones sobre clases

La clase `JVM` contiene la máquina virtual en sí, contamos con las 3 zonas de memoria mencionadas, un `StackFrame`, el `Heap` y el `ClassArea`. Tenemos 2 tipos de ejecución básica no simbólica, ejecutando haciendo uso del `OPCODE` directamente leyendo del `bytecode`, y ejecución mediante `InstructionHandle` llamada `IHPC`.

En la siguiente figura podemos ver un diagrama de clases para la interacción entre estas estructuras que hemos comentado.

### 3. Creando una máquina virtual Java

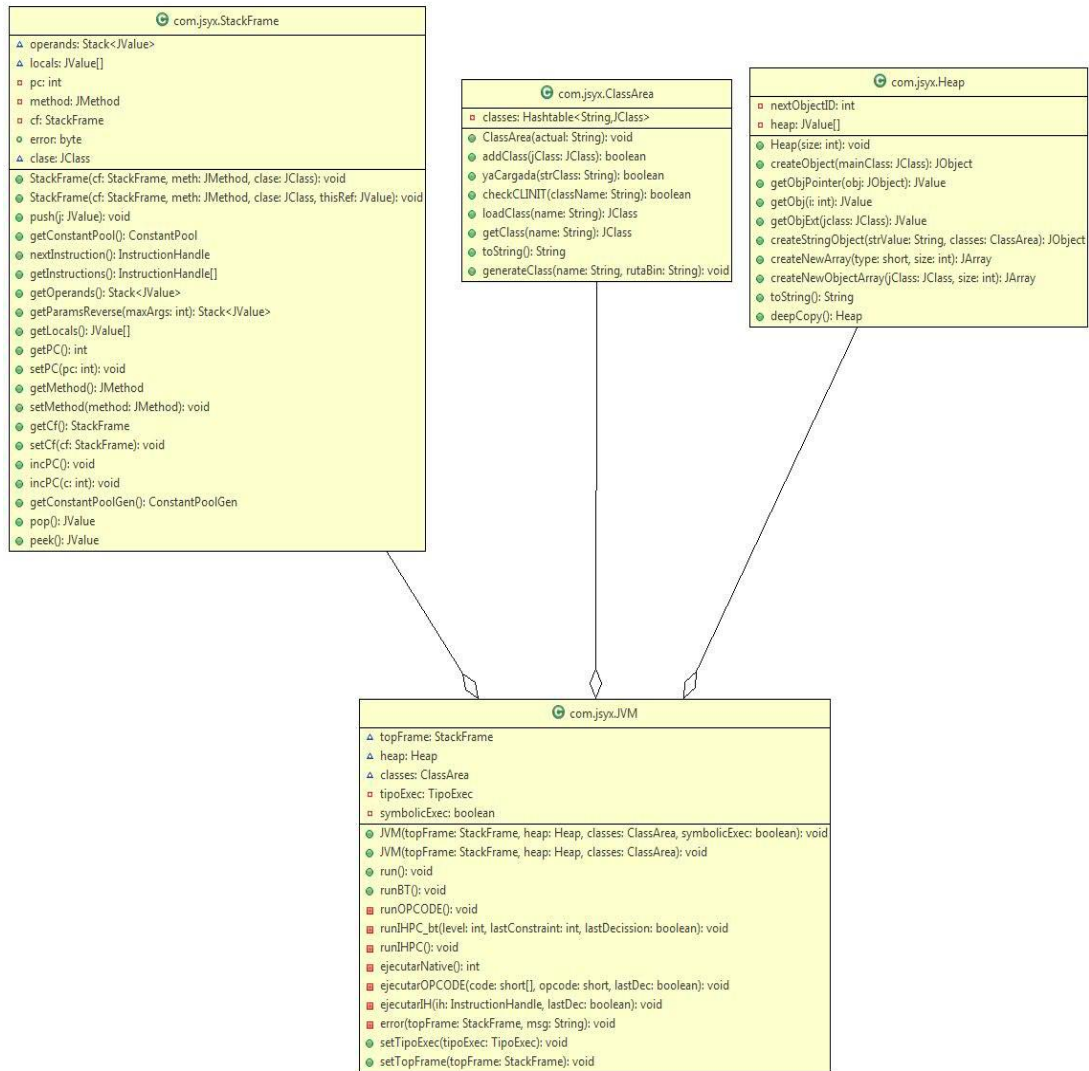


Figura 10. Diagrama de clases base de la JVM

En la ejecución *OPCODE* usamos la clase *Constants* de la librería *BCEL* para detectar el *OPCODE* y ejecutamos acorde a la instrucción obtenida, si necesitamos información extra para la instrucción tenemos que obtenerla haciendo uso directamente de la *ConstantPool* que nos facilita *BCEL*.

En la ejecución *IHPC* al disponer del *InstructionHandle* podemos usarlo para obtener la información que necesitamos para la ejecución en la mayoría de instrucciones, aunque en algunos casos no basta con ello y tendremos que actuar como con *OPCODE*, de todos modos en la mayoría de ocasiones facilita el trabajo el disponer de la instrucción.

Para la ejecución hemos diseñado métodos genéricos para cada instrucción, de esta manera pueden ser usados para ambos tipos de ejecución variando simplemente la obtención de la información que es necesaria pasarle para que la instrucción realice su operación.

El `StackFrame` como se comentó anteriormente dispone de la pila de operandos, la zona de variables locales, contador de programa, método actual y `StackFrame` padre, además añadimos la clase del método que se está ejecutando para facilitar algunas operaciones.

El Heap lo hemos diseñado mediante un *array* de `JValue` y un puntero con la siguiente posición libre del *array* además disponemos de métodos para crear los distintos elementos dinámicos.

No hemos implementado un recolector de basura, si bien las aplicaciones que hemos ejecutado no generaban demasiada sobrecarga aunque debería ser una ampliación a tener muy en cuenta en futuras revisiones de este proyecto.

El `ClassArea` que mantiene la información de clases se ha diseñado mediante un `HashTable` con clave el nombre de clase. Permite realizar la carga de clases, generación de clases usando *BCEL* comprobar si es necesario ejecutar el *CLINIT* y otros métodos de utilidad.

También disponemos de un surtido de clases recubridoras como se comentó antes, `JClass` para clases, `JMethod` para métodos, `JField` para campos de clases.

En la siguiente imagen podemos ver un diagrama de estas clases, todas ellas hacen uso de las clases proporcionadas por *BCEL* y ofrecen métodos para acceder a éstas.

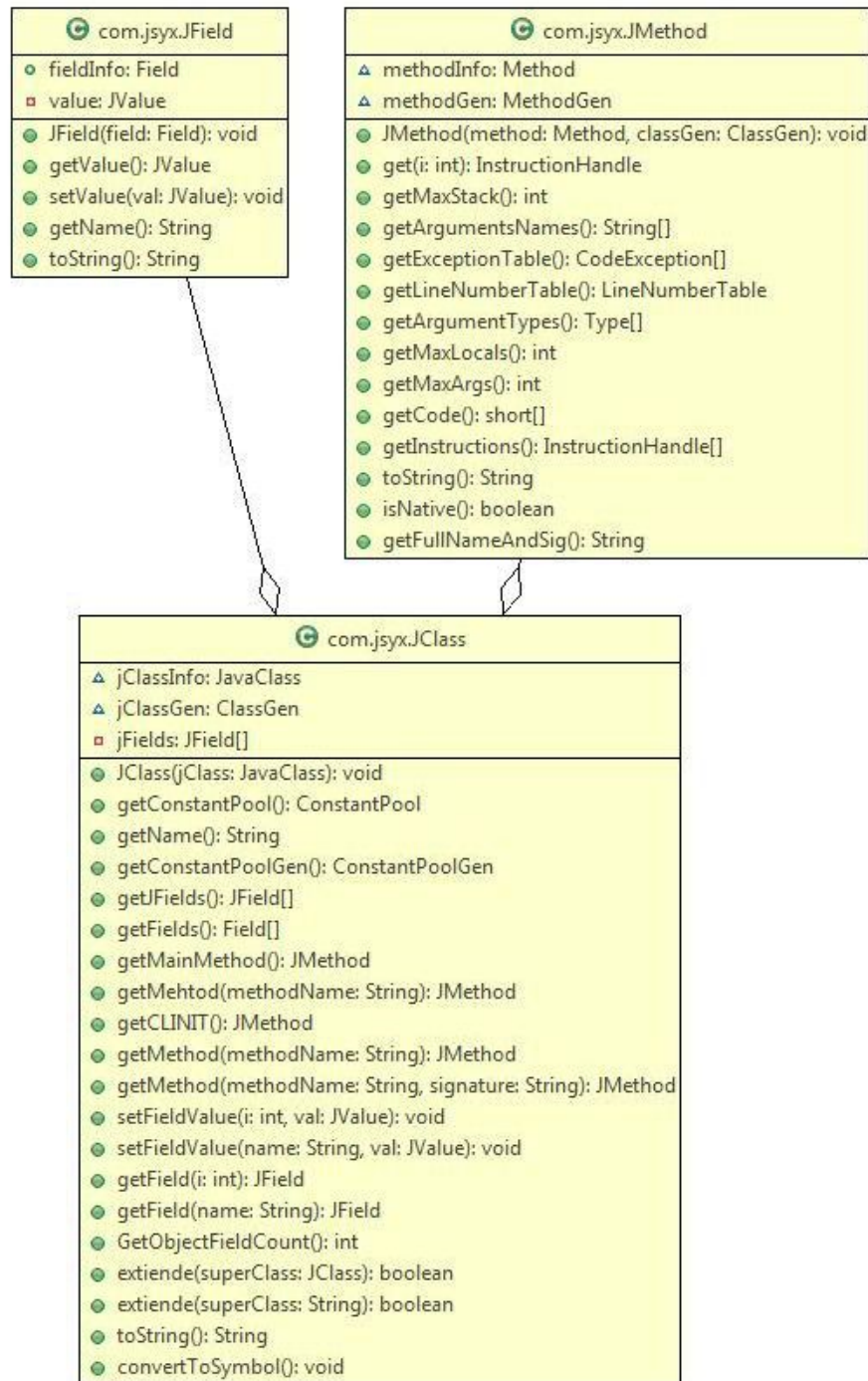


Figura 11. Diagrama de clases recubridoras de jsyX

Además se permite la ejecución de métodos nativos mediante la clase `Natives`, aunque bastante limitada. La ejecución de métodos nativos se realiza comprobando en la máquina virtual si el método está marcado con el flag `NATIVE`, si es así se obtiene su nombre y *signature* y se pasa a comprobar usando la clase `Natives` si se dispone de una implementación del mismo, en ese caso se realiza la ejecución y tras ella la *JVM* sigue su curso. Hemos optado por realizar esta ejecución directamente codificando usando *Java* para facilitar su implementación.

En la siguiente imagen podemos ver a esta clase, solo implementamos algunos métodos para probar su funcionamiento.

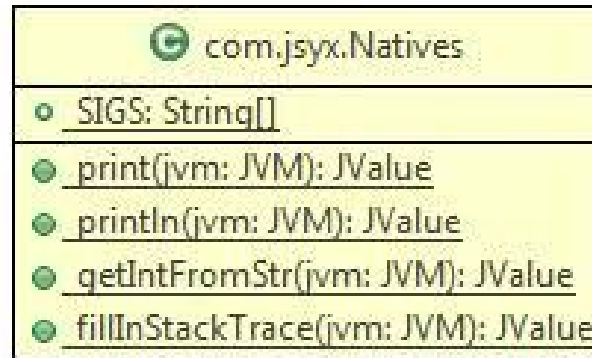


Figura 12. Diagrama de la clase Natives para ejecución nativa de jsyX

En cuanto a tipos se han incluido los que maneja la máquina virtual además para facilitar el trabajo se han creado algunos elementos extra para el trabajo con *arrays* y objetos. La clase base es JValue, abstracta de la que todas los demás extienden que contiene un elemento Type de BCEL.

En la siguiente imagen se puede ver un esquema de los distintos tipos que permite nuestra máquina virtual.

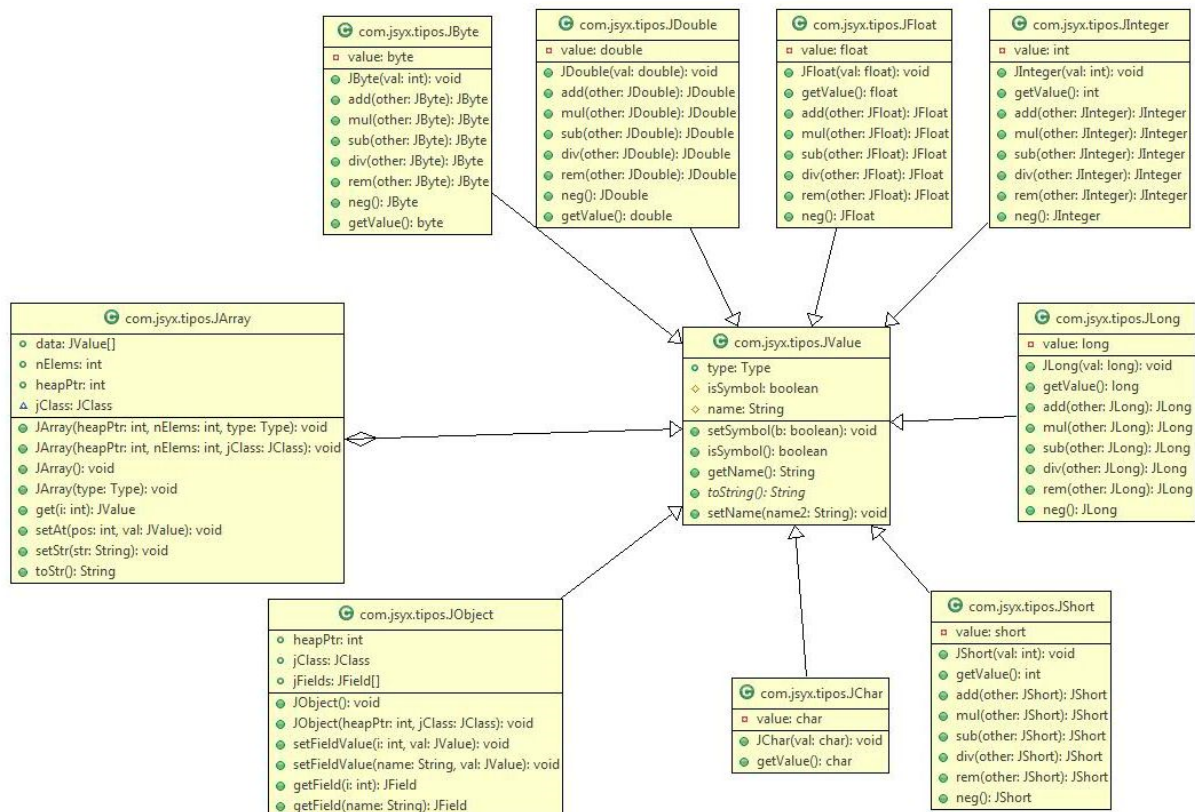


Figura 13. Diagrama de clases con los tipos de jsyX

#### 3.7.3. Listado de instrucciones admitidas

Ver **apéndice A** para una lista completa de las instrucciones admitidas en la ejecución de *jSyX*.

#### 3.7.4. Problemas durante la implementación

A la hora de desarrollar esta parte la mayoría de problemas relacionados con la máquina virtual de *Java* no vinieron de *BCEL*, ya que gracias a esta librería pudimos abstraernos de la intrincada labor de leer y parsear los archivos *.class*, además de darnos muchos accesos directos a la información que necesitábamos para realizarla.

Aunque sí es cierto que para la generación del programa fuera del entorno de *Eclipse* tuvimos que añadir el directorio actual al Repository de *BCEL*, esto al *classpath* que maneja, para que consiguiera encontrar el *.class* correctamente.

Una de las partes más conflictivas fue realizar todo el Heap y objetos, para ello tuvimos que leer bastante y buscar ejemplos de cómo otra gente lo había implementado para adaptarlo a nuestra forma.

También fue problemático para que la ejecución aceptara propiedades *static*, y su inicialización a nivel de clase, la clave fue el método *<clinit>*, que complementa al *<init>* cuando es necesaria la inicialización de dichas propiedades. Y no solo *static*, también cuando se realiza una inicialización a nivel de clase esta es realizada en este método especial.

Para permitir que ejecutara métodos nativos no fue tan difícil, aunque sí que ejecute todos los métodos nativos que hacen falta para que funciones cosas como las excepciones, cuanto más bajas en la jerarquía más complicado se vuelve así que en nuestra ejecución solo se trata *Throwable*, que se encuentra en el pico de la jerarquía.

En cuanto a excepciones la instrucción base *athrow* fue una de las más complejas, sobre todo para permitir que la excepción fuera relanzada sino se encontraba un *handler* correcto.

[16]

### 4. Creando la ejecución simbólica

**E**n este apartado hablaremos sobre la adaptación de la máquina virtual de *Java* que desarrollamos previamente para permitir la ejecución simbólica con restricciones.

Primero daremos algunas informaciones generales sobre programación simbólica y programación con restricciones, aportando explicaciones sobre la librería *Choco*, que nos permitirá realizar la parte de restricciones, hablando finalmente de la implementación de la adaptación de nuestra *JVM* a la ejecución simbólica.



### 4.1. Programación Simbólica

La ejecución simbólica es una forma de ejecución en la cual en vez de suministrar entradas normales al programa, se le suministran símbolos representando valores arbitrarios.

Este tipo de ejecución es una extensión de la ejecución normal [1], en donde los símbolos en vez de tomar valores obtendrán expresiones simbólicas. Dichas expresiones simbólicas están basadas en una conjunción de restricciones de las que hablaremos más adelante. La ejecución seguirá el camino normal hasta encontrarse con alguna instrucción en la cual se ve afectado símbolo donde se le asignará alguna de dichas restricciones variando el flujo de la ejecución. Estas restricciones son debido a que hay que considerar cada una de las decisiones en las que el símbolo puede tomar un rango valor u otro. Cada una de estas decisiones supondrán cada una de las nuevas ramas del árbol y la ejecución pasará a ser indeterminista.

Para explorar cada uno de los escenarios crearemos un árbol de ejecución definido por cada una de las rutas de ejecución, es decir, una secuencia de declaraciones a ejecutar. Cada una de estas rutas serán independiente [17]. Una vez alcanzada una de la hojas del árbol, los símbolos estarán formados por un conjunto de restricciones que nos devolverán un rango donde es factible ejecutar dicha entrada para una ruta concreta, en caso contrario dicha ruta no podrá ser alcanzada durante la ejecución normal.

Dado que considerar todos los casos de prueba en ocasiones se torna difícil a ojos del programador de software donde los programas aumentan en complejidad, comprobando todos los posibles escenarios resulta una herramienta eficaz para *testing*, debido a que la ejecución simbólica generará automáticamente cada uno de los escenarios posibles, así como un rango de valores para las entradas.

A raíz de los diversos comportamientos de un programa en el que dichos comportamientos pueden ser infinitos y para limitar recursos, nuestra ejecución no será del todo simbólica ya que propondremos un límite de profundidad del árbol en cual el usuario podrá prefijar antes de la ejecución.

### 4.2. Programación con restricciones

La programación con restricciones es una aproximación alternativa a la programación en donde el proceso es limitado a la generación de unos requisitos(*constraints*) y una solución a dicho requisitos en términos generales o en un dominio específico [18].

Estas requisitos serán cada una de las decisiones que evaluará la máquina virtual para realizar una instrucción que varíe la ejecución del programa, como por ejemplo una evaluación de salto sobre un símbolo o la asignación de un valor a un símbolo.

Podemos separar la características de la programación con restricciones:

- Modelo(*model*): Generación de una representación del problema en términos de *constraints*.

- Resolutor(*solver*): Resolución del modelo según la representación de las *constraints*.

Como veremos más adelante Choco nos permitirá tener dos clases diferenciadas para cada una de estas tareas. Cada *constraint* caracterizará las entradas expresadas mediante una formula asignada por el *model*. Un resolutor de restricciones o *solver* es un teorema automático que devolverá, si es factible, un rango de valores donde los *constraints* sean posibles.

Cada una de las ramas de la ejecución contendrá su propio número de restricciones, como ya mencionamos anteriormente cada rama es independiente.

### 4.3. Choco

Choco es una librería java para programación con restricciones. Está construido sobre un mecanismo basado en eventos con estructuras que aceptan *backtracking* [19].

Choco hace una clara separación del modelado y el resolutor. La Figura 14 representa la arquitectura general de choco.

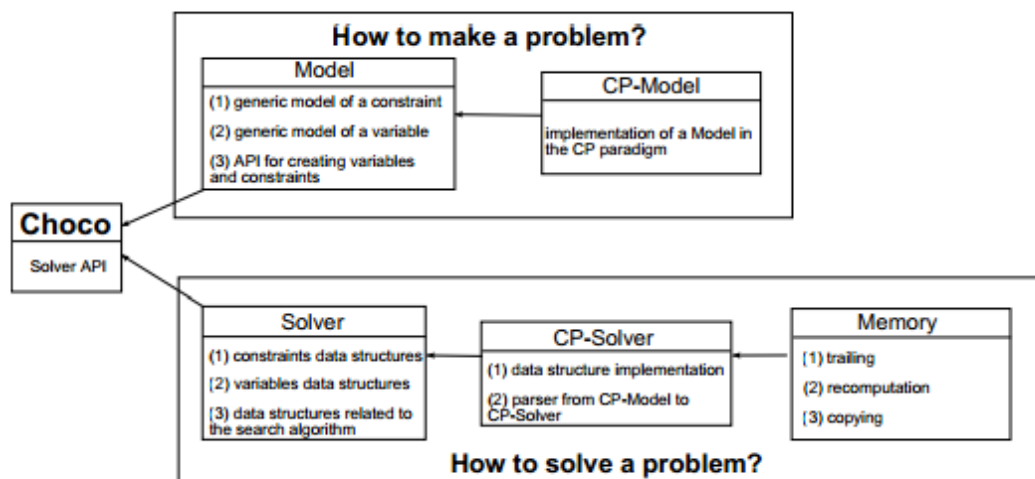


Figura 14: Arquitectura general de Choco. Existen dos partes claramente diferenciadas, una para el Modelado y otra para el Resolutor.

El modelo nos permite describir el problema de una manera sencilla, en ella guardaremos las variables con sus respectivas restricciones.

Una variable es definida por un nombre, un tipo y un rango de valores. El rango de valores será clave en el rendimiento por lo que usaremos rangos acotados para enteros. Cada símbolo ira asociado a una variable del *model* mediante su nombre, y a ella se le añadirán las consiguientes *constraints* que encuentre durante la ejecución.

### 4.3.1. Model

Choco nos provee de las *constraint* necesarias para nuestro programa, estas son las restricciones binarias tales como igualdad, o mayor y menor que. Estas *constraints* encuentran su homólogo en las instrucciones de la maquina virtual.

### 4.3.2. Solver

El resolutor se encargara de la resolución: leyendo el modelo, estableciendo la estrategia de búsqueda y los métodos de resolución. En nuestro caso la estrategia de búsqueda será primero en profundidad.

El *solver* será llamado cada vez que lleguemos a una hoja del árbol de ejecución, y restaurado si se quiere obtener la siguiente rama. El modelo sin embargo no necesita ser restaurado, es suficiente con eliminar las *constraint* independientes de la rama actual.

## 4.4. Implementación

Para poder llevar a cabo la ejecución simbólica y el árbol de ejecución correspondiente hemos implementado el siguiente algoritmo escrito en una aproximación a pseudocódigo para una mayor compresión:

Código:

```
private void run_bt(int nivel, int últimaConstraint, bool
                    últimaDecision){
    while (esTopFrame?() && topFrameError?() && nivel < límite) {
        if (esNativo?()){
            ejecutarNativo();
        }
        else {
            ih = conseguirSiguienteInstruccion();
            if (nuevaRama()){

                if (ultimaDecisionTomada()){
                    //Rama Contraria
                    ejecutarInstruccion(ih,false);
                    últimaDecision != últimaDecisión;
                }
                else {
                    //Copia parámetros
                    copiaParentFrame();
                    copiaTopFram();
                    copiaFrame();

                    últimaConstraint = marcarUltimaConstraint();
                    ejecutarInstruccion(ih,false);

                    if (!esHoja?()){
                        run_bt(nivel++, últimaConstraint,
                              últimaDecision);
                    }
                }
            }
        }
    }
}
```

```

        if (!esLimite()){
            resetSolver();
            eliminarConstraintsAcumuladas();
        }
        //Restablecer estado
        lastDecission = !lastDecission;
        topFrame = parentFrame;
        heap = parentHeap;
    }

    }
    else {
        //Si no se produce nueva rama se ejecuta normal
        ejecutarIH(ih, false);
    }
}
if (esHoja?()){
    solucionarSolver();
    guardarSolución();
}
else {
    //Alcanzado el limite de búsqueda
    imprimirErrorMaxNivel();
}
}
}

```

El algoritmo proporciona un árbol de ejecución completo hasta el nivel de profundidad del árbol prefijado con anterioridad como parámetro de entrada. El bucle *while* se ejecutará hasta que no existan más instrucciones en el Frame, se encuentre el límite o un error durante la ejecución.

El siguiente paso es comprobar si la siguiente instrucción puede variar el árbol de ejecución simbólica con una nueva rama. Estas instrucciones serán las instrucciones de *branch*, *getField* y *putField* que crearán un cambio en el flujo de instrucciones del programa. Si no fuese una de estas instrucciones o no involucrase a un símbolo, la ejecución procedería con normalidad. Una vez sabido que estamos en una nueva rama queda comprobar si ya hemos recorrido o no dicha rama. Si ya la hubiésemos visitado recorreríamos la rama contraria. Sino la hemos visitado se efectúa una copia de todos los parámetros relevantes a la hora de hacer el *backtraking*, así como se marca el último *constraint* para poder eliminar posteriormente al hacer *backtraking* todas las *constraint* hasta la última afectada por esa rama. Si no es nodo hoja continuaríamos bajando en profundidad en el árbol. Una vez llegado al nodo hoja resolvemos el *solver* y guardaríamos la solución o bien si hemos llegado al límite de profundidad mostraríamos un error.

Al volver del *backtraking* necesitaremos restaurar el estado anterior y resetear el *solver* para poder volver a ejecutarlo en la siguiente solución.

A parte de este algoritmo las clases del paquete Tipos han sido modificada para soportar símbolos con sus respectivos rangos de valores para cada uno de ellos y nombres con los que identificar las variables en el *model* y *solver*. En cada una de las clases del paquete Tipo los

métodos encargados de modificar los JValues han sido convenientemente adaptados a la ejecución simbólica añadiendo el constraint asociado a dicha operación de modificación. No entramos en detalle de cada uno de los métodos debido a que cada uno es independiente y no siguen un convenio. La clase *JVM* donde se encuentran las instrucciones añade funcionalidad simbólica a cada una de ellas donde se ve modificado algún símbolo, por el mismo motivo anteriormente mencionado no evaluaremos como ha sido implementada cada una de las instrucciones para la ejecución simbólica.

### 4.4.1. Problemas durante la implementación

Uno de los problemas que nos encontramos, fue la cantidad de diferentes instrucciones y tipos que existen dentro de la máquina virtual, donde cada uno de ellos supone una implementación diferente.

Para adaptar los objetos a simbólica seguimos un procedimiento similar para los valores primitivos, el mayor problema fue que cada propiedad de un objeto es un símbolo, además de los valores *static* de clase, así que tuvimos que convertirlos en símbolos tras su creación, añadiéndolos al modelo de *Choco*.

Para la ejecución de las instrucciones *getfield* and *putfield*, reusamos el código para el lanzamiento de excepciones (Throwable en nuestro caso), para el caso de que el objeto sea nulo, así que no nos dieron demasiados problemas.

Dentro de los *arrays* nos hemos encontrado con bastantes problemas, ya que choco no recogía las restricciones necesarias para ellos debido a la incertidumbre que supone no saber la longitud de un *array*, por lo que decidimos simplificar la ejecución haciendo que el usuario estableciese una longitud por defecto.

## 4.5. Ejemplo

**A**nalizaremos la ejecución simbólica utilizando sobre tipos primitivos como puede ser *int*. con el siguiente ejemplo:

Código *Java*:

```
public int foo(int x, int y {  
    if (x > 5) {  
        int temp = y;  
        return y;  
    }  
    else if (x == 0) {  
        x = 5;  
        return y;  
    }  
    else return y;  
}
```

Código *bytecode*:

```
0  iload_1  
1  iconst_5  
2  if_icmple 9 (+7)
```

```

5  iload_2
6  istore_3
7  iload_2
8  ireturn
9  iload_1
10 ifne 17 (+7)
13 iconst_5
14 istore_1
15 iload_1
16 ireturn
17 iload_2
18 ireturn

```

En este caso la entrada tiene dos tipos enteros. Esta entrada será sustituida por dos símbolo dentro del rango de enteros, a 'x' se le asignará el nombre 'A' y a 'y' 'B'. La ejecución seguirá su curso normal hasta llegar al primer salto(2 *if\_icmple* 9).

Llegados al primer salto existe la decisión de tomar o no dicho salto según el valor del símbolo, como dicho valor no es conocido daremos por correcta una de las decisiones. Estas decisiones son las *constraints* que irán enlazadas con su correspondiente *Variable*(Choco) que concuerda con el nombre del símbolo.

Las *constraints* las iremos almacenando en el *model* del *ConstraintSolver* según el camino que escojamos. En nuestro programa siempre coge la opción de salto como TRUE en primera instancia, así los IFs se tomarán como falsos y las condiciones en los *loops* como verdaderas dada la implementación de la maquina virtual.

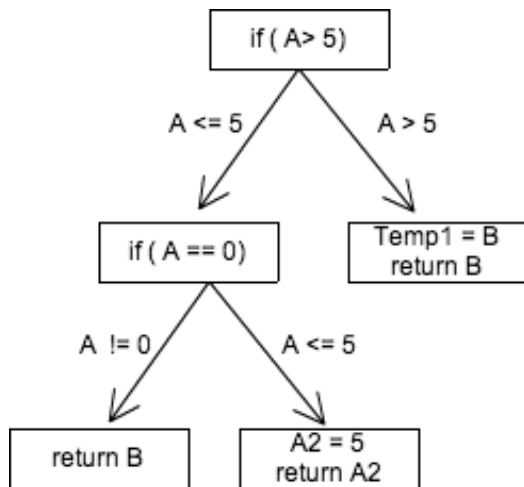


Figura 15: Árbol de ejecución foo con nomenclatura de símbolos.

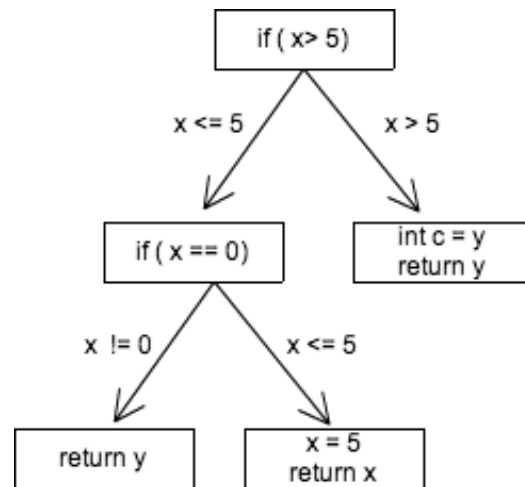


Figura 16 : Árbol de ejecución foo.

En cada decisión que tomemos se creará una nueva rama en el árbol así el primer camino que cogerá será :  $x \leq 5$ . Volveríamos a obtener una nueva *constraint* en un nuevo salto(10 ifne). Esta restricción será añadida al modelo como una conjunción de todas las restricciones anteriores.

Así al finalizar la ejecución del método *foo* obtendremos que  $x \leq 5$  y  $x \neq 0$  por lo que ejecutando el *solver* obtendremos un rango dentro los enteros donde  $x < 0$ , 'y' al no haber sido modificado tendría el rango de todos los enteros.

Luego se procede mediante *backtracking* eliminando las *constraints* acumuladas obtenidas en la rama recorrida, y ejecutando la siguiente rama sin explorar del árbol. Este procedimiento se repite hasta llegar al máximo nivel prefijado con anterioridad por el usuario para así poder controlar la profundidad a la que se quiere explorar el árbol, o llegando a todos los posibles caminos sino se alcanza dicho nivel.

La segunda ruta ejecutará la condición de salto como falsa automáticamente. En la ejecución de la instrucción de asignación de 'x' tendremos que obtener una copia del símbolo dado que si no la realizásemos estaríamos añadiendo una nueva *constraint* al modelo que nos daría una resolución errónea en la que  $x = 0$  y  $x \neq 0$ , por lo que es necesario crear dicha copia. Así al acabar la ejecución obtendríamos como valor de entrada  $x \leq 5$  &&  $x \neq 0$  por lo que ejecutando el resolutor obtendríamos  $x = 0$  como entrada y el programa devolvería la copia de x con valor 5.

En la última rama nos encontramos con un nuevo problema asignar a una variable local un símbolo. Ya que los símbolos no tienen valores hasta ejecutar el *solver* no sabremos qué valor asignar a la variable, por lo que tendremos que añadir esta nueva variable como símbolo temporal y añadirla a nuestro modelo. Una vez resuelto el modelo obtendremos un rango para dicha variable al igual que con los datos de entrada y los copias de dichas entradas.

**E**n este apartado hablaremos sobre el cliente para el uso de la aplicación. Primero daremos una perspectiva general del cliente. A continuación daremos información de la librería usada para facilitar tanto el uso del cliente para el usuario como para la elaboración del control de parámetros. Después daremos algunos detalles de la implementación de nuestro cliente. Por último mostraremos ejemplos de ejecuciones de la aplicación realizando distintos usos de los parámetros establecidos.



## 5.1. Información sobre el cliente

Para el desarrollo de la parte cliente decidimos dar acceso tanto a la ejecución no simbólica normal como a la ejecución simbólica. Este comportamiento está encapsulado en la clase `Exec`.

Para facilitar el uso del programa hemos implementado un sistema de paso de parámetros ayudándonos de la librería *Commons CLI* de la que hablaremos a continuación.

La clase `Exec` se encarga de preparar la máquina virtual para la ejecución en base a los parámetros pasados. El comportamiento varía según se elija la ejecución simbólica o no, así como el tipo de parámetros necesarios.

A continuación se muestra una captura de los distintos tipos de parámetros que la aplicación puede recibir, permitiendo versiones cortas y largas de los mismos.

```

c:\Users\Noa\Documents>java -jar jsyx.jar
usage: jsyx
--array-size <SIZE>    define tamaño de arrays para ejecución
                        simbólica (por defecto 5)
-h,--bytecode          imprime el bytecode del .class especificado con
                        -c o del metodo con -m
-c <classname>         nombre completo de clase a usar (*)
-d,--debug             muestra información de ejecución (no por
                        defecto)
-h,--help              imprime este mensaje de ayuda
--heap-size <SIZE>     define tamaño de heap (por defecto 100)
-i,--ihpc              ejecución no simbólica con instruction handler
                        (por defecto)
-l,--log               generación de archivo de log y traza para
                        simbólica con todos los caminos en el arbol de
                        ejecución (no por defecto)
--limit-size <SIZE>    define límite de profundidad de ejecución (por
                        defecto 5)
-m <methodname>        metodo a ejecutar dentro de clase (*) con
                        simbólica)
-n,--normal            ejecución no simbólica
-o,--opcode            ejecución no simbólica con opcode
-s,--symbolic          ejecución simbólica (por defecto)
-u,--user              muestra información al usuario sobre los caminos
                        de éxito (no por defecto)
-v,--version           imprime la versión del programa
* indica parametro obligatorio
C:\Users\Noa\Documents>

```

Figura 17. Mensaje de ayuda para uso de *jsyX*

Como se ve en la Figura 17 en ejecución simbólica es necesario especificar tanto el nombre de clase como el método a ejecutar dentro de la clase, para ejecución no simbólica bastará con especificar la clase, pues será el método *main* el que por defecto será ejecutado. Se debe especificar la ruta hasta el *.class* si no se encuentra en el mismo directorio que la aplicación.

Como se ha comentado antes hemos implementado dos formas de ejecución que hemos denominado *IHPC* y *OPCODE*, aunque la ejecución con *OPCODE* solo está disponible en ejecución no simbólica.

Para ambas ejecuciones se proporciona información conforme se realiza la ejecución, usando el parámetro `-d`. Para la ejecución simbólica se permite la generación de un archivo de log usando el parámetro `-l`, que contendrá información completa de los resultados de la ejecución. En cambio el parámetro `-u` permite obtener información resumida de la ejecución.

Hemos añadido la opción de poder ver directamente el *bytecode* del *.class* sobre el que estemos trabajando mediante el parámetro `-b`. En ejecución normal mostrará una representación de la información completa de la clase. Con ejecución simbólica se mostrará solamente el *bytecode* asociado al método de clase que se indique.

Por último también es posible definir el tamaño del Heap a usar por la aplicación y el tamaño de arrays para ejecución simbólica, ya que en nuestra implementación, estos tienen que ser creados antes de lanzar el método a estudiar.

Mencionar que al ejecutar el programa sin parámetros el mensaje de ayuda será mostrado, y también está disponible ver la versión del programa usando el parámetro `-v`, como se ve en la siguiente imagen.

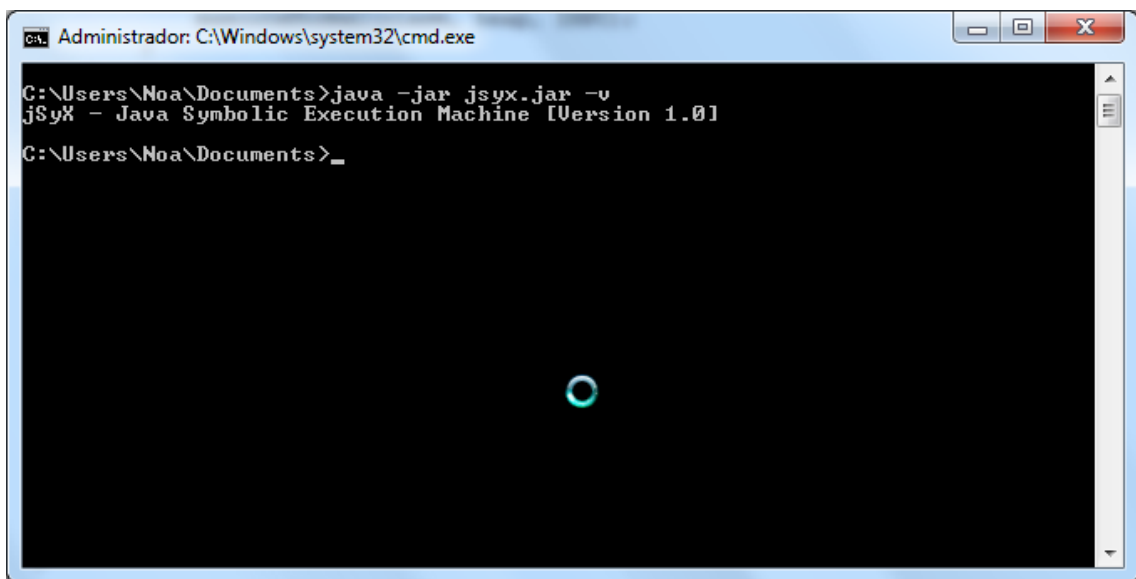


Figura 18. Obtener versión de jSyX usando parámetros

## 5.2. Commons CLI

La librería *Commons CLI* proporciona una API para parsear opciones de línea de comandos pasados a programas. Además permite generar mensajes de ayuda automáticamente a partir de las opciones disponibles, así como ejemplos de uso. [20]

Soporta diferentes tipos de opciones:

- *POSIX* like Options (ie. `tar -zxvf foo.tar.gz`)
- *GNU* like long Options (ie. `du --human-readable --max-depth=1`)
- *Java* like properties (ie. `Java -DJava.awt.headless=true -DJava.net.useSystemProxies=true Foo`)
- Short Options with value attached (ie. `gcc -O2 foo.c`)
- long Options with single hyphen (ie. `ant -projecthelp`)

Hay 3 etapas a la hora de procesar los parámetros de línea de comandos:

- Definición

- *Parseo*
- Interrogatorio

En la primera se definen las distintas opciones que el programa será capaz de reconocer usando la clase `Options`.

Tras definir estas opciones se pasa al proceso de *parseo*, dependiendo de las opciones que hayamos establecido deberemos usar un *parser* u otro (`CommandLineParser`). El resultado de este parseo es un objeto de tipo `CommandLine`.

Con el objeto `CommandLine` obtenido, podemos realizar consultas sobre qué tipo de parámetro se ha usado y acorde a ello realizar distintos tipos de modificaciones en la ejecución.

Para una lista de típicos escenario de uso se puede consultar la siguiente referencia. [21]

### 5.3. Detalles de implementación

Como hemos mencionado en la introducción de este capítulo la clase `Exec` es el punto de partida, hace uso de la mayoría de clases de la aplicación.

Lo primero es controlar el paso de parámetros, con este objetivo se usa la clase `JCmdLine` que encapsula el uso de parámetros facilitando distintos métodos para realizar las etapas mencionadas en la sección 5.2. Según la etapa de interrogatorio se establecen distintos *flags* que variarán el resultado de la ejecución. Si los parámetros no son los correctos o no se reconocen el programa finalizará dando un mensaje de error identificando este hecho.

Después de esto el flujo normal es:

1. Creación de Heap
2. Creación de área de clases
3. Regeneración del `.class` para *BCEL*
4. Creación del `JClass`
5. Creación del `JMethod`
6. Creación de `StackFrame` con método y clase
7. Creación de la *JVM*
8. Ejecución de `<clinit>` si necesario
9. Inicio de ejecución de la *JVM*

Para la ejecución simbólica es necesario realizar la creación de los parámetros del método concreto previamente a su ejecución, esto no es un problema para tipos de datos primitivos, pero para objetos y *arrays* es necesario realizar esta inicialización usando la propia *JVM*. Además se modifican estos parámetros y sus nombres en *Choco* para convertirlos en símbolos.

Para la ejecución no simbólica no es necesaria esta creación pues seguirá el flujo de un programa *Java* normal.

En el siguiente diagrama de clases puede verse la relación básica entre los distintos componentes para el funcionamiento del cliente.

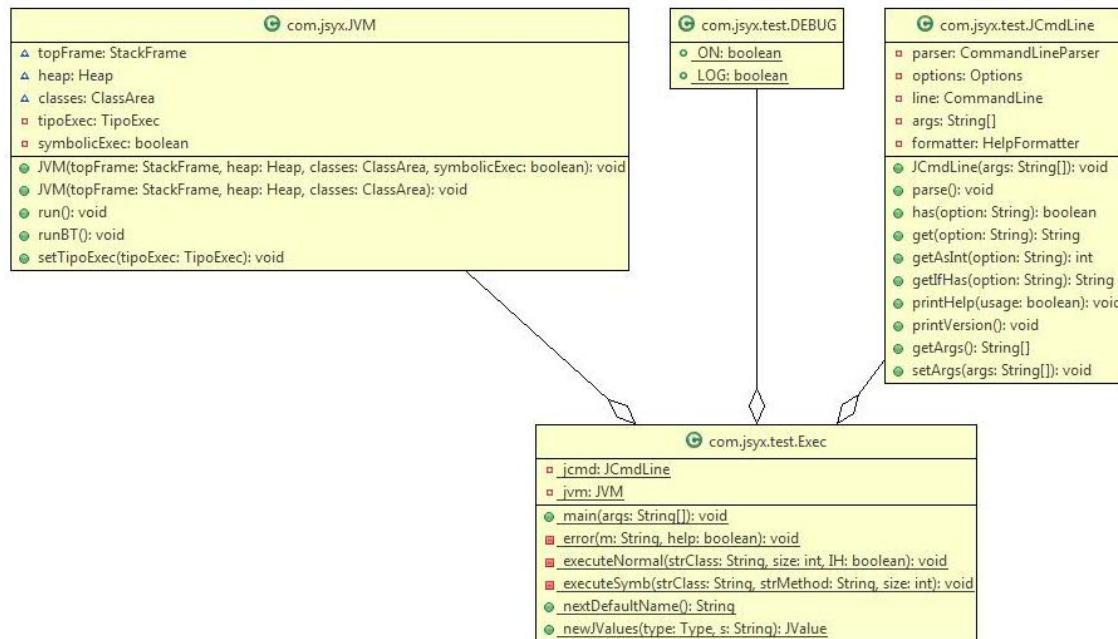


Figura 19. Diagrama de clases para cliente

Exec modifica las constantes de la clase DEBUG y varía a la JVM dependiendo de los resultados obtenidos usando JCmdLine.

## 5.4. Ejemplos de uso

A continuación mostraremos algunos ejemplos de uso de *jSyX*, organizándolos en pruebas con no simbólica y pruebas con simbólica, variando los parámetros usados para proveer distintas opciones.

### 5.4.1. Ejemplos de ejecución no simbólica

Con la ejecución no simbólica como hemos comentado simplemente tenemos que especificar el nombre de la clase (del *.class*) que queremos ejecutar.

#### Ejemplo 1

En la siguiente imagen podemos ver la ejecución de un programa que crea una cadena de texto con el texto “cadena”.

#### CÓDIGO JAVA:

```

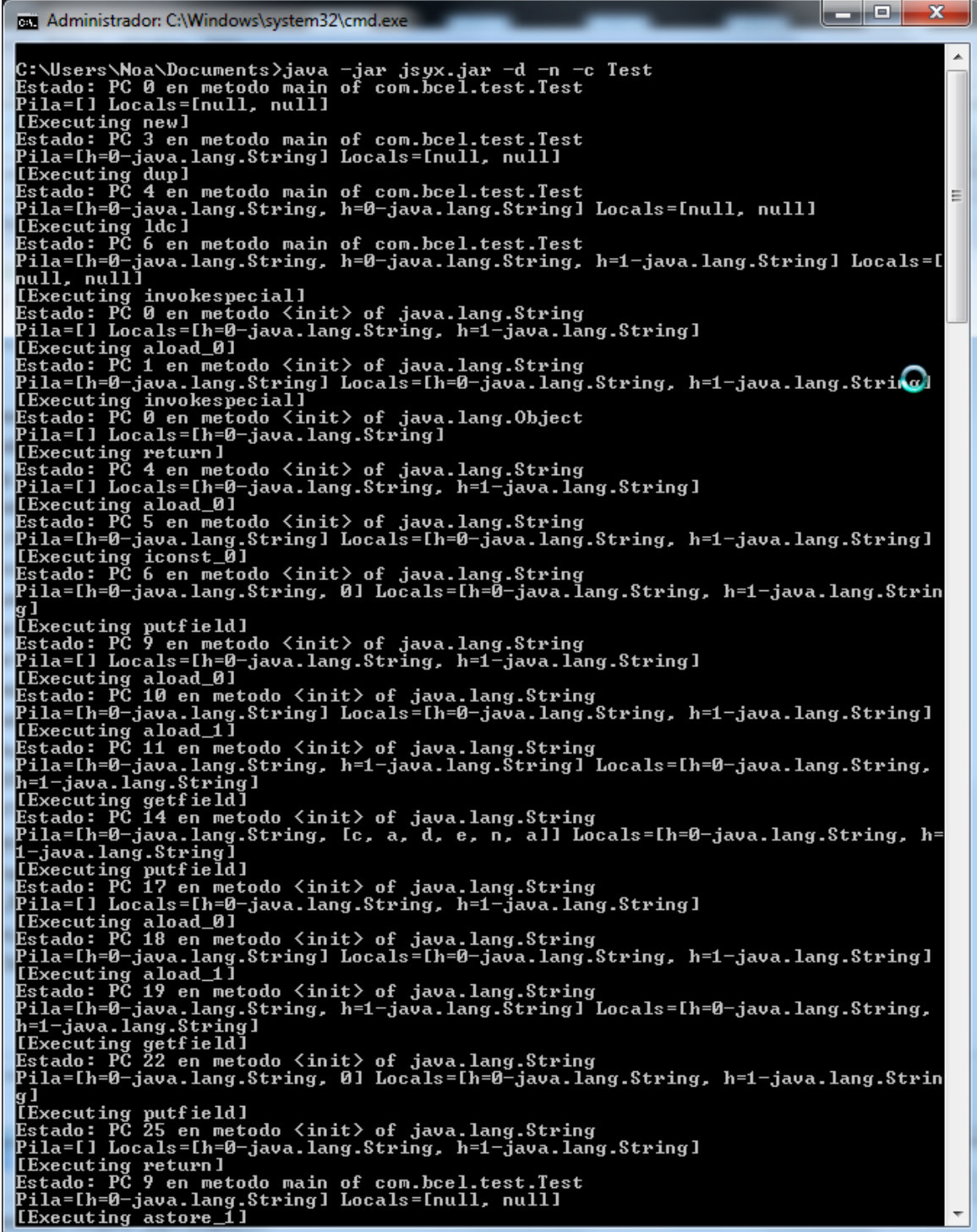
public Class Test {
    public static void main(String[] args) {
        String a = new String("cadena");
    }
}
  
```

#### CÓDIGO BYTECODE:

## 5. Cliente

```
0 new java.lang.String [16]
3 dup
4 ldc <String "cadena"> [18]
6 invokespecial java.lang.String(java.lang.String) [20]
9 astore_1 [a]
10 return
```

### EJECUCIÓN:



```
C:\Users\Noa\Documents>java -jar jsyx.jar -d -n -c Test
Estado: PC 0 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, null]
[Executing new]
Estado: PC 3 en metodo main of com.bcel.test.Test
Pila=[h=0-java.lang.String] Locals=[null, null]
[Executing dup]
Estado: PC 4 en metodo main of com.bcel.test.Test
Pila=[h=0-java.lang.String, h=0-java.lang.String] Locals=[null, null]
[Executing ldc]
Estado: PC 6 en metodo main of com.bcel.test.Test
Pila=[h=0-java.lang.String, h=0-java.lang.String, h=1-java.lang.String] Locals=[null, null]
[Executing invokespecial]
Estado: PC 0 en metodo <init> of java.lang.String
Pila=[] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing aload_0]
Estado: PC 1 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing invokespecial]
Estado: PC 0 en metodo <init> of java.lang.Object
Pila=[] Locals=[h=0-java.lang.String]
[Executing return]
Estado: PC 4 en metodo <init> of java.lang.String
Pila=[] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing aload_0]
Estado: PC 5 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing iconst_0]
Estado: PC 6 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String, 0] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing putfield]
Estado: PC 9 en metodo <init> of java.lang.String
Pila=[] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing aload_0]
Estado: PC 10 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing aload_1]
Estado: PC 11 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String, h=1-java.lang.String] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing getfield]
Estado: PC 14 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String, [c, a, d, e, n, a]] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing putfield]
Estado: PC 17 en metodo <init> of java.lang.String
Pila=[] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing aload_0]
Estado: PC 18 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing aload_1]
Estado: PC 19 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String, h=1-java.lang.String] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing getfield]
Estado: PC 22 en metodo <init> of java.lang.String
Pila=[h=0-java.lang.String, 0] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing putfield]
Estado: PC 25 en metodo <init> of java.lang.String
Pila=[] Locals=[h=0-java.lang.String, h=1-java.lang.String]
[Executing return]
Estado: PC 9 en metodo main of com.bcel.test.Test
Pila=[h=0-java.lang.String] Locals=[null, null]
[Executing astore_1]
```

Figura 20. Prueba de ejecución no simbólica con `-n -d -c`

Como vemos se hace uso de `-n` para ejecución no simbólica, `-d` para mostrar información de ejecución y `-c` para especificar el fichero `.class`. En cada paso de ejecución se muestra el estado de la misma, la instrucción que se está ejecutando, dentro de que método y dentro de que clase. Además el valor de la pila y las variables locales. Como estamos con objetos se muestra su posición en el Heap y el tipo de objeto, en este caso String. Además podemos ver como se pasa desde el método *main* de la clase Test al propio constructor de String *<init>*, tras esa construcción el control pasa de nuevo al Test.

### Ejemplo 2

En este ejemplo vemos la ejecución de un programa que crea un *array* de enteros y rellena la primera posición.

#### CÓDIGO JAVA:

```
public Class Test {
    public static void main(String[] args) {
        int[] array = new int[3];
        if(array.length>=3)
            array[0]=20;
    }
}
```

#### CÓDIGO BYTECODE:

```
0  iconst_3
1  newarray int [10]
3  astore_1 [array]
4  aload_1 [array]
5  arraylength
6  iconst_3
7  if_icmplt 15
10 aload_1 [array]
11 iconst_0
12 bipush 20
14 iastore
15 return
```

#### EJECUCIÓN:

```

C:\Users\Noa\Documents>java -jar jsyx.jar -d -o -n -c Test
Estado PC 0 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, null]
[Executing iconst_3]
Estado PC 1 en metodo main of com.bcel.test.Test
Pila=[3] Locals=[null, null]
[Executing newarray]
Estado PC 3 en metodo main of com.bcel.test.Test
Pila=[[null, null, null]] Locals=[null, null]
[Executing astore_1]
Estado PC 4 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, [null, null, null]]
[Executing aload_1]
Estado PC 5 en metodo main of com.bcel.test.Test
Pila=[[null, null, null]] Locals=[null, [null, null, null]]
[Executing arraylength]
Estado PC 6 en metodo main of com.bcel.test.Test
Pila=[3] Locals=[null, [null, null, null]]
[Executing iconst_3]
Estado PC 7 en metodo main of com.bcel.test.Test
Pila=[3, 3] Locals=[null, [null, null, null]]
[Executing if_icmplt]
Estado PC 10 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, [null, null, null]]
[Executing aload_1]
Estado PC 11 en metodo main of com.bcel.test.Test
Pila=[[null, null, null]] Locals=[null, [null, null, null]]
[Executing iconst_0]
Estado PC 12 en metodo main of com.bcel.test.Test
Pila=[[null, null, null], 0] Locals=[null, [null, null, null]]
[Executing bipush]
Estado PC 14 en metodo main of com.bcel.test.Test
Pila=[[null, null, null], 0, 20] Locals=[null, [null, null, null]]
[Executing iastore]
Estado PC 15 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, [20, null, null]]
[Executing return]

C:\Users\Noa\Documents>

```

Figura 21. Prueba de ejecución no simbólica con `-n -o -d -c`

En este caso la ejecución se realiza usando el parámetro `-o` para que la ejecución funcione con *OPCODE*, la ejecución es la misma que si se realizara sin él, pero por dentro la forma de acceder a las instrucciones es distinta. Podemos comentar algunas particularidades de este código, como el uso de la instrucción `iconst_3` que añade directamente un 3 a la pila, la instrucción `newarray` que crea el *array* de tamaño el valor que exista en la cima de la pila, o `arraylength`, que obtiene directamente el tamaño del *array*.

### Ejemplo 3

En este ejemplo vemos la ejecución de un programa que crea un objeto y hace uso de un método propio para establecer una propiedad del mismo.

#### CÓDIGO JAVA:

```

public Class Test {
    public static void main(String[] args){
        A a = new A(20);
        for(int i = 0; i < 2; i++){
            if(i==1)
                a.set(i);
        }
    }
}

```

#### CÓDIGO BYTECODE:

```

0  new com.bcel.Test.A [16]
3  dup
4  bipush 20
6  invokespecial com.bcel.Test.A(int) [18]
9  astore_1 [a]
10  iconst 0
11  istore_2 [i]
12  goto 28
15  iload_2 [i]
16  iconst 1
17  if_icmpne 25
20  aload_1 [a]
21  iload_2 [i]
22  invokevirtual com.bcel.Test.A.set(int) : void [21]
25  iinc 2 1 [i]
28  iload_2 [i]
29  iconst 2
30  if_icmplt 15
33  return

```

#### EJECUCIÓN:

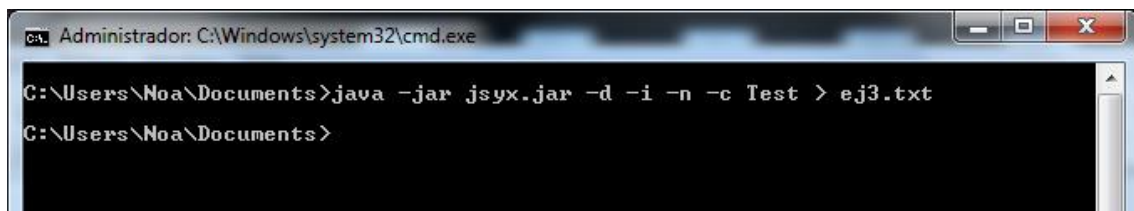


Figura 22. Prueba de ejecución no simbólica con `-n -i -d -c` y redirección

En esta ocasión hacemos la ejecución usando explícitamente el parámetro `-i` para hacer uso de *IHPC*, además redireccionamos la salida del programa para generar un archivo de texto con la ejecución. De esta manera podemos observar cómo ha sido la ejecución revisando este archivo.

En este ejemplo hacemos uso de la clase para pruebas de objetos que mencionamos anteriormente, simplemente contiene un entero y modificamos ese valor haciendo uso de un método de la clase, de ahí el uso de la instrucción *invokevirtual*.

Revisando el archivo `ej3.txt` podemos ver como se realiza esta operación en términos de la máquina virtual. Se realiza la llamada al método teniendo en la pila el objeto que lo llama y los parámetros que toma, en este caso solo el entero 1, estos se convierten en variables locales del método *set* que modifica el objeto usando la instrucción *putfield*.

```

Estado: PC 22 en metodo main of com.bcel.Test.Test
Pila=[h=0-com.bcel.Test.A, 1] Locals=[null, h=0-com.bcel.Test.A, 1]
[Executing invokevirtual]
Estado: PC 0 en metodo set of com.bcel.Test.A
Pila=[] Locals=[h=0-com.bcel.Test.A, 1]
[Executing aload_0]
Estado: PC 1 en metodo set of com.bcel.Test.A
Pila=[h=0-com.bcel.Test.A] Locals=[h=0-com.bcel.Test.A, 1]
[Executing iload_1]
Estado: PC 2 en metodo set of com.bcel.Test.A
Pila=[h=0-com.bcel.Test.A, 1] Locals=[h=0-com.bcel.Test.A, 1]
[Executing putfield]
Estado: PC 5 en metodo set of com.bcel.Test.A
Pila=[] Locals=[h=0-com.bcel.Test.A, 1]

```



[Executing return]

#### Ejemplo 4

En esta ocasión probamos los métodos nativos implementados.

#### CÓDIGO JAVA:

```
public Class Test {
    public static void main(String[] args) {
        // metodos nativos
        print("-----print nativo-----");
        println("-----println nativo-----");
        int a = getIntFromStr("2");
    }

    public static native void print(String str);
    public static native void println(String str);
    public static native int getIntFromStr(String str);
}
```

#### CÓDIGO BYTECODE:

```
0  ldc <String "-----print nativo-----"> [16]
2  invokestatic com.bcel.Test.Test.print(Java.lang.String) : void [18]
5  ldc <String "-----println nativo-----"> [22]
7  invokestatic com.bcel.Test.Test.println(Java.lang.String) : void
[24]
10 ldc <String "2"> [27]
12 invokestatic com.bcel.Test.Test.getIntFromStr(Java.lang.String) :
int [29]
15 istore_1 [a]
16 return
```

#### EJECUCIÓN:

```
Administrador: C:\Windows\system32\cmd.exe

C:\Users\Noa\Documents>java -jar jsyx.jar -d -n -c Test
Estado: PC 0 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, null]
[Executing ldc]
Estado: PC 2 en metodo main of com.bcel.test.Test
Pila=[h=0-java.lang.String] Locals=[null, null]
[Executing invokestatic]
Estado: PC 0 en metodo print of com.bcel.test.Test
Pila=[] Locals=[h=0-java.lang.String]
-----print nativo-----Estado: PC 5 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, null]
[Executing ldc]
Estado: PC 7 en metodo main of com.bcel.test.Test
Pila=[h=1-java.lang.String] Locals=[null, null]
[Executing invokestatic]
Estado: PC 0 en metodo println of com.bcel.test.Test
Pila=[] Locals=[h=1-java.lang.String]
-----println nativo-----
Estado: PC 10 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, null]
[Executing ldc]
Estado: PC 12 en metodo main of com.bcel.test.Test
Pila=[h=2-java.lang.String] Locals=[null, null]
[Executing invokestatic]
Estado: PC 0 en metodo getIntFromStr of com.bcel.test.Test
Pila=[] Locals=[h=2-java.lang.String]
Estado: PC 15 en metodo main of com.bcel.test.Test
Pila=[2] Locals=[null, null]
[Executing istore_1]
Estado: PC 16 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, 2]
[Executing return]
C:\Users\Noa\Documents>
```

Figura 23. Prueba de ejecución no simbólica con `-d -n -c` de métodos nativos

Como vemos los métodos están declarados como *native*, como se ha explicado anteriormente esto hará que la ejecución se realice a través de la clase Natives que contiene la implementación de estos métodos, la JVM se “desentiende” de esta ejecución, esperando obtener lo que se requiera de cada método.

Como vemos el uso de *ldc* se realiza para cargar un String en la pila del que hacen uso los 3 métodos, observar como el objeto String “2” es eliminado de las variables locales y convertido en un 2 entero que se coloca en la cima de la pila.

Podemos ver la implementación de estos métodos en la clase Natives.

```
public static JValue println(JVM jvm) {
    JObject ostr = (JObject) JVM.topFrame.locals[0];
    JArray arraystr = (JArray) ostr.getField(0).getValue();
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < arraystr.nElems; i++) {
        sb.append(arraystr.get(i));
    }
    System.out.println(sb);
    return null;
}

public static JValue getIntFromStr(JVM jvm) {
    JObject ostr = (JObject) JVM.topFrame.locals[0];
    JArray arraystr = (JArray) ostr.getField(0).getValue();
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < arraystr.nElems; i++) {
        sb.append(arraystr.get(i));
    }
    return new JInteger(new Integer(sb.toString()));
}
```

Como vemos en ambos casos se obtiene el objeto String de las variables locales y después obtenemos el JField que contiene la cadena de texto en sí, estas cadenas las hemos implementado como un JArray de JChar, generamos un StringBuffer con su contenido y actuamos en consecuencia, imprimiendo el valor o creando un valor entero. Este valor es devuelto a la JVM que lo añadirá en la pila dependiendo de si hay un valor o no (*null* o no), para que pueda ser usado por el resto del programa.

### Ejemplo 5

Probaremos ahora el acceso a propiedades *static* de clase.

#### CÓDIGO JAVA:

```
public class Test {
    public static void main(String[] args) {
        A a = new A(5);
        a.valStatic=a.get();
        int b=a.valStatic;
    }
}
```

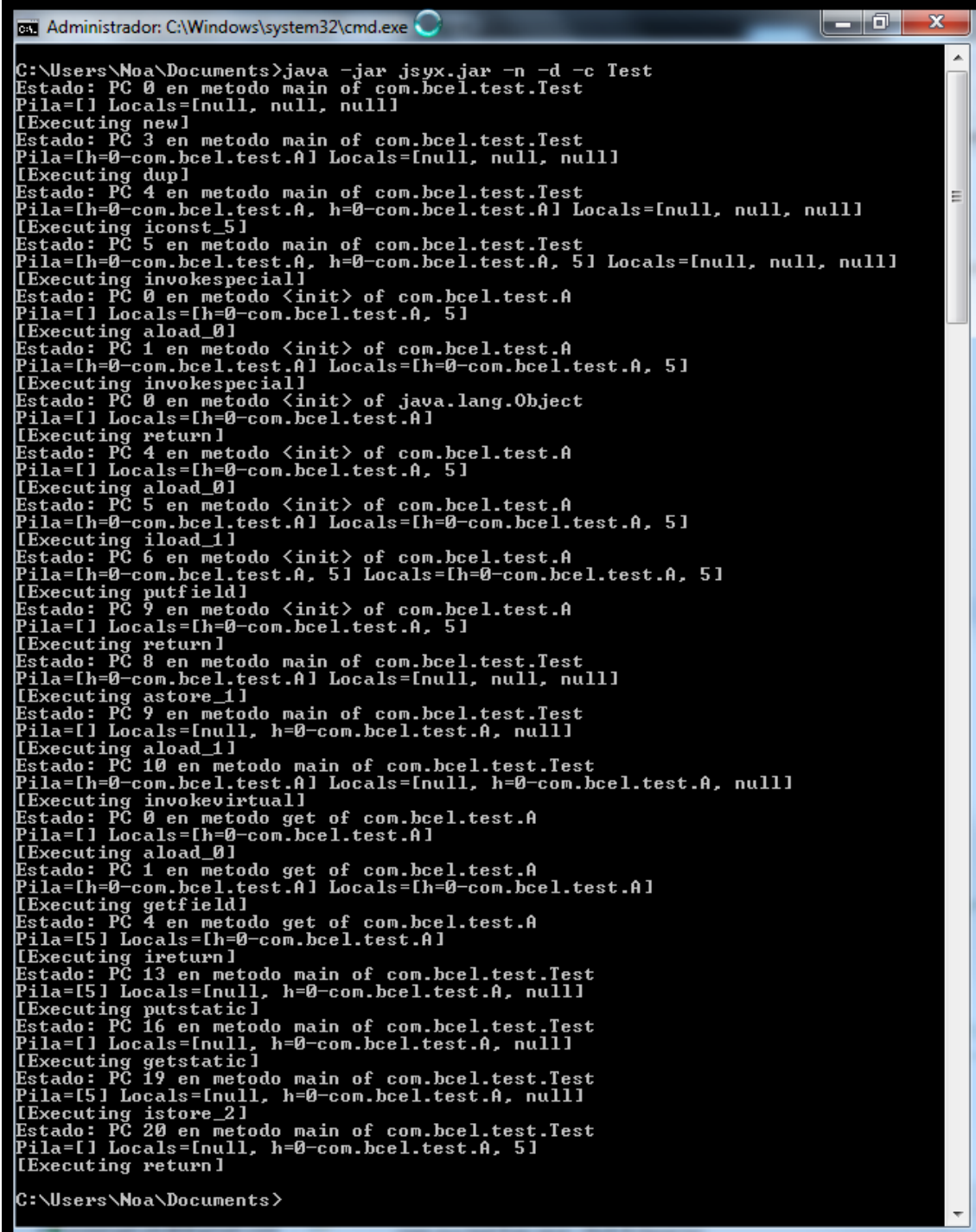
#### CÓDIGO BYTECODE:

```
0  new com.bcel.test.A [16]
3  dup
4  iconst_5
```

```

5  invokespecial com.bcel.test.A(int) [18]
8  astore_1 [a]
9  aload_1 [a]
10 invokevirtual com.bcel.test.A.get() : int [21]
13 putstatic com.bcel.test.A.valStatic : int [25]
16 getstatic com.bcel.test.A.valStatic : int [25]
19 istore_2 [b]
20 return

```

**EJECUCIÓN:**


```

C:\Users\Noa\Documents>java -jar jsyx.jar -n -d -c Test
Estado: PC 0 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, null, null]
[Executing new]
Estado: PC 3 en metodo main of com.bcel.test.Test
Pila=[h=0-com.bcel.test.A] Locals=[null, null, null]
[Executing dup]
Estado: PC 4 en metodo main of com.bcel.test.Test
Pila=[h=0-com.bcel.test.A, h=0-com.bcel.test.A] Locals=[null, null, null]
[Executing iconst_5]
Estado: PC 5 en metodo main of com.bcel.test.Test
Pila=[h=0-com.bcel.test.A, h=0-com.bcel.test.A, 5] Locals=[null, null, null]
[Executing invokespecial]
Estado: PC 0 en metodo <init> of com.bcel.test.A
Pila=[] Locals=[h=0-com.bcel.test.A, 5]
[Executing aload_0]
Estado: PC 1 en metodo <init> of com.bcel.test.A
Pila=[h=0-com.bcel.test.A] Locals=[h=0-com.bcel.test.A, 5]
[Executing invokespecial]
Estado: PC 0 en metodo <init> of java.lang.Object
Pila=[] Locals=[h=0-com.bcel.test.A]
[Executing return]
Estado: PC 4 en metodo <init> of com.bcel.test.A
Pila=[] Locals=[h=0-com.bcel.test.A, 5]
[Executing aload_0]
Estado: PC 5 en metodo <init> of com.bcel.test.A
Pila=[h=0-com.bcel.test.A] Locals=[h=0-com.bcel.test.A, 5]
[Executing iload_1]
Estado: PC 6 en metodo <init> of com.bcel.test.A
Pila=[h=0-com.bcel.test.A, 5] Locals=[h=0-com.bcel.test.A, 5]
[Executing putfield]
Estado: PC 9 en metodo <init> of com.bcel.test.A
Pila=[] Locals=[h=0-com.bcel.test.A, 5]
[Executing return]
Estado: PC 8 en metodo main of com.bcel.test.Test
Pila=[h=0-com.bcel.test.A] Locals=[null, null, null]
[Executing astore_1]
Estado: PC 9 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, h=0-com.bcel.test.A, null]
[Executing aload_1]
Estado: PC 10 en metodo main of com.bcel.test.Test
Pila=[h=0-com.bcel.test.A] Locals=[null, h=0-com.bcel.test.A, null]
[Executing invokevirtual]
Estado: PC 0 en metodo get of com.bcel.test.A
Pila=[] Locals=[h=0-com.bcel.test.A]
[Executing aload_0]
Estado: PC 1 en metodo get of com.bcel.test.A
Pila=[h=0-com.bcel.test.A] Locals=[h=0-com.bcel.test.A]
[Executing getfield]
Estado: PC 4 en metodo get of com.bcel.test.A
Pila=[5] Locals=[h=0-com.bcel.test.A]
[Executing ireturn]
Estado: PC 13 en metodo main of com.bcel.test.Test
Pila=[5] Locals=[null, h=0-com.bcel.test.A, null]
[Executing putstatic]
Estado: PC 16 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, h=0-com.bcel.test.A, null]
[Executing getstatic]
Estado: PC 19 en metodo main of com.bcel.test.Test
Pila=[5] Locals=[null, h=0-com.bcel.test.A, null]
[Executing istore_2]
Estado: PC 20 en metodo main of com.bcel.test.Test
Pila=[] Locals=[null, h=0-com.bcel.test.A, 5]
[Executing return]
C:\Users\Noa\Documents>

```

Figura 24. Prueba de ejecución no simbólica con `-n -d -c` sobre statics

Reutilizamos nuestra clase objeto A de prueba, asignándole una propiedad *static* pública. En este caso podemos ver como se hace uso de la instrucción *getstatic* y *putstatic* para trabajar directamente sobre propiedades de clase. Al contrario que con *getfield* y *putfield* no se encuentra ningún objeto en la pila a la hora de hacer la llamada a dichas instrucciones pues como sabemos no dependen de él. Simplemente se pone el valor y es asignado directamente al *field* de la clase.

### Ejemplo 6

Mostraremos como utilizar la aplicación para obtener una descripción del *.class* de una clase.

#### CÓDIGO JAVA:

```
public class A {
    private int campo1;
    public static int campo2;

    public A(int a, int b) {
        campo1 = a;
        campo2 = b;
    }

    public int getCampo1() {
        return campo1;
    }

    public void setCampo1(int a) {
        campo1 = a;
    }

    public void operacion() {
        for (int i = 0; i < 10; i++) {
            if (campo1 < 2)
                campo2++;
        }
    }

    public static native void nativo1();
    public static native void nativo2();
}
```

#### EJECUCIÓN:

```

C:\Users\Noa\Documents>java -jar jsyx.jar -n -b -c A
-----Resumen de clase-----
public class com.bcel.test.A extends java.lang.Object
filename                A
compiled from            A.java
compiler version         50.0
access flags             33
constant pool            35 entries
ACC_SUPER flag          true

Attribute(s):
    SourceFile(A.java)

2 fields:
    private int campo1
    public static int campo2

6 methods:
    public void <init>(int a, int b)
    public int getCampo1()
    public void setCampo1(int a)
    public void operacion()
    public static native void nativo1()
    public static native void nativo2()

-----Constant Pool-----
1>CONSTANT_Class[7](name_index = 2)
2>CONSTANT_Utf8[1](<"com/bcel/test/A">)
3>CONSTANT_Class[7](name_index = 4)
4>CONSTANT_Utf8[1](<"java/lang/Object">)
5>CONSTANT_Utf8[1](<"campo1">)
6>CONSTANT_Utf8[1](<"I">)
7>CONSTANT_Utf8[1](<"campo2">)
8>CONSTANT_Utf8[1](<"<init>">)
9>CONSTANT_Utf8[1](<"(II)V">)
10>CONSTANT_Utf8[1](<"Code">)
11>CONSTANT_Methodref[10](class_index = 3, name_and_type_index = 12)
12>CONSTANT_NameAndType[12](name_index = 8, signature_index = 13)
13>CONSTANT_Utf8[1](<"()U">)
14>CONSTANT_Fieldref[9](class_index = 1, name_and_type_index = 15)
15>CONSTANT_NameAndType[12](name_index = 5, signature_index = 6)
16>CONSTANT_Fieldref[9](class_index = 1, name_and_type_index = 17)
17>CONSTANT_NameAndType[12](name_index = 7, signature_index = 6)
18>CONSTANT_Utf8[1](<"LineNumberTable">)
19>CONSTANT_Utf8[1](<"LocalVariableTable">)
20>CONSTANT_Utf8[1](<"this">)
21>CONSTANT_Utf8[1](<"Lcom/bcel/test/A;">)
22>CONSTANT_Utf8[1](<"a">)
23>CONSTANT_Utf8[1](<"b">)
24>CONSTANT_Utf8[1](<"getCampo1">)
25>CONSTANT_Utf8[1](<"()I">)
26>CONSTANT_Utf8[1](<"setCampo1">)
27>CONSTANT_Utf8[1](<"(I)V">)
28>CONSTANT_Utf8[1](<"operacion">)
29>CONSTANT_Utf8[1](<"i">)
30>CONSTANT_Utf8[1](<"StackMapTable">)
31>CONSTANT_Utf8[1](<"nativo1">)
32>CONSTANT_Utf8[1](<"nativo2">)
33>CONSTANT_Utf8[1](<"SourceFile">)
34>CONSTANT_Utf8[1](<"A.java">)

-----Metodos de clase-----
public void <init>(int a, int b)
Signature: (II)V
Code(max_stack = 2, max_locals = 3, code_length = 14)
0:      aload_0
1:      invokespecial    java.lang.Object.<init> ()V <11>

```

Figura 25. Prueba de ejecución no simbólica con `-n -b -c` para visionado de bytecode

Reutilizamos y modificamos nuestra clase de prueba A para usarla como ejemplo. Vemos parte de la salida, primero un resumen de la clase, indicando atributos, campos y métodos de clase, después vemos la *Constant Pool* de la que ya hemos hablado anteriormente, y después una descripción de cada uno de los métodos mostrando sus elementos y el código *bytecode* en sí.

Concretando con la *Constant Pool* podemos ver la cadena de referencias de la que hemos hablado, por ejemplo examinando la entrada 11 que refiere a un método, podemos saber que clase lo contiene en el índice 3 que a su vez hace referencia al nombre con el índice 4

java.lang.Object, siguiendo el otro campo, vamos al índice 12 donde encontramos el nombre en el índice 8, y vemos que se trata del método *<init>*, es decir, el constructor, y encontrar la signatura en el índice 13 (V).

## 5.4.2. Ejemplos de ejecución simbólica

### **Ejemplo 1**

Mostraremos como utilizar la aplicación para obtener una descripción concreta de un método

#### CÓDIGO JAVA:

```
public class A {
    private int campo1;
    public static int campo2;

    public A(int a, int b) {
        campo1 = a;
        campo2 = b;
    }

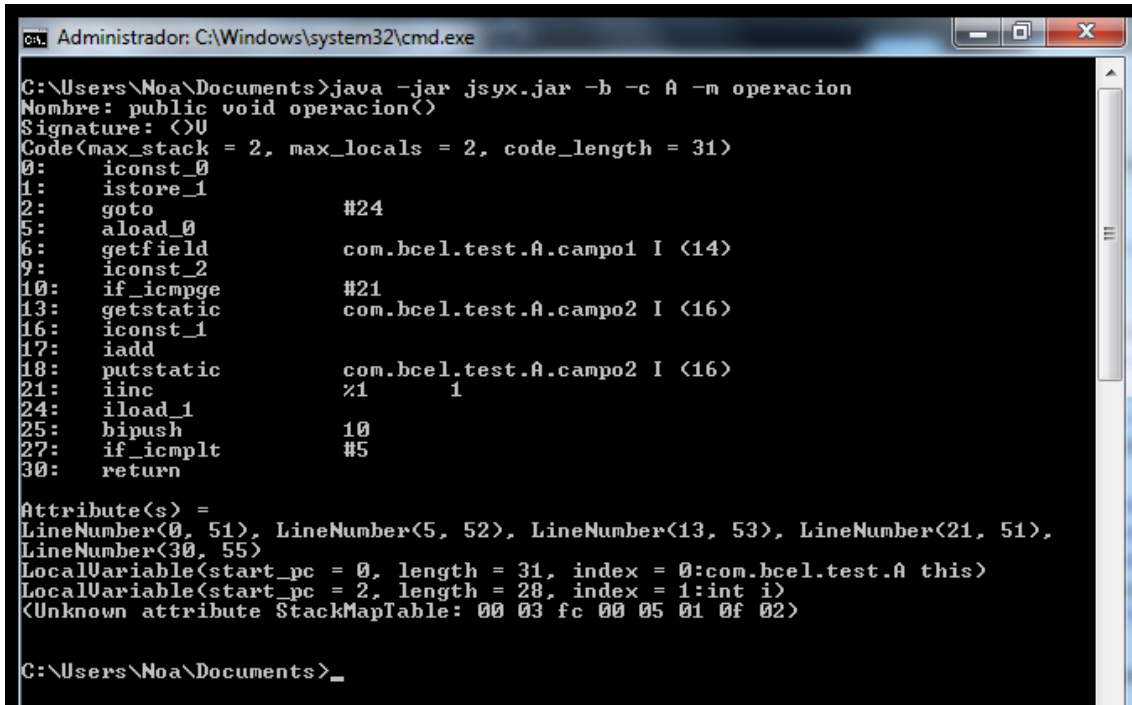
    public int getCampo1() {
        return campo1;
    }

    public void setCampo1(int a) {
        campo1 = a;
    }

    public void operacion() {
        for (int i = 0; i < 10; i++) {
            if (campo1 < 2)
                campo2++;
        }
    }

    public static native void nativo1();
    public static native void nativo2();
}
```

#### EJECUCIÓN:



```

C:\Users\Noa\Documents>java -jar jsyx.jar -b -c A -m operacion
Nombre: public void operacion()
Signature: <>V
Code(max_stack = 2, max_locals = 2, code_length = 31)
0:   iconst_0
1:   istore_1
2:   goto      #24
5:   aload_0
6:   getfield   com.bcel.test.A.campo1 I <14>
9:   iconst_2
10:  if_icmpge   #21
13:  getstatic   com.bcel.test.A.campo2 I <16>
16:  iconst_1
17:  iadd
18:  putstatic   com.bcel.test.A.campo2 I <16>
21:  iinc        #1      1
24:  iload_1
25:  bipush      10
27:  if_icmplt    #5
30:  return

Attribute(s) =
LineNumber(0, 51), LineNumber(5, 52), LineNumber(13, 53), LineNumber(21, 51),
LineNumber(30, 55)
LocalVariable(start_pc = 0, length = 31, index = 0:com.bcel.test.A this)
LocalVariable(start_pc = 2, length = 28, index = 1:int i)
<Unknown attribute StackMapTable: 00 03 fc 00 05 01 0f 02>

C:\Users\Noa\Documents>_

```

Figura 26. Prueba de ejecución simbólica con `-b -c y -m` para visionado de bytecode de método

Reutilizamos la clase del ejemplo anterior, supongamos que queremos ver una descripción del método `operacion`, para ello debemos especificar el parámetro `-m` y no marcar como ejecución no simbólica. Notar las referencias a la *Constant Pool* mediante el símbolo `#`, por ejemplo para obtener el `campo1`, hace referencia al índice 14, en la figura 26 podemos ver que se trata de una constante `FieldRef`, que nos lleva hasta la clase `A` y siguiendo la constante `NameAndType` hasta el nombre `Campo1` y tipo entero.

También mencionar la signatura del método, debido a que no tiene parámetros y no devuelve nada se declara como `()V`.

## Ejemplo 2

En esta ocasión ejecutaremos el método `abs()` con ejecución simbólica sin generación de log.

### CÓDIGO JAVA:

```

public static int abs(int x) {
    if (x >= 0)
        return x;
    else
        return -x;
}

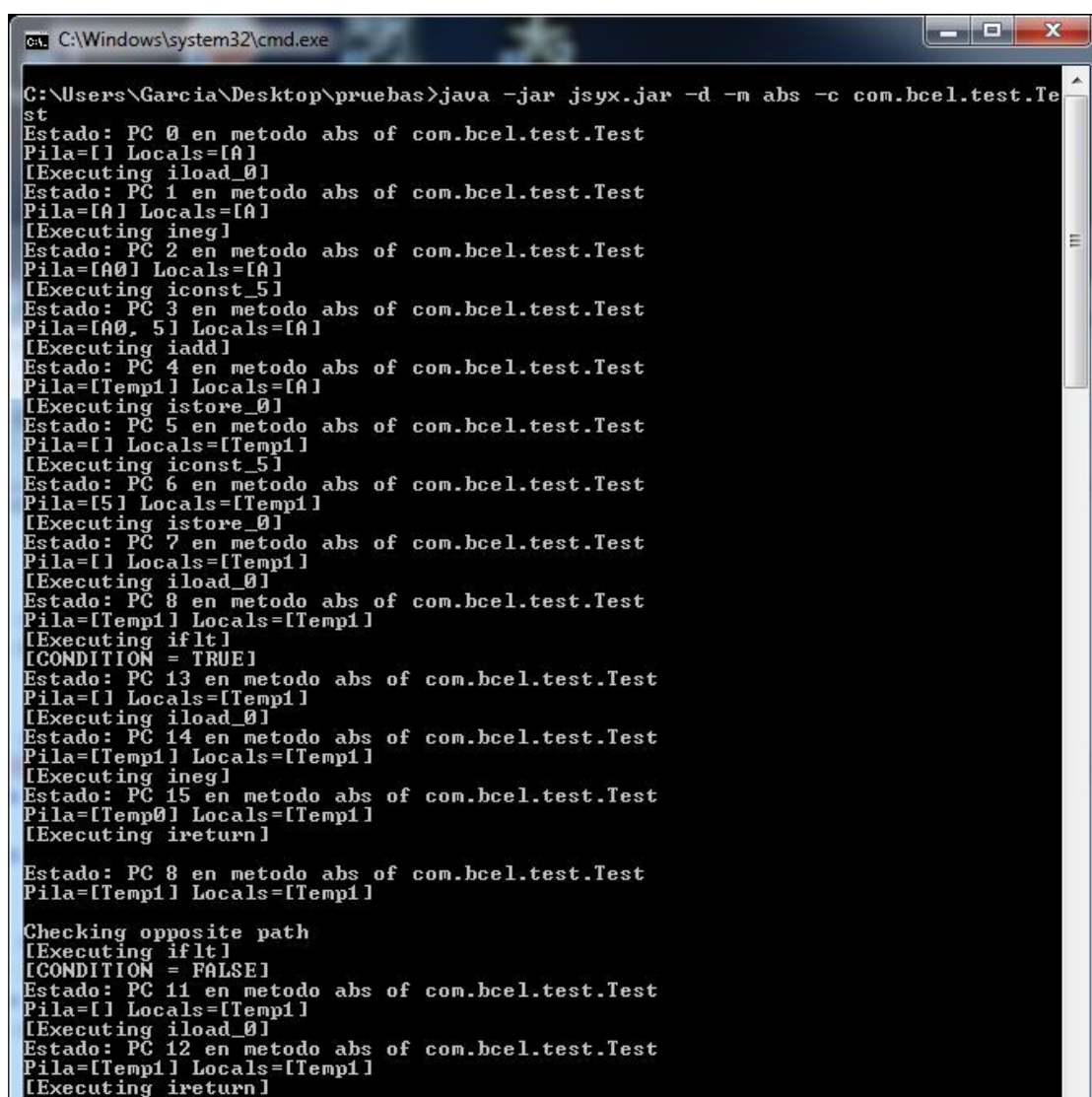
```

### CÓDIGO BYTECODE:

```

0:   iload_0
1:   iflt      #6
4:   iload_0
5:   ireturn
6:   iload_0
7:   ineg
8:   ireturn

```

**EJECUCIÓN:**


```

C:\Windows\system32\cmd.exe

C:\Users\Garcia\Desktop\pruebas>java -jar jsyx.jar -d -m abs -c com.bcel.test.Test
Estado: PC 0 en metodo abs of com.bcel.test.Test
Pila=[] Locals=[A]
[Executing iload_0]
Estado: PC 1 en metodo abs of com.bcel.test.Test
Pila=[A] Locals=[A]
[Executing negl]
Estado: PC 2 en metodo abs of com.bcel.test.Test
Pila=[A0] Locals=[A]
[Executing iconst_5]
Estado: PC 3 en metodo abs of com.bcel.test.Test
Pila=[A0, 5] Locals=[A]
[Executing iadd]
Estado: PC 4 en metodo abs of com.bcel.test.Test
Pila=[Temp1] Locals=[A]
[Executing istore_0]
Estado: PC 5 en metodo abs of com.bcel.test.Test
Pila=[] Locals=[Temp1]
[Executing iconst_5]
Estado: PC 6 en metodo abs of com.bcel.test.Test
Pila=[5] Locals=[Temp1]
[Executing istore_0]
Estado: PC 7 en metodo abs of com.bcel.test.Test
Pila=[] Locals=[Temp1]
[Executing iload_0]
Estado: PC 8 en metodo abs of com.bcel.test.Test
Pila=[Temp1] Locals=[Temp1]
[Executing iflt]
[CONDITION = TRUE]
Estado: PC 13 en metodo abs of com.bcel.test.Test
Pila=[] Locals=[Temp1]
[Executing iload_0]
Estado: PC 14 en metodo abs of com.bcel.test.Test
Pila=[Temp1] Locals=[Temp1]
[Executing negl]
Estado: PC 15 en metodo abs of com.bcel.test.Test
Pila=[Temp0] Locals=[Temp1]
[Executing ireturn]

Estado: PC 8 en metodo abs of com.bcel.test.Test
Pila=[Temp1] Locals=[Temp1]

Checking opposite path
[Executing iflt]
[CONDITION = FALSE]
Estado: PC 11 en metodo abs of com.bcel.test.Test
Pila=[] Locals=[Temp1]
[Executing iload_0]
Estado: PC 12 en metodo abs of com.bcel.test.Test
Pila=[Temp1] Locals=[Temp1]
[Executing ireturn]

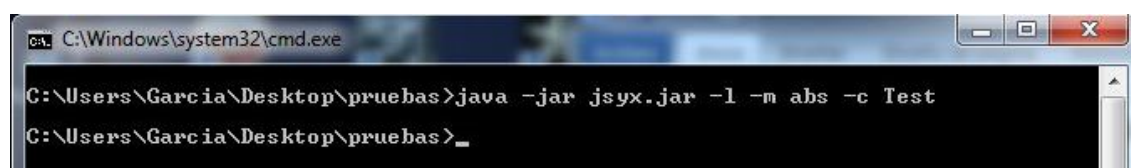
```

Figura 27. Prueba de ejecución simbólica con `-d -m -c` sobre `abs`

En la salida vemos que el programa solo ejecuta este método, internamente se realiza la preparación de los parámetros y de la máquina virtual para permitir dicha ejecución. Toma el entero `x` como un símbolo, denominándolo `A`, también vemos la generación de una variable temporal debido a la asignación sobre el símbolo.

**Ejemplo 3**

Ejecutaremos el mismo método `abs()` pero generando el archivo de log.

**EJECUCIÓN:**


```

C:\Windows\system32\cmd.exe

C:\Users\Garcia\Desktop\pruebas>java -jar jsyx.jar -l -m abs -c Test
C:\Users\Garcia\Desktop\pruebas>_

```

Figura 28. Prueba de ejecución simbólica con `-l -m -c` sobre `abs`



## 5. Cliente

---

Por pantalla no se muestra nada pero si revisamos se habrá creado un directorio de log donde podemos encontrar un archivo log.txt con el siguiente contenido:

### CONTENIDO LOG.TXT:

```
Log   Mon Jun 10 12:32:28 CEST 2013

Path -> C:\Users\Garcia\Desktop\pruebas
Class:Method -> Test:abs

Inputs Types
A [-10, 10] -> Integer

-Path 0
Solution -> True
-Inputs
A = A:-1
-Copies
A0 = A0:1
-Temps
No temporals variables

-Constraints
A:-1 <= -1
A0:1 + A:-1 = 0
-Output
A0

-Path 1
Solution -> True
-Inputs
A = A:10
-Copies
A0 = A0:-10
-Temps
No temporals variables

-Constraints
A0:-10 + A:10 = 0
A:10 >= 0
-Output
A
```

El archivo nos informa de fecha y hora de ejecución, directorio, clase y método sobre la que estamos actuando y los símbolos de entrada usados. Además nos muestra los distintos caminos que se han generado, 2 en este caso, especificando si se ha encontrado solución, si se han generado copias o valores temporales. Además de las restricciones que el código ha generado y la salida esperada.

### **Ejemplo 4**

En esta ocasión ejecutaremos un método con objetos para ver cómo se nombran y el uso de excepciones con nuestro Throwable.

### CÓDIGO JAVA:

```
public static void obj(A a, int c) {
```

```
    if (c >= 0) {  
        a.set(c);  
    }  
}
```

**CÓDIGO BYTECODE:**

```
0:    iload_1  
1:    iflt     #9  
4:    aload_0  
5:    iload_1  
6:    invokevirtual com.bcel.test.A.set (I)V (2)  
9:    return
```

**EJECUCIÓN:**

```

C:\Windows\system32\cmd.exe

C:\Users\Garcia\Desktop\pruebas>java -jar jsyx.jar -d -l -m obj -c Test
Estado: PC 0 en metodo obj of com.bcel.test.Test
Pila=[] Locals=[h=0-com.bcel.test.A, C]
[Executing iload_1]
Estado: PC 1 en metodo obj of com.bcel.test.Test
Pila=[C] Locals=[h=0-com.bcel.test.A, C]
[Executing iflt]
[CONDITION = TRUE]
Estado: PC 9 en metodo obj of com.bcel.test.Test
Pila=[] Locals=[h=0-com.bcel.test.A, C]
[Executing return]

Estado: PC 1 en metodo obj of com.bcel.test.Test
Pila=[C] Locals=[h=0-com.bcel.test.A, C]

Checking opposite path
[Executing iflt]
[CONDITION = FALSE]
Estado: PC 4 en metodo obj of com.bcel.test.Test
Pila=[] Locals=[h=0-com.bcel.test.A, C]
[Executing aload_0]
Estado: PC 5 en metodo obj of com.bcel.test.Test
Pila=[h=0-com.bcel.test.A] Locals=[h=0-com.bcel.test.A, C]
[Executing iload_1]
Estado: PC 6 en metodo obj of com.bcel.test.Test
Pila=[h=0-com.bcel.test.A, C] Locals=[h=0-com.bcel.test.A, C]
[Executing invokevirtual]
Estado: PC 0 en metodo set of com.bcel.test.A
Pila=[] Locals=[h=0-com.bcel.test.A, C]
[Executing aload_0]
Estado: PC 1 en metodo set of com.bcel.test.A
Pila=[h=0-com.bcel.test.A] Locals=[h=0-com.bcel.test.A, C]
[Executing iload_1]
Estado: PC 2 en metodo set of com.bcel.test.A
Pila=[h=0-com.bcel.test.A, C] Locals=[h=0-com.bcel.test.A, C]
[Executing putfield]
[CONDITION = NULL]
PC 6 en metodo obj of com.bcel.test.Test
Pila=[] Locals=[h=0-com.bcel.test.A, C]
No se encontro handler para h=1-java.lang.Throwable con mensaje: A is NULL

Estado: PC 2 en metodo set of com.bcel.test.A
Pila=[h=0-com.bcel.test.A, C] Locals=[h=0-com.bcel.test.A, C]

Checking opposite path
[Executing putfield]
[CONDITION = NOTNULL]
Estado: PC 5 en metodo set of com.bcel.test.A
Pila=[] Locals=[h=0-com.bcel.test.A, C]
[Executing return]
Estado: PC 9 en metodo obj of com.bcel.test.Test
Pila=[] Locals=[h=0-com.bcel.test.A, C]
[Executing return]

Log Mon Jun 10 12:48:49 CEST 2013

Path -> C:\Users\Garcia\Desktop\pruebas
Class:Method -> Test:obj

Inputs Types
A -> Object
A.valor [-10, 10] -> Integer
C [-10, 10] -> Integer

-Path 0
Solution -> True
-Inputs
C = C:-1

```

Figura 29. Prueba de ejecución simbólica con `-l -d -m -c` sobre objetos

En este caso usamos tanto `-l` como `-d`, obteniendo en la salida también lo mostrado en el log. En concreto en esta ejecución podemos ver los 3 caminos distintos, generados por el valor `if` y por el posible valor `null` del objeto `A`. También resaltar en el caso de que el objeto sea `null` el lanzamiento del `Throwable` indicando que no se ha encontrado ningún *handler* para manejarlo, terminando la ejecución.

Señalar también la notación para nombrar propiedades de objetos, *nombreObjeto.nombrePropiedad*, en este caso, *A.valor*.

### Ejemplo 5

Probaremos a ejecutar un método simple con *arrays*, hacemos uso de la opción `--array-size` para definir el tamaño de *arrays*.

#### CÓDIGO JAVA:

```
public static int array(int[] array) {
    if (array[0] == 5)
        return 0;
    else
        return 1;
}
```

#### CÓDIGO BYTECODE:

```
0:   aload_0
1:   iconst_0
2:   iaload
3:   iconst_5
4:   if_icmpne    #9
7:   iconst_0
8:   ireturn
9:   iconst_1
10:  ireturn
```

#### EJECUCIÓN:

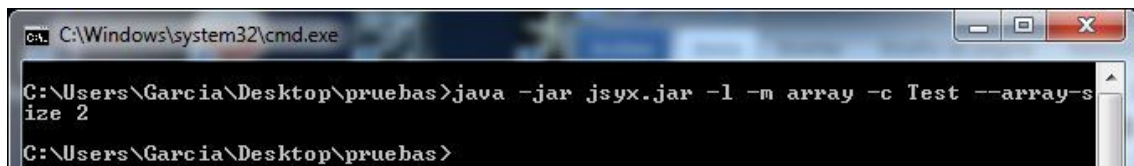


Figura 30. Prueba de ejecución simbólica con `-l -m -c --array-size` sobre *arrays*

Filtramos el contenido del archivo de log, vemos que la notación para *arrays* es fácilmente reconocible, el *array* en sí y luego sus diferentes elementos según su posición.

#### CONTENIDO LOG.TXT:

```
Inputs Types
A -> Array of int
A[0] [-10, 10] -> Integer
A[1] [-10, 10] -> Integer

-Path 0
Solution -> True
-Inputs
-Temps
A[0] = A[0]:10
A[1] = A[1]:10

-Constraints
A[0]:10 != 5
-Output
1

-Path 1
Solution -> True
-Temps
```

## 5. Cliente

---

```
A[0] = A[0]:5  
A[1] = A[1]:10
```

-Constraints

```
A[0]:5 == 5
```

-Output

```
0
```

**E**n esta parte final hablaremos sobre las conclusiones que hemos obtenido tras la elaboración de este proyecto.

### 6.1. Reflexión final

A lo largo de este proyecto, hemos construido una herramienta capaz de ejecutar de manera normal o simbólica cualquier archivo *.class*. Esto nos ha proporcionado un gran conocimiento de cómo funcionan las máquinas virtuales, en especial la máquina virtual de java. Gracias a librerías como *BCEL*, hemos sido capaces de conseguir toda la información necesaria para poder descomponer el archivo *.class* y conseguir crear una máquina virtual capaz de ejecutar dichas instrucciones. Esto nos ha otorgado un entendimiento a nivel de *bytecode* de cómo funciona el repertorio de instrucciones de *Java*, los *frames*, *heaps* y *Constant Pool* propios de la *JVM*.

Otro concepto que hemos adquirido en este proyecto es la ejecución simbólica. En ella aprendemos el concepto de ejecutar un programa sin necesidad de tener valor para los datos de entrada a favor de tener símbolos, donde se generaran automáticamente un rango de valores. Este rango de valores irá en función de restricciones que gracias al uso de la librería *choco* nos ha permitido entender como funciona la programación con restricciones y el potencial que conlleva. Trabajar con un proyecto tan longevo, en el mundo de la informática como es *choco* que nació en 1999 con una gran documentación nos ha permitido afianzar conceptos del uso de librerías de *Java*.

También ha sido interesante el trabajo con la librería *Commons Cli* para la creación del cliente de la aplicación, que facilita en gran medida su uso y permite codificar los parámetros típicos de cualquier programa para ejecución desde línea de comandos de una manera sencilla y directa (aunque hasta cierto punto algo limitada), pero que de todos modos provee una manera de acceder a las funcionalidades de *JSyX* de una forma bastante transparente.

### 6.2. Trabajo futuro

Este documento es una aproximación a la ejecución simbólica, dado a la complejidad y la extensión de realizar una máquina de ejecución simbólica para *Java* aún quedaría añadir funcionalidad al proyecto

Así podemos enumerar las siguientes mejoras:

- Añadir opciones de búsqueda: la aplicación solo recorre el árbol primero en profundidad lo que en muchos casos no sería la opción completa ni óptima. Por ello definir una estrategia de búsqueda personalizada generaría una mejora sustancial en cuanto a rendimiento.
- Fijar el límite de iteraciones en bucle o *timeout*: otra opción para fijar el tamaño del árbol podría ser fijar el número de iteraciones que se quieren efectuar dentro de un bucle o el tiempo máximo de ejecución que queramos que el programa este en ejecución
- Herencia: es uno de los aspectos más importantes de la programación orientada a objetos, pero también una de las más problemáticas para la ejecución simbólica. Cada casting propondría nuevas ramas al árbol y en algún uso de objetos que hereda de otra

clase, no podremos llegar a estar seguros de que clase se trata, resultando nuevas *constraints* que tendrían que ser definidas en el choco.

- Uso de *arrays* sin limitación de tamaño: esto supone la creación de un nuevo resolutor de constraint capaz de resolver las restricciones creadas solo para los arrays.
- Uso de excepciones: en nuestro caso solo se permite el uso de la clase padre de la jerarquía de excepciones, *Throwable*, para permitir otros tipos se deberían implementar algunos métodos nativos que permitieran más variedad.



- [1] J. C. King, «Symbolic execution and program testing vol. 19, no. 7, pp.385 -394,» 1976. [En línea]. Available: <http://www.cs.uiuc.edu/~madhu/cs598-fall10/king76symbolicexecution.pdf>. [Último acceso: 9 6 2013].
- [2] National Institute of Standards and Technology, «The economic impacts of inadequate infrastructure for software testing,» 2002.
- [3] N. Tillmann y J. d. Halleux, «Pex - white box test generation for .NET in TAP'08,» Abril 2008.
- [4] C. Cadar, V. Ganesh, P. Pawlowski, D. Dil y D. Engler, «EXE: Automatically generating inputs of death in CSS'06,» Noviembre 2006.
- [5] P. Godefroid, N. Klarlund y K. Sen, «DART: Directed Automated Random Testing in PLDI'05,» Junio 2005.
- [6] vmware, «Virtualization Basics,» [En línea]. Available: <http://www.vmware.com/es/virtualization/virtualization-basics/virtual-machine.html>. [Último acceso: 4 Junio 2013].
- [7] Computer Weekly, «Write once, Run anywhere,» Mayo 2002. [En línea]. Available: <http://www.computerweekly.com/feature/Write-once-run-anywhere>. [Último acceso: 6 Junio 2013].
- [8] T. Lindholm y F. Yellin, «Oracle JVM Specification,» 1999. [En línea]. Available: <http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html>. [Último acceso: 2013 Junio 3].
- [9] IBM, « Compilador Just-In-Time,» [En línea]. Available: <http://pic.dhe.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.lnx.70.doc%2Fuser%2Fjit.html>. [Último acceso: 19 Junio 2013].
- [10] J. Pearce, «JVM Organization,» [En línea]. Available: <http://www.cs.sjsu.edu/~pearce/modules/lectures/co/jvm/organization.htm>. [Último acceso: 3 6 2013].
- [11] Apache Foundation, «Apache Commons BCEL manual,» 17 Octubre 2011. [En línea]. Available: <http://commons.apache.org/proper/commons-bcel/manual.html>. [Último acceso: 3 Junio 2013].

- [12] B. Bumgarner, «Historia del magic number,» 28 Enero 2003. [En línea]. Available: <http://radio-weblogs.com/0100490/2003/01/28.html>. [Último acceso: 5 Junio 2013].
- [13] Oracle, «JVM Specification Constant Pool,» [En línea]. Available: <http://docs.oracle.com/javase/specs/jvms/se5.0/html/ClassFile.doc.html#20080>. [Último acceso: 5 Junio 2013].
- [14] J. Pearce, «JVM Simple Code,» [En línea]. Available: <http://www.cs.sjsu.edu/~pearce/modules/projects/compOrg/jvm/index.htm>. [Último acceso: 3 6 2013].
- [15] M. Maniruzzaman, «CodeProject,» 16 Marzo 2008. [En línea]. Available: <http://www.codeproject.com/Articles/24029/Home-Made-Java-Virtual-Machine>. [Último acceso: 3 Junio 2013].
- [16] B. Venners, «How the Java Virtual Machine Handles Exceptions,» Enero 1997. [En línea]. Available: <http://www.artima.com/underthehood/exceptions.html>. [Último acceso: 5 Junio 2013].
- [17] R.-G. Xu, «Symbolic Execution Algorithms for Test Generation,» 2009. [En línea]. Available: <http://www.mpi-sws.org/~rupak/Papers/RuGangXuThesis.pdf>. [Último acceso: 9 6 2013].
- [18] K. R., «Principles of Constraint Programming,» 1999.
- [19] The choco team, «Choco: an Open Source Java Constraint Programming Library,» 2009. [En línea]. Available: <http://www.emn.fr/z-info/choco-solver/uploads/pdf/choco-presentation.pdf>. [Último acceso: 9 6 2013].
- [20] Apache Foundation, «Apache Commons CLI,» 27 Febrero 2013. [En línea]. Available: <http://commons.apache.org/proper/commons-cli/>. [Último acceso: 3 Junio 2013].
- [21] Apache Foundation, «Apache Commons CLI usage,» 27 Febrero 2013. [En línea]. Available: <http://commons.apache.org/proper/commons-cli/usage.html>. [Último acceso: 3 Junio 2013].

## Apéndice A: Listado instrucciones *bytecode* admitidas

- NOP
- Operaciones con pila
  - Instrucciones que ponen una constante en la pila.
    - ICONST\_M1
    - ICONST\_0
    - ICONST\_1
    - ICONST\_2
    - ICONST\_3
    - ICONST\_4
    - ICONST\_5
    - ACONST\_NULL
    - LCONST\_0
    - LCONST\_1
    - FCONST\_0
    - FCONST\_1
    - DCONST\_0
    - DCONST\_1
    - BIPUSH
    - SIPUSH
  - Instrucciones que cargan un elemento desde la *Constant Pool*
    - LDC
    - LDC2\_W
  - Instrucciones que cargan una variable en la pila
    - ILOAD
    - ILOAD\_0
    - ILOAD\_1
    - ILOAD\_2
    - ILOAD\_3
    - LLOAD
    - LLOAD\_0
    - LLOAD\_1
    - LLOAD\_2
    - LLOAD\_3
    - FLOAD
    - FLOAD\_0
    - FLOAD\_1
    - FLOAD\_2
    - FLOAD\_3
    - DLOAD
    - DLOAD\_0
    - DLOAD\_1
    - DLOAD\_2
    - DLOAD\_3
    - ALOAD
    - ALOAD\_0

- ALOAD\_1
  - ALOAD\_2
  - ALOAD\_3
- Instrucciones que cargan valores de *arrays*
  - IALOAD
  - LALOAD
  - FALOAD
  - DALOAD
  - CALOAD
  - SALOAD
  - BALOAD
  - AALOAD
- Instrucciones que guardan en una variable local un valor de la pila
  - ASTORE
  - ASTORE\_0
  - ASTORE\_1
  - ASTORE\_2
  - ASTORE\_3
  - ISTORE
  - ISTORE\_0
  - ISTORE\_1
  - ISTORE\_2
  - ISTORE\_3
  - LSTORE
  - LSTORE\_0
  - LSTORE\_1
  - LSTORE\_2
  - LSTORE\_3
  - FSTORE
  - FSTORE\_0
  - FSTORE\_1
  - FSTORE\_2
  - FSTORE\_3
  - DSTORE
  - DSTORE\_0
  - DSTORE\_1
  - DSTORE\_2
  - DSTORE\_3
  - IASTORE
  - LASTORE
  - FASTORE
  - DASTORE
  - CASTORE
  - SASTORE
  - BASTORE
  - AASTORE
- Instrucciones para operaciones genéricas de pila
  - POP
  - POP2
  - DUP
- Operaciones aritméticas
  - Suma

- IADD
    - LADD
    - FADD
    - DADD
  - Resta
    - ISUB
    - LSUB
    - FSUB
    - DSUB
  - Multiplicación
    - IMUL
    - LMUL
    - FMUL
    - DMUL
  - División
    - IDIV
    - LDIV
    - FDIV
    - DDIV
  - Resto
    - IREM
    - LREM
    - FREM
    - DREM
  - Negación
    - INEG
    - LNEG
    - FNEG
    - DNEG
  - Incremento
    - IINC
- Operaciones con objetos y *arrays*
  - Para manejo objetos
    - NEW
    - GETFIELD
    - PUTFIELD
    - GETSTATIC
    - PUTSTATIC
  - Para manejo de *arrays*
    - NEWARRAY
    - ANEWARRAY
    - ARRAYLENGTH
- Instrucciones para excepciones
  - ATHROW
- Instrucciones para control de flujo del programa
  - Instrucciones de salto condicional
    - IFNONNULL
    - IFNULL
    - IFEQ
    - IFNE
    - IFLT
    - IFGE

- IFGT
  - IFLE
  - IF\_ICMPEQ
  - IF\_ICMPNE
  - IF\_ICMPLT
  - IF\_ICMPGE
  - IF\_ICMPGT
  - IF\_ICMPLE
  - IF\_ACMPEQ
  - IF\_ACMPLT
  - LCMP
  - DCMPL
  - DCMPLG
- Instrucciones de salto incondicional
  - GOTO
- Instrucciones para invocación de métodos y de retorno
  - Para retorno de valores
    - RETURN
    - IRETURN
    - LRETURN
    - FRETURN
    - DRETURN
    - ARETURN
  - Para invocación de métodos
    - INVOKESPECIAL
    - INVOKESTATIC
    - INVOKEVIRTUAL