Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Ejecución Simbólica Dinámica en Evosuite: Estudio e Implementación

Tesis de Licenciatura en Ciencias de la Computación

Ignacio Manuel Lebrero Rial

Director: Dr. Juan Pablo Galeotti

Buenos Aires, 2020

EJECUCIÓN SIMBÓLICA DINÁMICA EN EVOSUITE: ESTUDIO E IMPLEMENTACIÓN

Evosuite [1] es una herramienta de generación automática de tests que utiliza, entre otros, un algoritmo genético para generar tests unitarios en clases escritas en código Java. En particular, el módulo de ejecución simbólica dinámica actual de la herramienta usa una técnica híbrida con el algoritmo genético previamente mencionado. En esta tesis, primero construimos un módulo independiente que utiliza sólo *ejecución simbólica dinámica*. En esta etapa, separamos el módulo actual de ejecución simbólica dinámica y reconstruimos el algoritmo de exploración basándonos en el de *SAGE* [2] para luego comparar el nuevo módulo contra el viejo sobre un benchmark ya construido [3]. Por ultimo, extendemos la expresividad del módulo para poder representar arreglos como elementos simbólicos.

Palabras claves: Ingeniería del Software, Testeo de Software, Análisis Estático, Análisis Dinámico, Ejecución Simbólica Dinámica, Ejecución Concolica, Generación Automática de Testing, Evosuite).

DYNAMIC SYMBOLIC EXECUTION ON EVOSUITE: STUDY AND IMPLEMENTATION

Evosuite [1] is an automated testing generation tool that uses, amoung others, a genetic algorithm tecnique to generate test cases for classes written in Java code. In particular, it's current dynamic symbolic execution module uses a hybrid approach combining it with the genetic algorithm. In this thesis, first we build a stadalone module that uses only dynamic sybolic execution. In this stage, we separate the dynamic simbolic execution module from the genetic algorithm and rebuild the exploration algorithm based on *SAGE's* [2] implementation to later compare it against the old one by using an already created benchmark [3]. Finally, we extend the expressiveness of the module to being able to represent arrays as symbolic elements.

Palabras claves: Software Engineering, Software Testing, Static Analisis, Dynamic Analisis, Dynamic Symbolic Execution, Concolic execution, Automated Testing Generation, Evosuite).

A mi familia.

Índice general

Índice de figuras

1. INTRODUCCIÓN

Si bien construir software forma gran parte del desarrollo, asegurar que es correcto y no contiene errores constituye otro desafío. Testear software tiende a ser tedioso y consumir una cantidad significativa de tiempo debido a tiempo de mantenimiento y creatividad intelectual que requiere hacerlo. Además, los costos de mantenimiento consisten aproximadamente del 67 % del tiempo de desarrollo, debido a esto tomarse el tiempo para testear de manera minuciosa es esencial para obtener un producto final de calidad. [4]

1.1. Motivación

Usualmente hay un conjunto de requerimientos a los que el software debe adherirse. Originalmente, el testeo manual de programas fue la técnica que usualmente se utilizó para probar que el software cumplía con sus requerimientos en la industria. Este acercamiento tiene sus contras, además del tiempo requerido para ejecutar todos los tests, hay un factor de error humano que puede ocurrir mientras se testea el sistema sumado a la necesidad de contratar a alguien para que haga la tarea. Esta idea termina siendo cara en términos tanto monetarios como temporales, además de ser propensa a errores.

Un acercamiento distinto es generar una pieza de software que se encargue de testear el sistema de alguna manera. Trabajo preliminar hecho por Howden [5] en definir oráculos (como un programa debería responder a un determinado input) proporcionó un paso adelante en el área dando lugar a técnicas como testing unitario, de integración y de sistema que pudieran ejecutarse de manera automática, haciendo más fácil el testeo de software.

El acercamiento previo trajo algunos cambios al desarrollo de software. Aplicar esas técnicas requería que los diseños de los sistemas fueran modulares y flexibles a cambios para poder ser testeados de manera correcta e independiente, lo cual empujó a la creación de software mantenible. El trabajo realizado por Gamma et al. [6] representó avances en diseño orientado a objetos en particular, proporcionando diferentes técnicas para construir tales sistemas y, como efecto secundario, hacerlos más fáciles de testear.

Un dato importante es que los lenguajes de programación, como cualquier otro lenguaje formal, se componen de reglas sintácticas y semánticas. Esto define tanto la gramática del lenguaje como su significado, pero sólo de manera computable. Esto es, dada una pieza de software, una computadora puede decidir si su construcción gramatical es válida y cuáles son los efectos que esas instrucciones tienen cuando se ejecuten en el sistema (e.g. suma, asignación de variables, asignación de memoria, etc.). Supongamos que se desarrolla un programa P_1 para calcular el impuesto a las ganancias, este contiene una serie de instrucciones que resultan, supuestamente, en el cálculo del valor del impuesto. Qué significa el impuesto a las ganancias y si la manera en la que el programa lo computa es correcta escapan al alcance del lenguaje de programación.

Dado el ejemplo anterior, asumiendo que un programa correcto P_1^* para calcular el impuesto a las ganancias existe, el programador puede crear un conjunto de tests para

comprobar que el comportamiento de P_1 es correcto. Estos estarían orientados a comprobar tanto que el programa termina correctamente (no genera errores durante su ejecución) como que el valor retornado es correcto. Esto define dos estrategias: (1) Ejecutar correctamente el programa y (2) el uso de un oráculo para comprobar la correctitud del resultado obtenido. El oráculo sólo puede ser comprendido por el programador que escribió los casos de prueba, a menos que se utilice algún lenguaje de especificación que pueda ser aplicado al programa. Como (1) requiere la ejecución de código sin importar si el valor retornado es correcto, el problema puede ser reducido a ejecutar líneas de código para revisar si contienen bugs. En este espacio entra una subsección de generación automatizada de tests, haciendo uso de las propiedades previamente mencionadas para generar tests de manera automática en un intento de reducir la cantidad de trabajo tedioso para los programadores y a su vez incrementar la calidad de los productos de software.

1.2. Trasfondo

1.2.1. Ejecución Simbólica Dinámica

Ejecución simbólica, como se menciona en Baldoni et. al. [7] es una técnica introducida en la década del 70, en particular, apunta a testear que algunas condiciones no se den, e.g. división por cero, dereferencias a NULL pointers, backdoors para evitar la autenticación de un programa. La idea es sistemáticamente explorar varios caminos posibles de ejecución sin necesariamente requerir inputs concretos. En particular, este trabajo va a estar enfocado en ejecución simbólica dinámica (DSE por las siglas en ingles a partir de ahora), una variante explorada por Godefroid et. al. [8] donde inputs concretos se usan para guiar los caminos de la ejecución simbólica. Esto es, la ejecución de un camino en el programa queda guiada por los inputs concretos y valores simbólicos son recolectados mientras se lo recorre, esto resulta en una condición de camino que habla de la forma que tienen los inputs para recorrerlo. Luego, la condición de camino es modificada para hablar de un nuevo camino no recorrido previamente y, por último, se recurre a un demostrador de teoremas para obtener valores concretos a partir de la nueva condición de camino generada, lo que resulta en un nuevo caso de prueba que se usa en la siguiente iteración como input concreto hasta que algún criterio para finalizar es alcanzado. Varias herramientas fueron implementadas como DART[8], CUTE [9] y CREST [10] para programas C, KLEE [11] construido sobre el framework LLVM, SAGE [2] y MAYHEM [12] orientados a testear aplicaciones en lenguaje ensamblador y PEX[13] orientado a .NET[14]. Muchas herramientas también están orientadas hacia Java como JCUTE [15] y CATG[16] por un lado y JFUZZ[17] y JDART[18] construidas sobre Java Path-Finder[19] por otro. Si bien, las últimas herramientas ofrecen una manera de ejecutar la técnica en programas Java, no están diseñadas para ser fácilmente extensibles para experimentar. En este trabajo intentamos reescribir el módulo de una herramienta existente.

1.2.2. Evosuite

Evosuite [20] es una herramienta de generación automática de tests para programas Java. Se enfoca principalmente en Search Based Software Testing [21] (SBST a partir de ahora) en el cual la herramienta fue creciendo incrementalmente a través de varios trabajos [22] [23]. En particular, en un punto se integró un módulo DSE construido sobre un algoritmo genético previamente agregado [3]. Esta tesis apunta a desarrollar, por un lado,

un módulo de DSE independiente en la herramienta junto con un algoritmo de exploración extensible [2] y, por otro, extender el soporte simbólico para soportar objetos complejos[24].

1.3. Estructura de la Tesis

Esta tesis está estructurada de la siguiente manera. En el capítulo 2 introduciremos el trasfondo teórico necesario para entender el resto de este trabajo. En el capítulo 3 presentamos el nuevo módulo de ejecución simbólica dinámica junto con su algoritmo de exploración. En el capítulo 4 presentaremos el soporte para arreglos simbólicos. En el capítulo 5 presentamos la experimentación preliminar. En el capítulo 6 daremos las conclusiones y el trabajo futuro.

2. PRELIMINARIES

En el capítulo anterior introdujimos la motivación y las nociones básicas de esta tesis. En este, explicaremos el trasfondo teórico requerido para entender el resto de este trabajo.

2.1. Testing

La generación automática de casos de prueba apunta a crear de manera automática una pieza de software que estará encargada de revisar que un programa dado P funcione como es esperado. Antes de entrar en profundidad en esto, veamos algunos conceptos básicos sobre testing.

2.1.1. Nociones Básicas

A la hora de crear una pieza de software, un programador puede introducir errores de manera no intencional en el código, estos pueden ir desde cosas simples como intentar dividir por cero hasta cosas mucho mas complejas como ser un error de redondeo silencioso en un cálculo (por utilizar una conversión entre tipos de datos numéricos por ejemplo) que resulte en un comportamiento no deseado en un módulo distinto que dependa de ese valor mucho mas adelante en el tiempo. Podemos desglosar este ejemplo en cuatro componentes:

Errores

Definición 2.1.1. Error Un error es una desviación no buscada ni intencional de lo que es correcto, esperado o verdadero.

Definición 2.1.2. Defecto Un defecto es un error en el código del programa, específicamente uno que puede crear una infección (y conducir a una falla). Estos son creados por los programadores de manera no intencional.

Definición 2.1.3. Infección Una infección es un error en el estado del programa, específicamente uno que puede llevar a una falla. Estos son causados por defectos y/o infecciones previas.

Definición 2.1.4. Falla Una falla es un error externamente visible (hacia el usuario) en el comportamiento del programa. Estos son causados por infecciones.

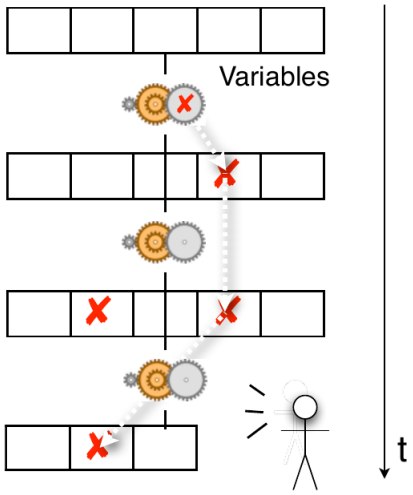

Fig. 2.1: Propagación de una falla en el tiempo.

Siguiendo a modo de ejemplo la figura 2.1 podemos ver que:

1. El programador crea un **defecto** en el código (La primer cruz roja en la figura, mirando de arriba hacia abajo).
2. Cuando es ejecutado, el defecto crea una **infección** en el estado del programa.
3. La infección se propaga a través de la ejecución.
4. La infección causa una falla en la última línea, la cual es vista por el usuario.

Casos de prueba

Por otro lado, un programador puede generar casos de prueba, estos apuntan a verificar que el programa sea correcto, más formalmente:

Definición 2.1.5. Caso de Prueba Un caso de prueba (test case de ahora en adelante) es una entidad compuesta por:

- **Input:** Una serie de valores de entrada para el programa a ser testeado.
- **Oráculo [5] :** Un procedimiento que verifica que el programa se comporta como es esperado dado un input predefinido.

Definición 2.1.6. Test Suite Una test suite es un conjunto de test cases.

Por lo general, a la par de una pieza de software también se genera un test suite, el cual tiene el objetivo de validar el comportamiento del software desarrollado.

Los test cases, a su vez, pueden ser categorizados según a que parte del sistema están orientados a testear:

Definición 2.1.7. Test Unitario Un test case se dice que es unitario si está orientado a ejercitar una parte individual de un programa (e.g. un módulo, una clase, un método, etc.).

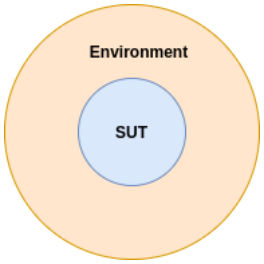

Fig. 2.2: Idea de un test unitario. El sujeto bajo prueba es testeado mientras que su entorno está simulado. Esto es, se lo mockea para simular una respuesta del mismo pero no se lo ejecuta de manera que el SUT quede aislado y su test no dependa de cosas externas.

Definición 2.1.8. Test de integración Un test case se dice que es de integración si está orientado a ejercitar varios módulos a la vez para evaluar la correcta interacción entre ellos.

Fig. 2.3: Idea de un test de integración: en la figura, cada cuadrado representa un módulo y la flecha roja representa el flujo de ejecución entre ellos.

Definición 2.1.9. Test de sistema Un test case se dice que es de sistema si está orientado a ejercitar un sistema en su totalidad.

Dependiendo de la categoría, el approach con el que desarrollas los tests puede variar significativamente. Por lo general, para test unitarios se utiliza un aislamiento completo de la pieza de código bajo test de manera que el entorno en el que se ejecuta pueda ser simulado, ya que la pieza a evaluar no necesita saber como funciona su contexto, esto ayuda a evitar que factores externos produzcan cambios en la sección que queremos evaluar. Por otro lado, cuando se desarrollan tests de sistema por lo general se utiliza alguna métrica a gran escala (e.g. performance, usabilidad, fiabilidad, instalación, etc.), lo que genera una necesidad por configurar el sistema entero lo cual puede llegar a ser una tarea significativamente tediosa y compleja.

2.1.2. Criterios de Testing

Cuando se desarrolla un test suite, por lo general se usa algún objetivo al cual el test suite está orientado y no simplemente a eliminar todos los errores del programa. Esto se debe a que sólo se puede probar que un sistema contiene errores pero nunca probar la ausencia de los mismos [25]. Debido a esto, se usan diferentes criterios objetivos de calidad para obtener una noción de que tan bueno puede ser un test suite con respecto a

algún elemento del **sujeto bajo prueba** (SUT desde ahora por las siglas en ingles). Una estructura muy usada es el *grafo de control de flujo* (CFG desde ahora por las siglas en ingles) por la facilidad en representar las instrucciones y los posibles caminos que puede haber en la ejecución. Una métrica relevante será la de ramas, estas son el conjunto de arcos que van de un nodo que represente una *estructura de control* (e.g. if o switch) hacia otro nodo. Por ejemplo, como se puede ver en la figura 2.4, el grafo contiene dos ramas {A, B} y {A, C},

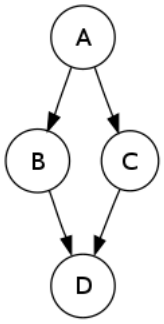

Fig. 2.4: Instrucción if representada en un CFG.

Varias métricas existen en la literatura [26], este trabajo hará foco en usar **cobertura de ramas** en particular.

Definición 2.1.10. Cobertura de Ramas Dado un test suite T con respecto a un SUT S, definimos la cobertura de ramas (branch coverage de ahora en adelante) de T sobre S como el porcentaje de ramas ejercitadas en S cuando T es ejecutado.

2.2. Java

Java es un lenguaje de programación orientado a objetos con una idea WORA (Write once run everywhere) por detrás. El lenguaje es primero compilado a una representación intermedia (*java byte-code*), la cual es luego interpretada por la *Máquina Virtual de Java* (JVM de ahora en adelante por sus siglas en ingles), esta es responsable de ejecutar las instrucciones en la máquina física subyacente. Esta tesis hará foco en Java 8 [27].

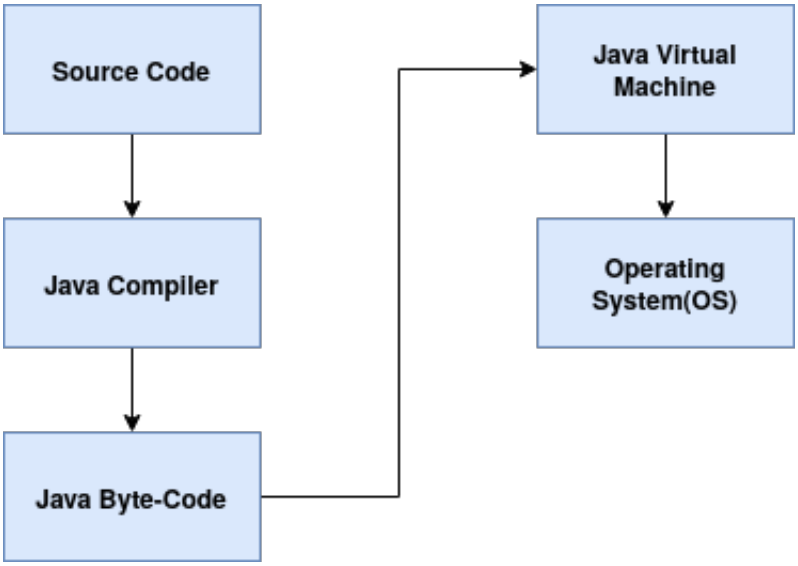

Fig. 2.5: Flujo de ejecución de código en Java

2.2.1. JVM

La JVM es responsable de ejecutar instrucciones en java byte-code sobre la actual computadora, esto es lo que permite que la idea WORA sea implementada, ya que el programa es compilado una sola vez y luego, puede ser ejecutado en cualquier sistema siempre y cuando sea soportado por la JVM.

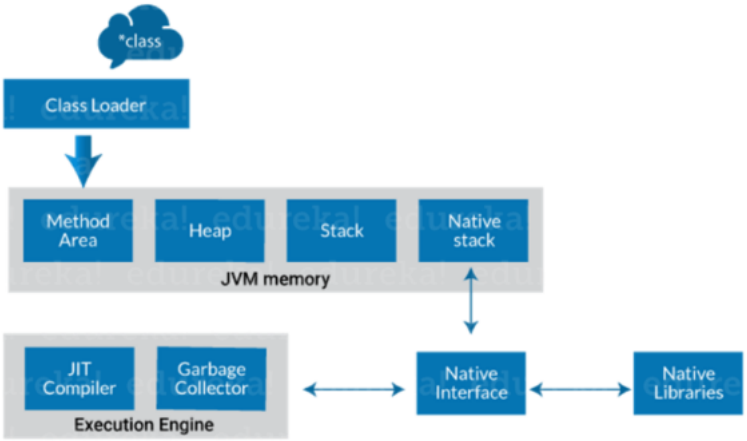

Fig. 2.6: Arquitectura de la JVM.

Class Loaders

Una de las responsabilidades de la JVM es *cargar el código* y luego *ejecutarlo* entre otras, esto se hace a través de *class loaders*. Cuando la JVM pide una clase, el class loader intenta localizarla y cargar su definición en el entorno de ejecución (run-time environment). Esto se hace, en parte, leyendo el byte-code de la clase, en este punto se puede introducir

instrumentado para manipular el byte-code usando la *API de instrumentado de Java* [28] o frameworks como *ASM* [29].

Memoria Interna

Recapitulando de la figura 2.6, las secciones relevantes de la memoria son:

- **Method Area:** Esta sección se encuentra compartida entre **todos** los threads de la JVM. Guarda toda la información a nivel clase, como:
  - **Field Data:** nombre, tipo, modificadores y atributos por campo.
  - **Method Data:** Igual que el campo pero para métodos incluyendo tipo de retorno, tipo de los parámetros en orden, etc... por método.
  - **Method Code:** byte codes, tamaño de stack de operandos, tamaños de las variables locales, tabla de variables locales, etc.
  - **Run time constant pool:** Constantes numéricas, referencias a campos, referencias a métodos, atributos, constantes de cada clase e interfaces, referencias a campos de métodos, etc.
- **Heap:** Esta sección se encuentra compartida de la misma manera que el *method area*. Contiene los datos de todos los objetos y sus correspondientes instancias de variables y arrays. Es importante mencionar que, a diferencia de lenguajes basados en *.NET*, el modelo de memoria de Java es simple. Sólo usa referencias para modelar memoria dinámica en lugar de punteros (C# por ejemplo usa ambos [14]), esto libera al programador de la responsabilidad de manejar manualmente el ciclo de vida de los objetos (e.g. usando un método destructor cuando ya no se lo necesite o liberando la memoria de punteros sin usar.), delegando esa responsabilidad al *garbage collector* el cual se encargará de hacer un pasaje por la memoria periódicamente para liberar el espacio de las referencias que ya no sean utilizadas y reorganizar la memoria.
- **Stack:** Para cada thread de la JVM, se crea un stack separado para guardar *method frames* y para controlar la invocación y el retorno. Por cada llamada, una nueva entrada (**frame)** será añadido a la cima del stack. Cada frame contiene:
  - **Local Variable Array:** Donde se guardan los parámetros del método y las variables locales.
  - **Operand Stack:** Actúa como un espacio de trabajo para realizar acciones intermedias.
  - **Constant Pool Reference:** Es donde los símbolos relacionados con los métodos son guardados.

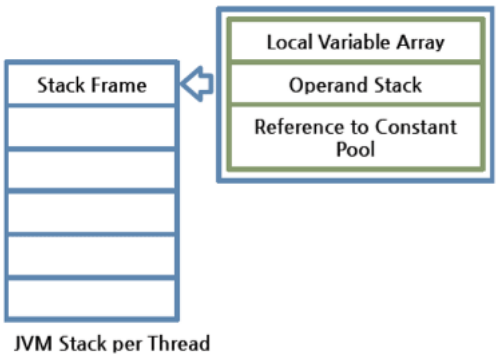

Fig. 2.7: Estructura del Stack de la JVM

Es importante notar que los operandos están separados en categorías según cuantas entradas en el arreglo de locales necesiten, en particular hay dos:

1. boolean, byte, char, short, int, float, reference y returnAddress
2. long y double

Los de la primera categoría ocupan un slot y los de la segunda dos, como analogía se los puede pensar como que los primeros representan operandos de *32bits* mientras que los segundos de *64bits*.

2.3. Teorías de Satisfacibilidad Módulo

El problema de teorías de satisfacibilidad módulo (SMT de ahora en adelante por las siglas en ingles) es un problema de decisión para fórmulas lógicas con respecto a una combinación de teorías de trasfondo expresadas en lógica de primer orden clásica con igualdad[30]. En particular un SMT-solver esta construido sobre un SAT-solver, este es usado para ver si una fórmula booleana puede ser satisfecha por alguna asignación de variables, por ejemplo:

$$(p \vee q) \wedge (\neg p \vee \neg q)$$

puede ser satisfecha usando la asignación

$$p = true \text{ y } q = false$$

Un SMT-solver extiende sobre esto agregando teorías de trasfondo, por ejemplo la teoría de los enteros, arreglos y reales entre otras. Por ejemplo usando la teoría de enteros podemos tener una fórmula de la forma

$$(x + 2 > 0 \wedge x * y > 0)$$

Siendo un resultado factible

$$x = 1 \text{ y } y = 1$$

En particular, en el caso de eliminar los cuantificadores, SMT subsume SAT el cual es un problema *NP-completo*. Incluso algunas fórmulas aceptadas por SMT solvers pertenecen a clases de complejidad más altas o incluso son indecidibles.

2.3.1. SMT-lib

SMT-lib[31] es un estándar desarrollado para facilitar la investigación y el desarrollo de SMT. Su principal foco es desarrollar un lenguaje de entrada/salida común para los SMT-solvers y además proveer un estándar riguroso sobre las descripciones de teorías de trasfondo usadas en sistemas SMT. Esto es muy útil a la hora diseñar una herramienta que utilice un SMT-solver, ya que permite asegurar ciertas cosas sobre el uso y las propiedades que cumplan las implementaciones del mismo y hace fácil la implementación en el caso de que se quiera intercambiar solvers. Además, el estándar no obliga a los solvers a implementar todas las teorías de trasfondo, con lo que de necesitarse un solver especializado en alguna teoría o que tenga soporte para elementos específicos se pueden intercambiar sin demasiado trabajo.

2.4. Ejecución Simbólica

Mientras que una ejecución concreta de un programa con un input específico explora un camino particular en el CFG, la idea detrás de esta técnica se centra en cómo generar un conjunto de inputs que cubran todos (o la mayor cantidad posible) los caminos. En lugar de usar valores concretos como inputs, se utilizan valores simbólicos que representen los valores de variables como expresiones simbólicas de los inputs. Luego, el programa es simbólicamente ejecutado, manteniendo un seguimiento del estado de la ejecución simbólica en una *memoria simbólica* σ mientras se simulan la ejecución de las instrucciones del programa. En particular, al encontrar una condición de brancheo (e.g. if), se genera un constraint sobre los valores simbólicos predicados en la condición y se lo acumula en una **condición de camino**, la cual representará la estructura que deben cumplir los inputs para haber tomado ese camino a lo largo del CFG durante la ejecución. En cada branch encontrado, la condición de camino actual debe ser feasible (e.g. debe existir algún conjunto de inputs para los cuales ese camino sera recorrido). Para corroborar que el camino sea feasible se usa un *demostrador de teoremas*, usualmente basado en un **SMT solver** [31], como Z3[32], CVC4[33] o Yices[34] por ejemplo, el cual resuelve satisfiabilidad y soluciones posibles para las condiciones de camino.

Más formalmente, dado un SUT P, una ejecución simbólica de P irá recolectando condiciones simbólicas sobre los valores asignados a variables v a lo largo del programa, los cuales son expresados en términos de los inputs. Una memoria simbólica σ es usada para mapear direcciones de memoria (donde se encuentran alocadas las variables) a valores simbólicos. Un valor simbólico es cualquier expresión e en alguna teoría T donde todas las variables libres son exclusivamente inputs del SUT. Para cualquier variable v, $\sigma[v]$ denota el valor simbólico de v en σ. Usamos la notación + para denotar la actualización de mapeos dentro de la memoria simbólica. Por ejemplo, $\sigma' = \sigma + [v \to e]$ es el mismo mapa que σ, excepto que $\sigma'[v] = e$. Al encontrar una instrucción de brancheo (e.g. if o $switch$) se generará una **restricción** sobre los inputs simbólicos y se la agregará a una **condición de camino**.

Definición 2.4.1. Restricción Una *restricción* (constraint a partir de ahora) es el resultado de diferentes statements del programa que definen el camino tomado por la ejecución (e.g. por un if o un $switch$) cuyo resultado depende de valores simbólicos.

Definición 2.4.2. Condición de Camino Sea un programa P y sea C su CFG asociado, una *condición de camino* (path condition (PC) a partir de ahora) de un camino c en C

es una formula booleana sobre los inputs simbólicos, la cual es una conjunción lógica de *constraints* que los inputs deben satisfacer para que una ejecución de P recorra el camino c.

2.4.1. Ejemplo Motivacional

Para dar una idea de como funciona la técnica veamos un ejemplo simple, usando el algoritmo 1 y su CFG asociado en la figura 2.8.

Algorithm 1: motivationalExample

```
Result: Integer
Input: int x, int y
1 int z = 2 * y;
2 if z == 3 then
3 |   return 0;
4 end
5 if z == 4 then
6 |   return 1;
7 else
8 |   ERROR;
9 end
```

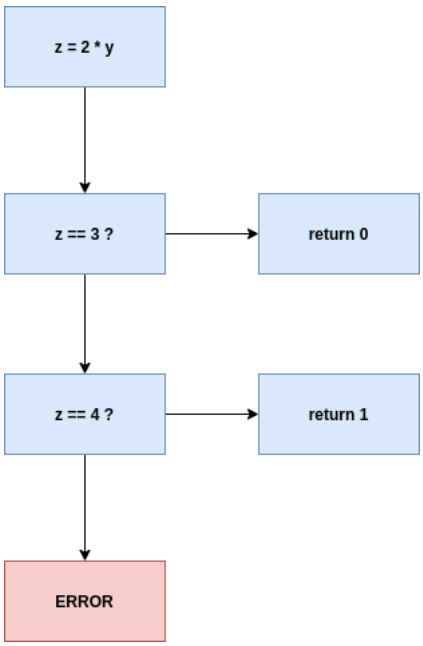

Fig. 2.8: CFG asociado al algoritmo 1

Al principio se crean los valores simbólicos de los inputs de entrada y se los asocia a las variables respectivas en la memoria simbólica σ:

$$\sigma = \{x = x_0, y = y_0\} \text{ y } PC = \emptyset$$

Al ejecutar la línea **1** se actualiza el estado simbólico agregando el valor z.

$$\sigma = \{x = x_0, y = y_0, z = 2 * y_0\} \text{ y } PC = \emptyset$$

El valor de z ahora predica sobre los elementos simbólicos de los inputs correspondientes. Luego pasamos a la línea **2** donde, si bien no se actualiza σ, se crea una constraint y se la agrega a la PC, quedando

$$\sigma = \{x = x_0, y = y_0, z = 2 * y_0\} \text{ y } PC = 2 * y_0 == 3$$

Con esto, consultamos con un constraint solver si la condición es feasible, en este caso no existe ningún y_0 perteneciente a lo enteros que satisfaga esa condición con lo que la rama **true** de ese branch resulta ser **infeasible**. Con esto el algoritmo hace backtracking sobre esta condición e intenta moverse por la rama **false**, quedando:

$$\sigma = \{x = x_0, y = y_0, z = 2 * y_0\} \text{ y } PC = 2 * y_0 \neq 3$$

Consultando con el constraint solver, devuelve que la PC es feasible (e.g. $y_0 = 0$ lo satisface), con lo que continuamos la ejecución. Luego pasamos a la línea **5**, repitiendo el agregado de una constraint a la PC:

$$\sigma = \{x = x_0, y = y_0, z = 2 * y_0\} \text{ y } PC = 2 * y_0 \neq 3 \wedge 2 * y_0 == 4$$

Siendo esta una PC feasible, continuamos la ejecución hacia **6** llegando a una salida, a esta altura podemos pedirle una solución al constraint solver sobre la PC para obtener los inputs que sigan este camino. En este caso nos dice que $y_0 = 2$ satisface todas las condiciones, dejando x libre para tener cualquier valor. Luego, hacemos backtracking sobre las condiciones de brancheo volviendo a la línea **5** y negándola:

$$\sigma = \{x = x_0, y = y_0, z = 2 * y_0\} \text{ y } PC = 2 * y_0 \neq 3 \wedge 2 * y_0 \neq 4$$

Como resulta feasible, avanzamos hacia la línea **8** donde llegamos a un estado de error, finalmente resolviendo esta PC contra un constraint solver podemos obtener un input tal que produce un error en la ejecución, siendo $y_0 = 1$ el mismo. Por último, haciendo backtracking llegamos a que recorrimos todos los caminos del CFG correspondiente, terminando así la ejecución simbólica.

2.4.2. Desafíos

Mientras que, en teoría, la técnica debería ser capaz de encontrar todos los caminos de ejecución del SUT y sus inputs concretos, este no es el caso en software de nivel industrial debido a varios factores como elementos del entorno (módulos externos, I/O, transferencias de red, etc.), modelos de memoria (Aritmética de punto flotante, elementos no simbólicamente representados como punteros o referencias, etc.) o incluso que la complejidad del software expanda demasiado los posibles estados de exploración. Definimos dos nociones para tener una idea de como estos elementos pueden impactar en la creación de un motor de ejecución simbólica.

Definición 2.4.3. Soundness: Decimos que una ejecución simbólica es sound si prevé la aparición de falsos negativos. E.g. todos los caminos que contengan un bug se garantiza que serán encontrados.

Definición 2.4.4. Completitud: Decimos que una ejecución simbólica es completa si prevé la aparición de falsos positivos. e.g. los inputs que aseguran ir por un camino determinado, lo toman.

Es fácil ver como un motor puede cumplir con estas definiciones pero en una manera parcial. Una clasificación de diferentes desafíos se da a continuación:

- **Explosión en la Cantidad de Estados a Explorar** Elementos de los lenguajes tales como bucles pueden hacer crecer exponencialmente el número de estados de ejecución. Es poco probable que un motor simbólico pueda exhaustivamente explorar todos los posibles estados en una cantidad razonable de tiempo para una pieza de software relativamente pequeña. Por otro lado, software de escala industrial es mucho más complejo tanto en tamaño como en complejidad, haciendo que la explosión de estados sea inmensa, en el peor de los casos el problema puede ser pensado como realizar una búsqueda en un espacio de exploración infinito, haciendo que se reduzca drásticamente la propiedad de soundness.

- **Entorno** Cómo el motor maneja interacciones a través de varios elementos del stack tecnológico es otro desafío interesante. Elementos como librerías externas o servicios del sistema (como llamadas al kernel o servicios del sistema operativo) que pudieran causar efectos secundarios como crear o escribir archivos que luego vayan a ser usados durante la ejecución.

- **Modelo de memoria** Cómo se modelan simbólicamente elementos como *enteros*, *variables de punto flotante*, *aritmética de punteros* o *referencias* (Si se las modela, decisiones como no modelar el heap por completo pueden ser tomadas y estos valores nunca serian registrados durante la ejecución.) y cómo se representan internamente en ese modelado puede impactar significativamente la expresividad del motor. Dependiendo de cómo sean implementados puede impactar la completitud del motor, ya que constraints no tan precisas serian registradas lo cual resultaría en la generación de inputs que podría no llevar a la ejecución por el camino esperado, en este caso se dice que hubo una **divergencia**.

  Definición 2.4.5. Divergencia: Sea PC una PC recolectada de un SUT P y sea I el input generado por un SMT solver para recorrer PC en P. Sea PC^* la PC recolectada al ejecutar P con inputs I, decimos que una divergencia ocurre cuando PC no es un prefijo de PC^*.

  Una definición informal es tan simple como decir que la ejecución del programa tomo un camino distinto del que se esperaba que tomara.

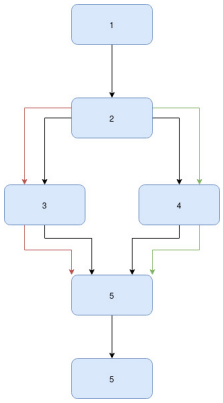

Fig. 2.9: Ejemplo de una divergencia en un subgrafo de un CFG. En la figura, flechas verdes representan el camino esperado y las rojas el camino ejecutado.

Godefroid et al. [35] observó que el porcentaje de divergencias era alrededor del 60 %, lo cual resulta en casi dos tercios de todos los inputs generados. Además, varios elementos pueden resultar en una divergencia, Chen et al.[36] hicieron un análisis usando CREST[10] sobre el lenguaje C con resultados interesantes como, parafraseando: *"Los tres patrones mas prevalentes, los cuales son excepciones, llamadas externas y types casting resultaron en aproximadamente el 82 % de las divergencias."*

- **Tiempos de resolución de constraints** La resolución de constraints es todavía uno de los principales desafíos de la técnica, principalmente por la cantidad de tiempo que consume comparada con el resto de los elementos de la ejecución simbólica. Dependiendo de cómo son explorados los caminos puede haber una mejora en performance, ideas como usar una cache para soluciones de caminos ya resueltos, descartar constraints no importantes de la PC y normalizar la PC ayudan en reducir el tiempo de ejecución [37]. Es importante notar que, como la ejecución está siendo simulada paso a paso, el SMT solver debe ser ejecutado cada vez que una instrucción de brancheo es encontrada para poder saber si explorar en profundidad por ese camino es feasible, lo cual resulta en varias llamadas al solver por cada camino que se ejecute en el SUT.

2.5. Ejecución Simbólica Dinámica

Una ejecución simbólica requiere que el motor simbólico "simule" una ejecución del programa, este acercamiento (si bien desafiante) requiere muchas consideraciones tanto para su implementación como para su análisis como vimos en la sección previa. Una idea por primera vez explorada en la herramienta DART[8], propone reusar el entorno de una ejecución concreta manteniendo en paralelo un estado simbólico a la ejecución, esto es, la ejecución concreta guía a la ejecución simbólica (Esta técnica también se llama ejecución concólica como un acrónimo a *CONCreta + simbOLICA*).

Esta idea simplifica el problema dado que no es necesario "simular" por completo la ejecución del programa, ya que la ejecución concreta provee a la ejecución simbólica de la información necesaria. Además, varias secciones que previamente no eran representables por la ejecución simbólica como módulos externos, elementos de I/O ahora pueden ser ejecutados exclusivamente de manera concreta mientras que el resto del programa puede

ser ejecutado tanto de manera simbólica como concreta manteniendo el estado simbólico σ y un estado concreto M tal que dado una variable v, $\sigma[v]$ contendrá su estado simbólico y $M[v]$ contendrá su estado concreto.

Existen varias ideas para hacer esto en Java, por ejemplo CATG[16] instrumenta el código usando la API de instrumentación de Java [28] para realizar un log de la traza del bytecode del programa y analizarlo luego. Otras herramientas como JCUTE[15] instrumenta el código usando el framework SOOT [38] para manipular el estado simbólico añadiendo de manera dinámica instrucciones en el bytecode a medida que el programa se va ejecutando para "mirar" a la ejecución concreta, en particular Evosuite realiza algo parecido y lo discutiremos en el capítulo 3. Herramientas como PEX[13] para .NET y DSC[39] llaman a esta idea un **intérprete sombra**, el cual espeja la ejecución concreta y simboliza su comportamiento. Otras herramientas como JDART[18] y JFUZZ[17], si bien realizan DSE, tienen una dependencia subyacente en Java Path-Finder[19] el cual ejecuta el programa en una VM personalizada.

2.5.1. Pérdida de completitud

Mientras que el programa es ejecutado concretamente, constraints son recolectadas en cada condición de brancheo. Una contra es que, si el modelo de memoria es incompleto, varios elementos de ese branch no se verán reflejados en la constraint recolectada (ya que no son modelados por el motor). En estos casos el motor simplemente concretiza los elementos no representables simbólicamente y los usa como literales. Esto reduce la completitud del motor además de poder generar divergencias mas adelante. Por ejemplo tomemos el siguiente método

Algorithm 2: pathDivergence

```
   Result: Integer
1  if x == hash(y) then
2  |  ...;
3  |  if y == -10 then
4  |  |  return -1;
5  |  end
6  end
7  return 0;
```

Como la función de hash no puede ser simbólicamente recorrida, se concretiza su valor de retorno. Supongamos que en algún punto se genero un test case usando DSE con los inputs *x = 10, y = 496* tal que *hash(496) = 10*, este explora la rama *true* del primer if. Como la función de hash no es recorrida simbólicamente se genera la constraint

$$x == 10$$

Luego, el segundo if es alcanzado y la constraint *y != -10* es agregada, lo que resulta en la PC:

$$x == 10 \wedge y! = 10$$

Luego de negar la última constraint obtenemos:

$$x == 10 \wedge y == 10$$

La cual es satisfacible y el test case con los inputs *x = 10, y = 10* se genera. Ahora, suponiendo que *hash(10) = 23*, al ejecutar ese test case el resultado sera -1 en lugar del 0 esperado, ya que se tomó la rama false en el primer if, resultando en una divergencia.

3. NUEVO MÓDULO

En el capítulo previo, dimos los conceptos teóricos requeridos para entender el resto de esta tesis. En este capítulo una idea de alto nivel sobre como se desarrolló el módulo de DSE será explicada así como el nuevo algoritmo de exploración del mismo.

Fig. 3.1: Etapas de generación de test suites de Evosuite.

En particular, Evosuite realiza 3 etapas de ejecución a la hora de generar una test suite como se puede ver en la figura 3.1. El módulo de DSE representa una estrategia particular agregada en la etapa *TestCaseGeneration*, la cual discutiremos en este capítulo. La etapa *StaticAnalysis* analiza el SUT y genera información relevante del mismo como dependencias, representación en CFG, etc. por último la etapa *TestSuitePostProcessing* realiza transformaciones como minimización del test suite[1] entre otras. No vamos a entrar en detalle sobre como estas últimas dos etapas funcionan pero algunas menciones puntuales se harán mas adelante.

3.1. Diseño de Alto Nivel

En la sección 2.4.2 discutimos los desafíos en ejecución simbólica, un motor de DSE por lo general esta diseñado de tal manera que resuelva esos problemas de forma lo mas aislada posible. i.e. herramientas como JDART[18] y JFUZZ[17], como se puede ver en las figuras 3.2 y 3.3 respectivamente, diseñaron módulos separados para cada responsabilidad. En JDART el módulo *explorer* soluciona la explosión de estados de exploración y resolución de constraints, una sección interesante es el módulo *JConstraints* dentro de esta, el cual en realidad es una librería externa[40] para hacer de interfaz contra SMT solvers. Finalmente el módulo *executor* hace de interfaz contra JPF, es importante notar que este módulo está diseñado de manera que JPF pueda ser intercambiado con otros ejecutores concólicos[18].

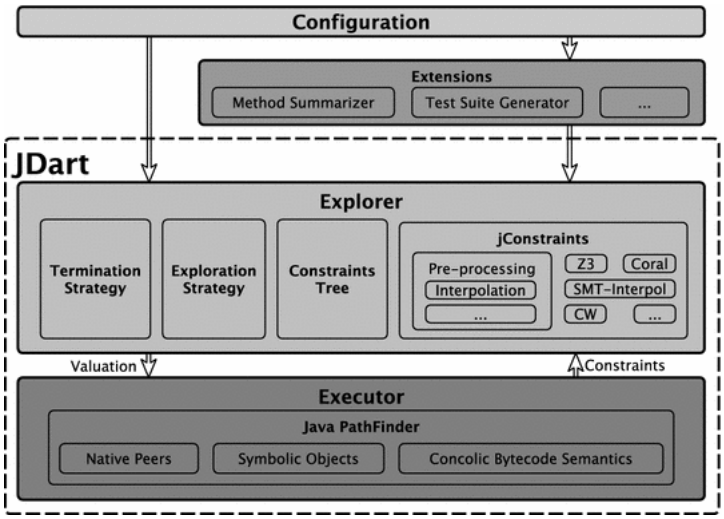

Fig. 3.2: Arquitectura de JDART.

De manera similar, JFUZZ[17] realiza la exploración en el módulo *Fuzzer* y tiene dos módulos mas para resolver la ejecución concólica y la resolución de constraints con un SMT solver.

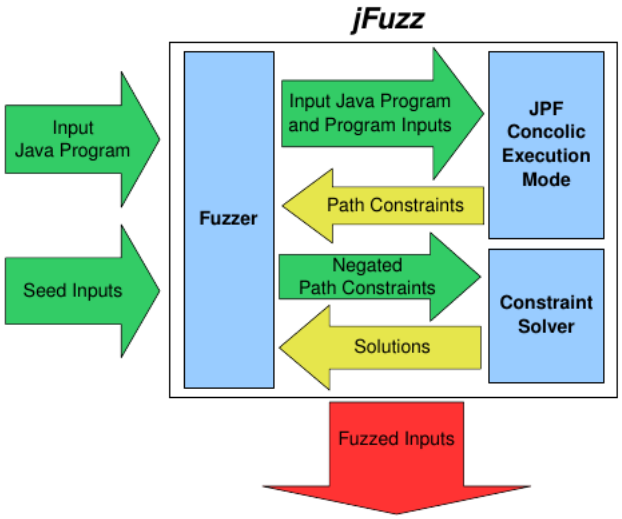

Fig. 3.3: Arquitectura de JFUZZ.

Evosuite utiliza un acercamiento similar en su diseño, como se ve a continuación

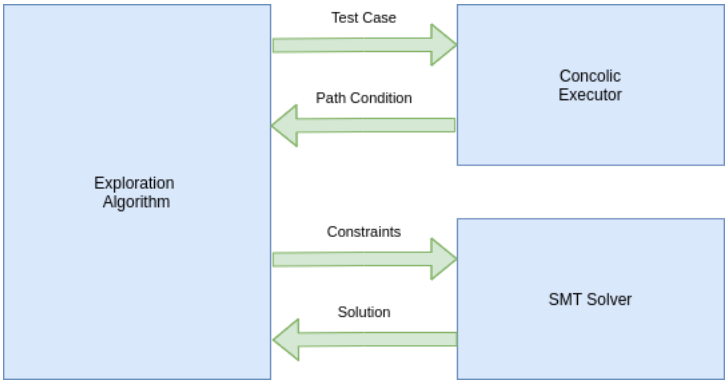

Fig. 3.4: Diseño del módulo de DSE de Evosuite.

En este diseño, *Exploration Algorithm* genera test cases basándose en soluciones dadas y decide cual explorar a continuación. *Concolic Executor* realiza la ejecución concólica, el mantenimiento de la memoria simbólica y la construcción de la PC y, por último, *SMT solver* transforma el conjunto dado de constraints en una query SMT, la cual es enviada a un SMT solver para ser resuelta y, mas tarde, parsear su respuesta para devolverla al módulo de exploración. En este capítulo hablaremos de *Exploration Algorithm*.

Antes de discutir el módulo en detalle, estas son algunas consideraciones para esta sección:

- Los diagramas en general serán dibujados usando diagramas similares a UML[41].
- Dado que Evosuite contiene varias capas de lógica, las cuales resultaran en una cantidad de diagramas y explicaciones considerables si las fuéramos a explicar en profundidad, por simplicidad estas secciones serán simplificadas y se alienta al lector a revisar el código fuente de Evosuite[42] para detalles específicos.

3.2. Exploration Algorithm

Como vimos antes, hay 2 responsabilidades principales para este módulo (1) dada una PC explorada, (a) generar nuevos caminos a explorar y decidir cual explorar a continuación, (b) realizar optimizaciones en las constraints para acelerar el proceso de resolución de las mismas y (2) transformar una solución del solver en un nuevo test case.

Existen varias técnicas para explorar, Liu et. al. [43] realizaron un estudio sobre las técnicas actuales tales como DSF y BFS como las técnicas clásicas por un lado y técnicas basadas en heurísticas como búsqueda guiada por fitness y búsqueda generacional por otro. Esta implementación esta basada en la búsqueda generacional propuesta en SAGE.

3.2.1. Búsqueda Generacional

Originalmente propuesta en Godefroi et. al. [35], esta búsqueda tiene cuatro méritos:

1. Es capaz de explorar el árbol de ejecución de grandes aplicación con un espacio de posibles inputs grande y muy profundos caminos.
2. Maximiza el número de nuevos tests generados en cada ejecución de DSE, mientras que evita cualquier test case redundante en la búsqueda.

3. Maximiza la cobertura de código tan rápido como sea posible para revelar vulnerabilidades mas rápido

4. Es resistente a divergencias, i.e. siempre que una divergencia ocurra, es capaz de recuperarse.

El algoritmo principal es bastante estándar, la idea general es generar un input inicial, agregarlo a una *work list* que va a contener todos los inputs generados pendientes de recorrer y luego concólicamente ejecutar cada uno de ellos mientras se van generando nuevos que mas tarde serán puntuados y agregados a la *work list* como se puede ver en el algoritmo 3. Algunas distinciones importantes son

1. La *work list* está ordenada por puntaje, esto es los inputs con puntajes mas altos serán explorados primero.

2. La función *score* da puntaje según la cobertura de bloques incremental que provee.

3. Si ocurre una divergencia, el input correspondiente tendrá puntaje 0. Esta penalización es la que ayuda al algoritmo a recuperarse rápidamente de una divergencia, ya que otros no divergentes tendrán prioridad en la *work list*.

Algorithm 3: Generational Algorithm Search

```
   Input: inputSeed
 1 inputSeed.bound = 0;
 2 workList =inputSeed;
 3 Run&Check(inputSeed);
 4 while workList not empty do                          // new children
 5 |   input = PickFirstItem(workList);
 6 |   childInputs = ExpandExecution(input);
 7 |   while childInputs not empty do
 8 |   |   newInput = PickOneItem(childInputs);
 9 |   |   Run&Check(newInput);
10 |   |   Score(newInput);
11 |   |   workList = workList + newInput;
12 |   end
13 end
```

La inicialización se realiza en las líneas 1, 2 y 3 (La función *Run&Check* ejecuta el input para revisar si un error fue encontrado), mientras que la iteración principal comienza en la línea 4. La creación de una nueva generación se realiza en las líneas 5 y 6 para finalmente darles puntajes en las líneas 8 a la 11.

La innovación principal de esta búsqueda es la manera en que se computan nuevos caminos explorables, esto se realiza en el método *ExpandExecution*.

ExpandExecution

Esta función intenta expandir cada constraint en la PC haciendo backtracking sobre las constraints generadas una por una, en lugar de la última como se hace en DFS (Así, maximizando el número de nuevos test cases generados por cada ejecución DSE). La variable

Bound es usada para limitar el backtracking de cada sub-búsqueda por encima del branch que fue generado por su padre, de esta manera se evitan las búsquedas redundantes.

Algorithm 4: ExpandExecution

```
   Input: input
1  childInputs = {};
2  PC = ComputePathConstraint(input)          // symbolic execution;
3  for j=input.bound; j <|PC|; j++ do
4  |  if (PC[0..(j-1)] and not(PC[j]))has a solution I then
5  |  |  newInput = input + I;
6  |  |  newInput.bound = j;
7  |  |  childInputs = childInputs + newInput;
8  |  end
9  end
10 return childInputs;
```

La ejecución simbólica es realizada en la línea 2, mientras que los nuevos inputs son generados en las líneas siguientes. Inputs empezando desde el largo restringido por *bound* son iterados en la línea 3 y las líneas 5, 6 y 7 crean un nuevo input si la PC actual es feasible.

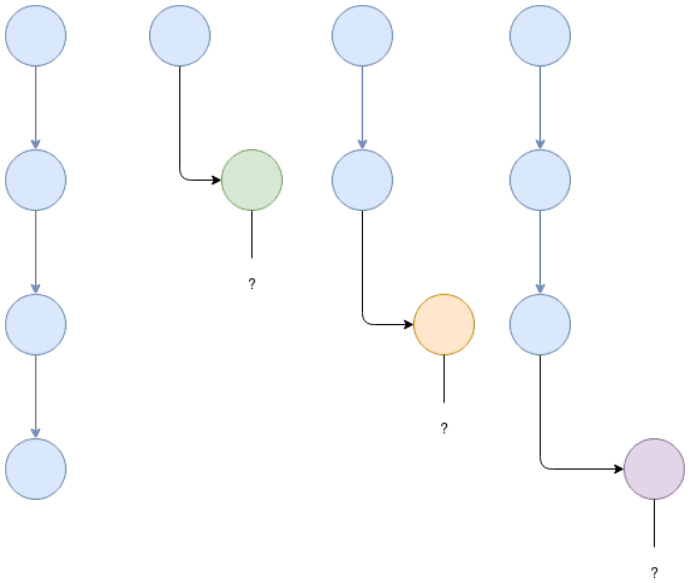

Fig. 3.5: Resultados de *ExpandExecution*. A la izquierda, la PC recorrida. Seguido podemos observar tres nuevas PC generadas negando cada nodo de brancheo de manera que se puede navegar un camino previamente no explorado. En la figura, Los nodos azules representas nodos de brancheo previamente explorados.

A modo de ejemplo sobre como se realiza la expansión, en la figura 3.5 los caminos generados para ser explorados tomados de los prefijos de la PC original son agregados a la *work list*. En la figura 3.6 podemos ver como el espacio de exploración es extendido en el árbol de exploración con nuevos posibles caminos.

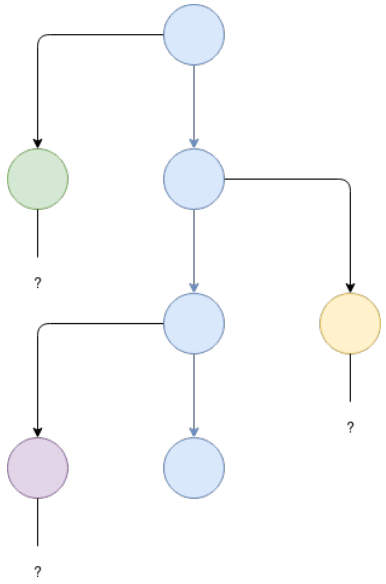

Fig. 3.6: Resultados de *ExpandExecution*. En la figura, los nodos azules representan los nodos de brancheo previamente explorados mientras que el resto son nuevas condiciones creadas para expandir el espacio de búsqueda en el árbol de ejecución.

3.2.2. Nuestro Algoritmo

Nuestra implementación se asemeja al algoritmo 3, pero generalizando algunos elementos. Cosas como la generación del input semilla, como realizar *ExpandExecution* y si el algoritmo debería seguir buscando es lógica que puede ser abstraída del algoritmo principal.

Algorithm 5: explore

```
   Input: SUT
1  seenChildren = {};
2  workList = {};
3  initialTestCase = buildInitialTestCase(SUT);
4  workList = workList + initialTestCase;
5  while ShouldKeepSearching(workList) do
6  |   testCase = getCurrentIterationBasedTestCase(workList);
7  |   TestSuite = TestSuite + testCase;
8  |   PC = ConcolicExecutor.execute(testCase);
9  |   hasPathConditionDiverged = hasDiverged(PC, testCase.originalPC);
10 |   if !shouldSkip(hasPathConditionDiverged, PC, seenChildren) then
11 |   |   seenChildren = seenChildren + PC;
12 |   |   newChildren = generateChildren(PC);
13 |   |   processChildren(workList, testCase, children, hasPathConditionDiverged);
14 |   end
15 end
```

La idea general es parecida, comenzamos creando un test case inicial (input seed) en la línea **3**, luego pasamos a la iteración principal, donde alguna condición para seguir iterando

es revisada en la línea **5**. Una vez dentro del bucle principal, se toma algún elemento pendiente de la work list en la línea **6**, se lo agrega a la test suite en la línea **7** y luego se lo ejecuta concólicamente en la línea **8** a través del módulo *ConcolicExecutor*. Más adelante miramos si la PC resultante divergió de la original en la línea **9**, esto es útil para realizar el chequeo por si la ejecución debería ser salteada en la línea **10** (hablaremos de esto en la sección *Divergencias* mas adelante). En caso de no ser salteada, se pasa a agregarla al conjunto de PC exploradas en la línea **11** y luego se generan hijos para explorar a partir de esa PC en la línea **12** para, finalmente, procesar estos hijos en la línea **13**.

En el algoritmo se pueden ver elementos marcados en rojo, estos representan estrategias implementadas de manera independiente que pueden ser intercambiadas en el algoritmo a decisión del programador. Estas implementan por defecto la misma lógica que el algoritmo de **SAGE**, esto es:

- **buildInitialTestCase:** genera algún input.
- **ShouldKeepSearching:** Devuelve verdadero si y sólo si aún existen elementos en la work list.
- **getCurrentIterationBasedTestCase:** Devuelve el tope de la workList (el elemento con mas puntaje).
- **generateChildren:** Ejecuta el algoritmo 4 (expandExecution).

A continuación nos enfocaremos en el proceso *processChildren*.

processChildren

Este método se encarga de armar los conjuntos de constraints y hacer chequeos contra la cache para, finalmente, llamar al módulo SMT Solver y armar un nuevo testCase basándose en los resultados.

Algorithm 6: processChildren

```
   Input: workList, currentTestCase, children, hasPathConditionDiverged
 1 for child in children do
 2     childQuery = buildQuery(child.PC);
 3     cacheElement = checkCache(childQuery, queryCache);
 4     if !cacheElement found Unsat? then
 5         smtSolution = ∅
 6         if cacheElement found Sat? then
 7             smtSolution = cacheElement.solution;
 8         else
 9             smtSolution = SMTSolver.solveQuery(childQuery);
10             queryCache[childQuery] = smtSolution;
11         end
12         newTestCase = createNewTestCase(currentTestCase, child, smtSolution,
            hasPathConditionDiverged);
13         workList = worklist + newTestCase;
14     end
15 end
```

La idea es continuar con el proceso de los hijos generados en el algoritmo 5. Por cada hijo, se procesan las constraints de su PC en la línea **2** para generar el conjunto de constraints relevantes a utilizar, luego se mira la cache por si un resultado previo puede ser reutilizado en la línea **3**, en caso de no ser satisfacible se pasa al próximo hijo (línea **4**), en caso de serlo se utiliza esa solución (líneas **6** y **7**) y en caso de no haber una solución en la cache se pasa a pedir una solución al módulo *SMTSolver* en la línea **9** para luego guardar su resultado en la cache en la línea **10**. Finalmente, se crea un nuevo test case en la línea **12** (en esta etapa se le asigna un puntaje, recordemos que nosotros usamos cobertura por ramas incremental, en caso de haber una divergencia su puntaje sera 0) y luego agregamos el nuevo test case a la work list en la línea **13**.

En **checkCache** se encuentra implementado el chequeo estándar contra la cache, hablaremos en esto a continuación.

Optimización de Constraints

Varias optimizaciones pueden ser aplicadas a las constraints para ser resueltas mas fácilmente por el solver, Chen et al. [44] hicieron una amplia enumeración de técnicas existentes tales como *constraints caching*, *counter-example cache* y *constraint independence optimization*. Nuestro algoritmo actualmente implementa:

1. **Constraint Normalization:** Esta idea se basa en que, a veces se generan constraints repetidas dentro de la PC. En estos casos, removiendo las versiones repetidas se crea una PC equivalente. Por ejemplo, la PC

   $$(x + y > 10) \wedge (y < 12) \wedge (z > 0) \wedge (y < 12)$$

   Es equivalente a

   $$(x + y > 10) \wedge (z > 0) \wedge (y < 12)$$

   Esta optimización se encuentra implementada en el método **buildQuery** dentro del algoritmo 6.

2. **Irrelevant Constraint Elimination[45]:** Esta basada en la observación de que, por lo general, una rama del programa depende solo de un pequeño número de constraints de la PC. Así, una optimización efectiva es remover de la PC aquellas constraints que son irrelevantes para decidir si la rama actual es feasible o no. Por ejemplo, sea la siguiente PC generada luego de ejecutar concólicamente un test case

   $$(x + y > 10) \wedge (z > 0) \wedge (y < 12)$$

   Luego, negando la última constraint obtenemos una nueva PC

   $$(x + y > 10) \wedge (z > 0) \wedge (y >= 12)$$

Podemos ver que es seguro eliminar la constraint sobre z, ya que esta no puede influir en la satisfacibilidad del branch $y >= 12$.

Hablando formalmente, sea P una PC, sean c_1 y c_2 constraints de P y R una relación sobre el conjunto de constraints de P tal que $c_1, c_2 \in R \leftrightarrow c_1$ y c_2 comparten alguna variable simbólica. De esta manera, siendo c_k la constraint que representa el branch negado ($y >= 12$ del ejemplo) se calcula la *clausura transitiva* de c_k, obteniendo así todos las constraints con elementos "alcanzables" por la constraint que representa al branch negado. En el ejemplo anterior la clausura transitiva resultaría en las constraints $(x + y > 10) \wedge (y >= 12)$.

Un detalle en la implementación es que, a la hora de generar un nuevo test case basándose en los inputs resultantes de usarla, debemos primero clonar el test case original y luego actualizar los nuevos valores. De usar un test case "fresco" (solo con valores por defecto), perderíamos la información sobre las constraints que eliminamos con la optimización y no podríamos asegurar que esos branches tomen el mismo camino en la ejecución.

Esta optimización se encuentra implementada en el método **buildQuery** dentro del algoritmo 6.

3. **Counter-Example Cache[44]:** Esta cache mapea un conjunto de constraints a contra ejemplos (soluciones ya obtenidas por el solver) y ejecuta tres optimizaciones.

   a) Cuando un subconjunto del conjunto de constraints actual no tiene solución, entonces tampoco lo tiene el conjunto de constraints actual. i.e. si las constraints $x > 10 \wedge x < 5$ no tienen soluciones, tampoco lo tendrán las constraints $x > 10 \wedge x < 5 \wedge y - 0$

   b) Cuando un superconjunto de constraints tiene solución, entonces esa solución también satisface al conjunto de constraints original. i.e. $x = 14$ es una solución para las constraints $x > 0 \wedge x < 5$, entonces también los es tanto para $x > 0$ como $x < 5$ de manera individual.

   c) Cuando un subconjunto del conjunto original tiene solución, es probable que esta también sea solución del conjunto original.

   Además, si una constraint es exactamente igual a la que se encuentra en la cache, directamente se usa ese resultado o se toma ese camino como infeasible según sea el caso.

   Esta optimización se encuentra en la implementación por defecto de **checkCache** dentro del algoritmo 6 donde, actualmente, solo (a) se encuentra implementada por dos motivos:

   - Ya que se aplica *Irrelevant Constraint Elimination*, no podemos usar la solución entera en cache de (b) ya que no podemos asumir que el superconjunto de constraints representa el mismo conjunto que el original (previo a usar irrelevant constraint elimination). Por ejemplo, supongamos que el siguiente camino se encuentra en la cache

     $$(x > 10) \wedge (z > 0) \wedge (y + z >= 3)$$

Y supongamos que su solución es $x = 11$, $y = 2$ y $z = 1$. Por otro lado, supongamos que en una iteración futura queremos resolver la constraint

$$(x < 6) \wedge (z > 0) \wedge (y + z >= 3)$$

En particular, luego de aplicar *Irrelevant Constraint Elimination*, nos quedará

$$(z > 0) \wedge (y + z >= 3)$$

La cual resulta ser un subconjunto de la primera, pero de utilizar esa solución estaríamos cambiando el valor de x por uno que no cumpliría con la constraint $(x < 6)$ de la PC sin optimizar. Esta implementación queda como trabajo futuro.

- Como (c) tiene carácter heurístico, requiere un poco más de análisis para ser implementada.

Divergencias

Las divergencias se calculan usando la definición 2.4.5. Luego de ejecutar concólicamente un test case en el método **shouldSkip** en el algoritmo 5, se checkea:

Algorithm 7: shouldSkip

```
Input: hasPathConditionDiverged, PC, seenChildren
if hasPathConditionDiverged then
    if is PC an element of seenChildren? then
        return true;
    end
    if is PC a subset of any element of seenChildren? then
        return true;
    end
end
return false;
```

En el algoritmo podemos ver que, en caso de que se encuentre una divergencia en la PC actual, se revisan las PC recorridas previamente, en caso de que sea exactamente (o un subconjunto de) una ya recorrida, entonces la divergencia resultó en una PC ya visitada y se la descarta.

3.2.3. Implementación

Como vimos al principio de este capítulo, nuestro módulo se encuentra dentro de la segunda fase de generación de Evosuite, en particular el punto de entrada de esta fase se hace a través de alguna estrategia definida, la nuestra sera la clase **DSEStrategy**. Esta es una herencia de **TestGenerationStrategy**, la cual es responsable del setup y ejecución de las estrategias de generación de test suites. Nuestra estrategia le pide una instancia del algoritmo de exploración a **DSEAlgorithmFactory** para luego configurar el algoritmo y ejecutarlo.

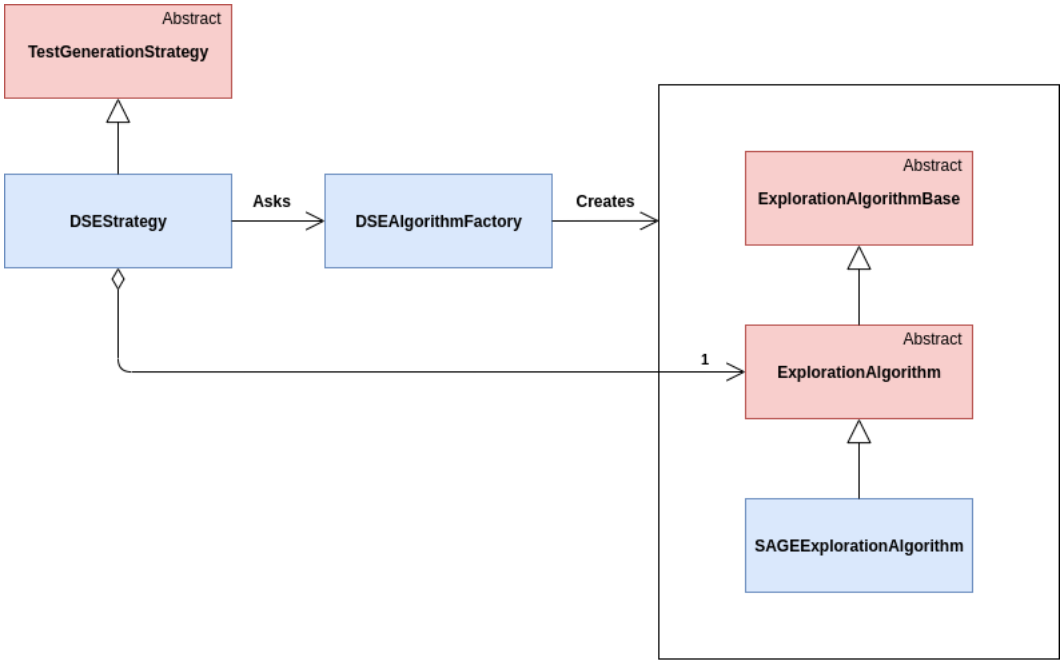

Fig. 3.7: Creación del módulo ExplorationAlgorithm

Los algoritmos 5 y 6 pueden ser encontrados en **ExplorationAlgorithm**, la secciones marcadas en rojo en el algoritmo son modeladas usando un patrón *Strategy*[6] de manera que puedan ser intercambiadas en el futuro para experimentar sobre estos elementos de la exploración. Como trabajo futuro se podrían incorporar nuevas implementaciones (actualmente solo están implementadas las que imitan el comportamiento de SAGE como vimos en la sección anterior) e incluso se podría realizar intercambio de heurísticas durante la ejecución haciendo **hot swapping** entre ellas mientras el algoritmo explora con algún criterio para el intercambio de estrategias. El diagrama de la clase se puede ver a continuación

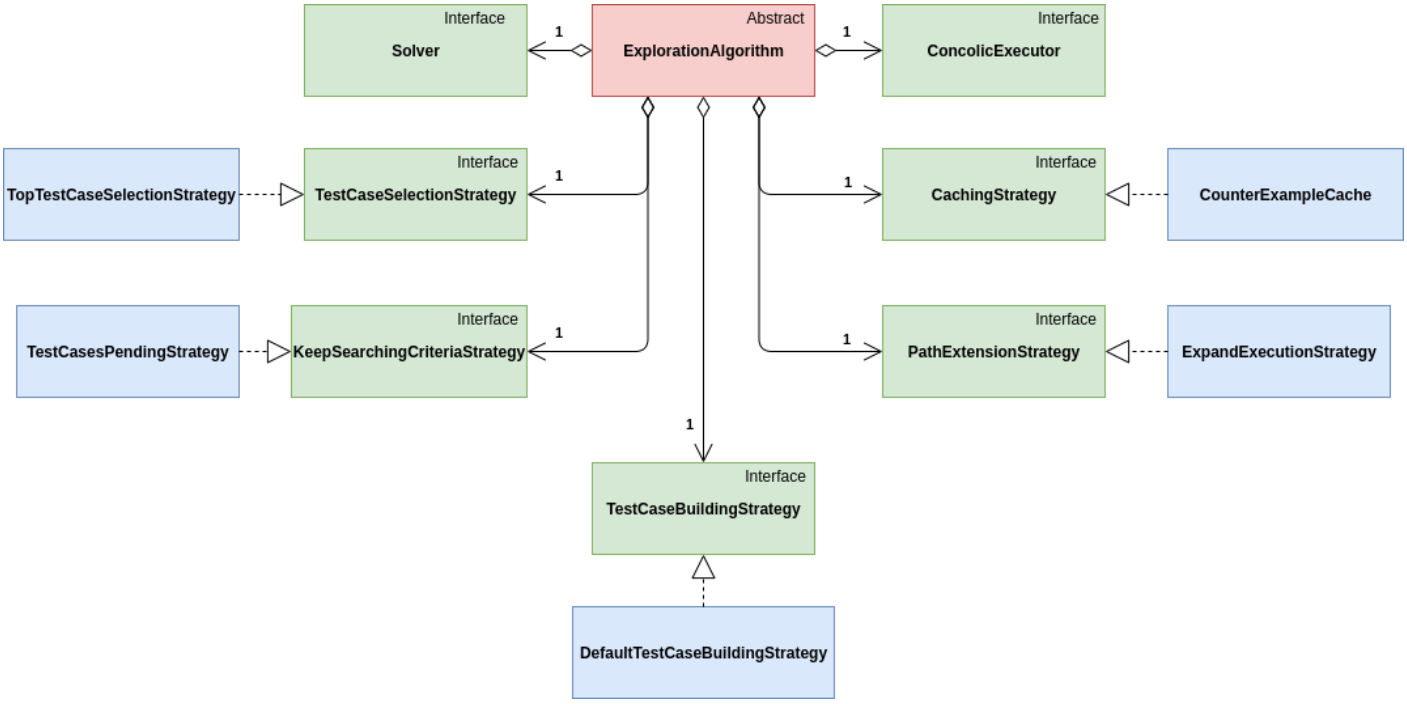

Fig. 3.8: ExplorationAlgorithm class diagram

Para realizar la correspondencia con los algoritmos 5 y 6:

- *buildInitialTestCase* representa *TestCaseBuildingStrategy*
- *shouldKeepSearching* representa *KeepSearchingCriteriaStrategy*
- *getCurrentIterationBasedTestCase* representa *TestCaseSelectionStrategy*
- *generateChildren* representa *PathExtensionStrategy*
- *checkCache* representa *CachingStrategy*

Además, esta clase hace interfaz contra los módulos *ConcolicExecutor* y *Solver*.

Por otro lado, una superclase de *ExplorationAlgorithm*, *ExplorationAlgorithmBase*, provee el cálculo de la cobertura y las condiciones de finalización del algoritmo además de realizar operaciones relacionadas con el test suite, podemos ver en la figura 3.9 como está formada esta clase.

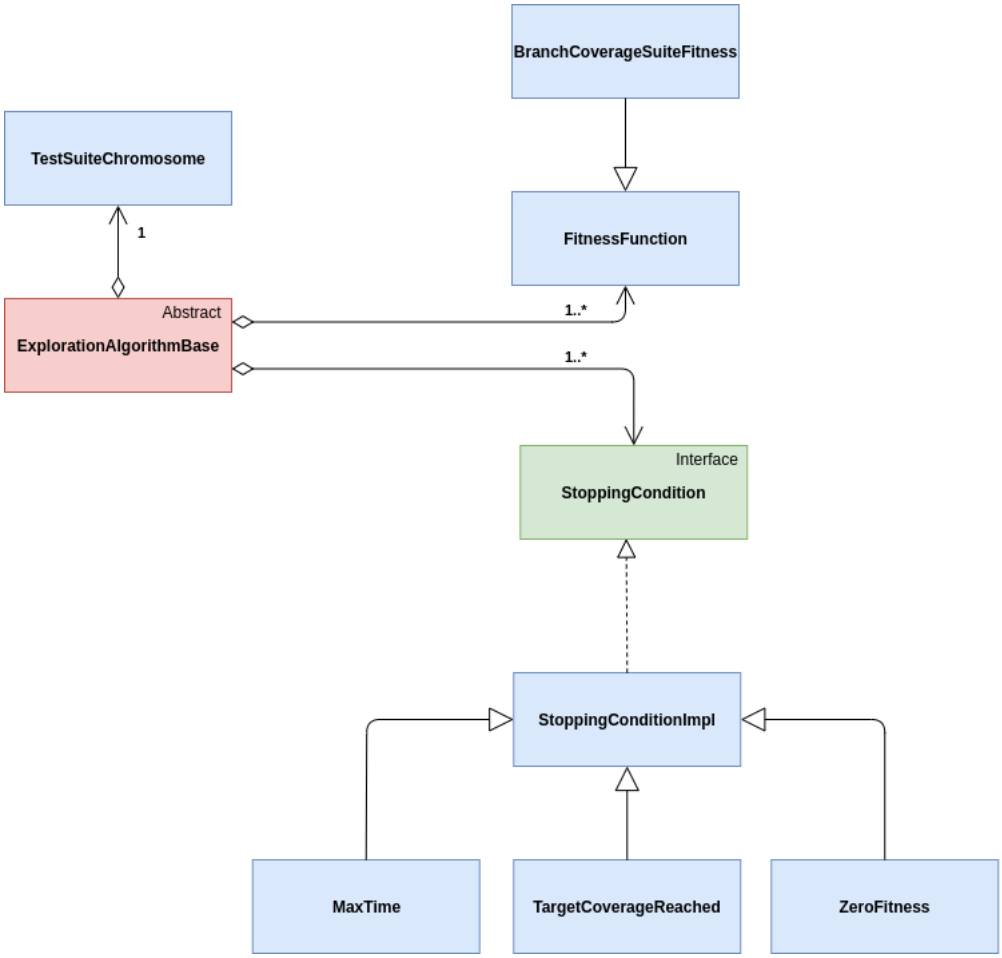

Fig. 3.9: Diagrama de Clases de ExplorationAlgorithmBase

En particular las funciones de fitness y las **StoppingConditions** serán notificadas en cada iteración (Usando un patrón **Observer**[6]) sobre como se actualizo el test suite y luego se las usara para calcular la cobertura y ver si el algoritmo debería dejar de buscar. Además la clase reutiliza **TestSuiteChromosome** del algoritmo genético. Actualmente se usan 3 condiciones para ver si dejar de buscar, cantidad de tiempo de búsqueda, haber llegado a un objetivo de cobertura o haber minimizado fitness en las funciones de fitness. Los tres son configurables.

Consideraciones

- Las **funciones de fitness** fueron tomadas del algoritmo genético usado para SBST dentro de la herramienta, en este módulo se usan para calcular la cobertura generada por el test suite. Este cálculo depende fuertemente de los elementos generados en la etapa de *staticAnalysis* como fue visto en la figura 3.1.

- Como la cobertura usada fue *cobertura por ramas*, nuestra función de puntaje se basa en el calculo de *cobertura por ramas incremental*. Abstraer la función de puntaje queda como trabajo futuro, por el momento depende de como mida la cobertura la función de fitness. Además, el calculo de la cobertura actual se hace de la siguiente

forma: (1) se agrega el nuevo test case al test suite (2) se calcula la nueva cobertura re ejecutando el test suite con el nuevo test agregado (3) se remueve el nuevo test case (4) se vuelve a ejecutar el test suite para recomponer los valores anteriores (y así, recalculando la cobertura). Finalmente, se devuelve la diferencia entre (4) y (2). Esto resulta en la re ejecución completa de test suite dos veces por cada calculo de cobertura incremental y puede ser mejorado mas adelante.

- La **work list** es implementada como una *Cola de Prioridad* donde el campo *score* del test case es usado para ordenar.

4. SOPORTE SIMBÓLICO PARA ARREGLOS

En el capítulo anterior dimos una introducción a la arquitectura del módulo de DSE de Evosuite y hablamos en detalle del módulo responsable de implementar el algoritmo de exploración. En este capítulo hablaremos sobre la resolución de la memoria simbólica σ, haciendo foco en el módulo *ConcolicExecutor* visto en la figura 3.4. En particular daremos una introducción conceptual al trackeo simbólico del motor para pasar a explicar el instrumentado y el contexto donde se realiza para luego pasar a la implementación del intérprete simbólico. Por ultimo, daremos la especificación del soporte para arreglos simbólicos dentro del motor haciendo algunas menciones del módulo *Solver*.

4.1. Concolic Executor

Recapitulando de la sección 3.1, este módulo tiene tres responsabilidades (1) Ejecutar de manera concólica un test case, (2) Mantener actualizado el estado simbólico asociado σ y (3) recolectar la PC resultante. A continuación dividiremos estas responsabilidades en dos partes: (1) la ejecución del test case y su instrumentado y (2) el mantenimiento del estado simbólico y recolección de la PC. Empecemos dando un vistazo general del módulo.

Como vimos en la sección 2.5, información sobre la ejecución es necesaria para crear y mantener el estado simbólico σ, varias ideas existen para esto como generar la traza y luego analizarla como lo hace CATG[16] u observar la ejecución concreta a medida que avanza como JCUTE[15] y DSC[39], nosotros nos centraremos en utilizar la segunda idea. Cabe destacar que tanto el diseño general del instrumentado y la arquitectura de (2) están tomados de la herramienta DSC [39] y luego fueron extendidos.

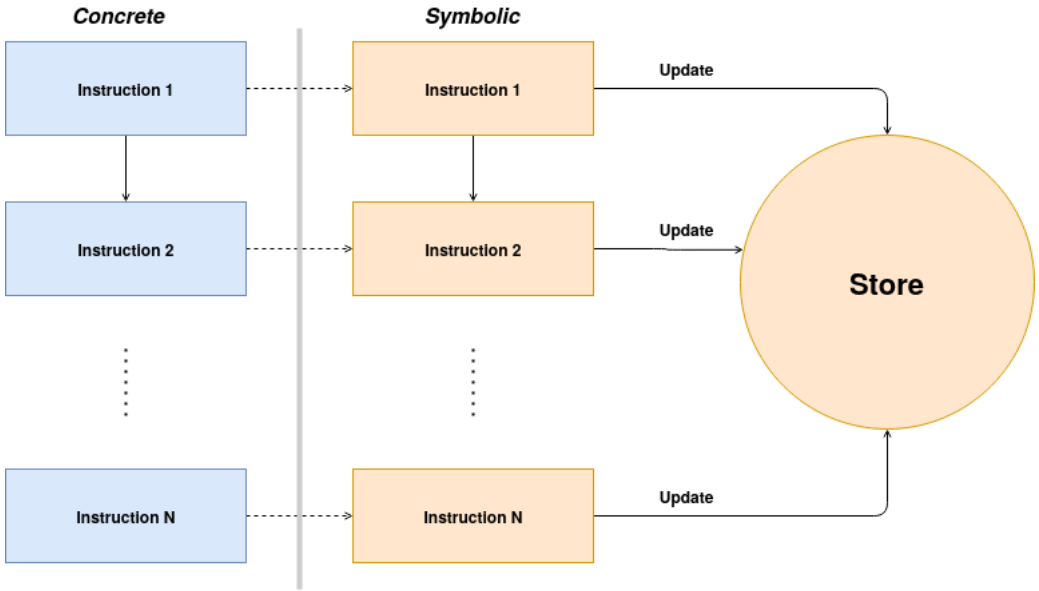

Fig. 4.1: En la figura, a la izquierda el flujo de instrucciones de una ejecución concreta, a la derecha el mapeo simbólico de esa instrucción obteniendo información de su contra parte concreta.

A modo de ejemplo se puede observar en la figura 4.1 como cada instrucción ejecutada genera su contra parte simbólica, la cual obtiene la información necesaria de la instrucción concreta para actualizar la memoria simbólica asociada σ. En particular, las instrucciones simbólicas se irán intercalando en la ejecución, siendo el resultado final algo similar al de la figura 4.4. Notar que es meramente a modo de ejemplo, la instrucción simbólica puede ser ejecutada antes o incluso separarse en varias secciones que se ejecuten antes y después de la concreta.

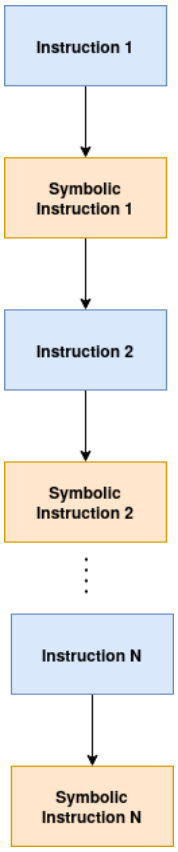

Fig. 4.2: Ejecución intercalada de instrucciones concretas e instrucciones que actualizan el estado simbólico.

Esto es lo que en los preliminares llamamos el *interprete sombra* que, literalmente, va interpretando las instrucciones del programa a medida que estas se van ejecutando. Gracias a esto podemos hacer un espejado de la ejecución y mantener un trackeo simbólico, lo que nos define dos desafíos para su implementación (1) el instrumentado de las instrucciones y (2) la creación de un interprete simbólico que lleve adelante el mantenimiento del estado simbólico σ de la ejecución que se mapean respectivamente a las dos divisiones vistas al principio de esta sección.

4.1.1. El Instrumentado

Recapitulando de la sección 2.2.1, la JVM utiliza class loaders para cargar de manera dinámica el byte-code de las clases en el entorno de ejecución. En este punto podemos manipular el byte-code generado de manera de agregar nuevas instrucciones intermedias. Esta

ejecución está dada dentro del contexto de un test case el cual, entre otras cosas, ejecutará elementos del SUT, recordemos del capítulo anterior que lo que ejecutamos concólicamente son casos de prueba unitarios y no solo el SUT.

Estructura de un test case

En particular, Evosuite modela un test case como una secuencia de statements[46] donde, por lo general, el método bajo prueba del SUT representara un statement de invocación utilizando variables previamente definidas.

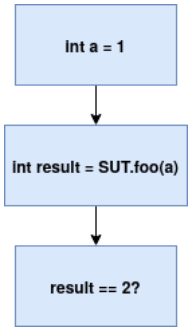

Fig. 4.3: Representación de un test case simple.

Luego, a la hora de ejecutar concólicamente el test case, se iteran los statements y se los ejecuta concólicamente de manera independiente pero manteniendo el mismo scope de ejecución, de esa manera simulando el entorno real de ejecución. No entraremos en detalle sobre esto y lo veremos a modo de caja negra.

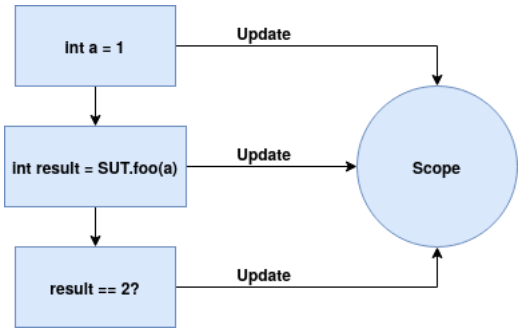

Fig. 4.4: Representación de un test case simple con scope.

Estrategia de instrumentado

A modo de ejemplo, recordemos que en Java los elementos de una clase también son objetos. En particular cada método de una clase posee un objeto asociado, de manera que al intentar llamar a un método en particular, se utilizará su class loader para ubicar su byte-code dentro del entorno de ejecución, Así, al crear un statement representando un método, este tendrá como elemento interno una referencia al método, esto sera útil para hacer el instrumentado del mismo.

En particular, para realizar el instrumentado utilizamos el framework de manipulación de byte-code *ASM*[29]. No entraremos en detalle sobre como funciona el framework, pero

a continuación veremos la idea general. Nuestra estrategia será solamente mirar las instrucciones que se van ejecutando (no modificaremos su contenido) para luego ir agregando instrucciones intermedias que espejen de manera simbólica como vimos al principio del capítulo. Para esto primero intercambiamos el class loader del objeto a instrumentar con uno personalizado como se ve a continuación.

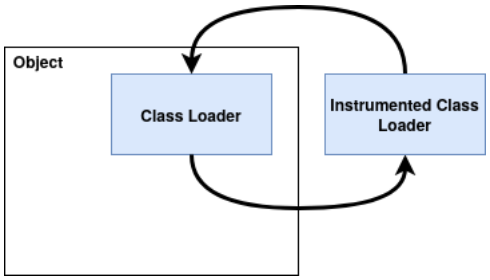

Fig. 4.5: Intercambio de classloader.

Las instrucciones wrappeadas harán primero una llamada al framework donde podremos, entre otras cosas, ver que instrucción se ejecutó y su contexto. En este paso simplemente hacemos de proxy contra el entorno simbólico, enviándole la información que necesite para emular la instrucción simbólicamente agregando instrucciones de callback hacia el interprete simbólico como se puede ver en la figura 4.6 de manera similar a la que vimos previamente.

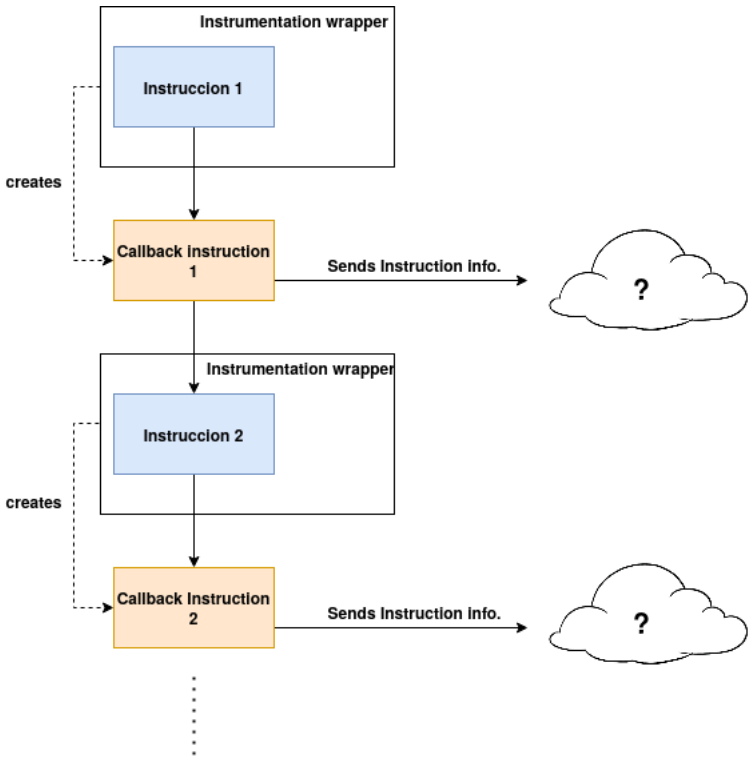

Fig. 4.6: Agregado de callbacks en el instrumentado.

De esta manera, el instrumentado en ASM resulta en la primera capa de emulación

simbólica, siendo la próxima la de interpretación simbólica de la instrucción wrappeada, esto lo veremos en la próxima sección.

4.1.2. Interpretación simbólica

En la sección anterior vimos como realizar el instrumentado del código y agregamos las funciones de callback para realizar el trackeo simbólico correspondiente. A continuación haremos foco en resolver el desafío (2) provisto en la sección anterior. Haciendo foco en lo que pasa después de la llamada al callback en la figura 4.6, necesitamos de alguna manera modelar simbólicamente lo que pasa dentro del entorno concreto, para esto modelamos una VM simbólica que imitara la arquitectura vista en la sección 2.2.1. En particular dividiremos el modelado en dos partes: (1) El conjunto de instrucciones (ISA) del bytecode de java dentro de las VM junto con su semántica y (2) la representación simbólica del entorno de ejecución similar al visto en la sección 2.2.1), en esta parte modelaremos el modelo de memoria de manera simbólica. Empecemos con el primero.

Las VMs Simbólicas

Luego de realizar el callback con la información sobre la instrucción ejecutada y su contexto, necesitamos dar semántica a las instrucciones resultantes pero de manera simbólica. Para facilitar la distribución de responsabilidades dentro del conjunto de instrucciones, armamos varias VM y repartimos las instrucciones según categorías, estas son:

- **CallVM:** Responsable de las instrucciones para llamadas explícitas entre procedimientos (i.e. INVOKEDYNAMIC, INVOKESTATIC, etc.).
- **JumpVM:** Responsable de las instrucciones de brancheo (i.e. IFGT, TABLESWITCH, etc.)
- **HeapVM:** Responsable de las instrucciones relacionadas con el manejo del heap y los campos estáticos (i.e. NEWARRAY, NEW, IALOAD, etc.).
- **LocalsVM:** Responsable de las instrucciones relacionadas con el manejo explícito del stack (i.e. ALOAD, LDC, etc.).
- **ArithmeticVM:** Responsable de las instrucciones que usan elementos del stack para realizar una operación y colocar ese resultado en el stack, sin manejo del heap o variables locales(i.e. LADD, DDIV, etc.).
- **SymbolicFunctionVM:** Responsable de las representación simbólica de las invocaciones no simbolizables. Como discutimos en la sección 2.4.2, uno de los desafíos de la técnica es el modelado del entorno, problemas como la ejecución de código no instrumentado provee perdida de completitud por no poder modelar simbólicamente ese bloque de la ejecución. La estrategia aplicada es tener una librería propia de modelado simbólico para estos elementos. Por ejemplo, para las clases de la librería estándar de Java como un String, una lista o un mapa, de esta manera proponemos nuestro modelado simbólico personalizado sobre esta clase sobre asunciones en las pre y pos condiciones del mismo, y con estas actualizar el estado simbólico de manera acorde. Al momento de escribir este documento se soportan parcialmente las clases: **Long**, **Byte**, **Math**, **Float**, **Short**, **Double**, **String**, **Reader**, **Integer**,

Boolean, **Character**, **TextRegex**, **BigInteger**, **StringReader**, **Regex.Pattern**, **StringBuffer**, **Regex.Matcher**, **StringBuilder** y **StringTokenize**.

- **OtherVM:** Instrucciones extras que no entran en las otras categorías(i.e. MONITORENTER, MONITOREXIT, etc.).

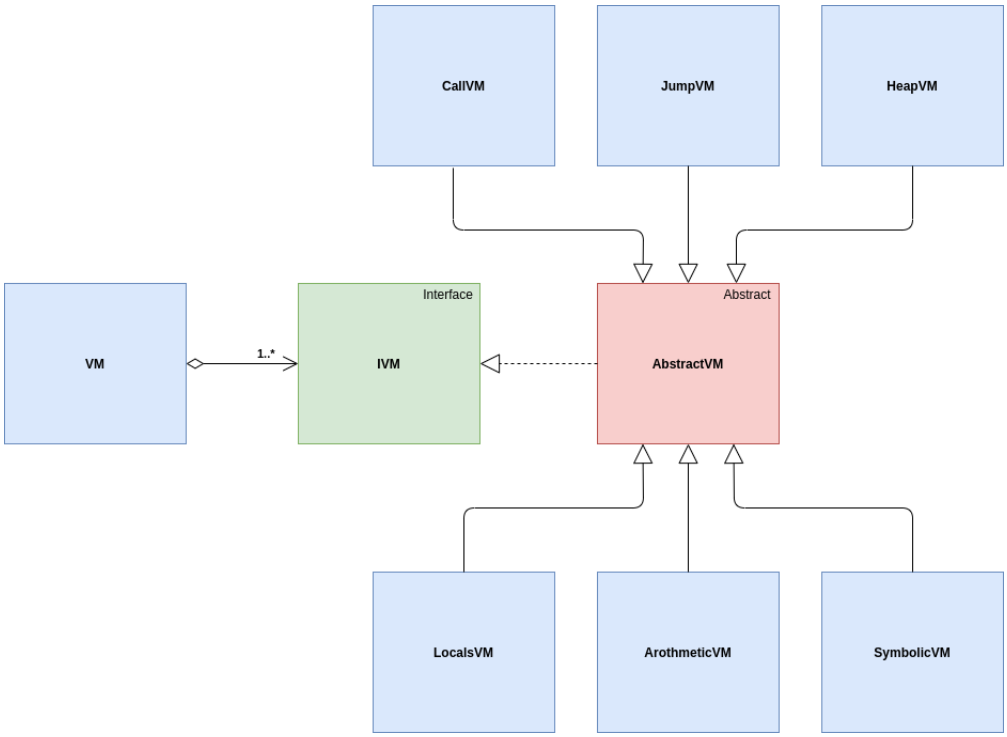

Fig. 4.7: Diagrama de clases de las VMs.

El diseño se puede ver en la figura 4.7, las VMs son herencia de una VM abstracta, la cual implementa una interfaz general con todas las instrucciones. Por separado, una VM principal es la que recibe los llamados desde las instrucciones de callback que creamos en la sección anterior a modo de ”evento” y las redirige a las VMs subscriptoras (Usando un patrón *Observer*) como se puede observar en la figura 4.8. De esta manera las VMs que tengan implementadas las instrucciones correspondientes sabrán como operar sobre ellas. Esta implementación permite extender las VMs según sea necesario mas adelante o incluso crear nuevas de acuerdo a como el lenguaje evolucione o si se desean implementar funcionalidades extras. No entraremos en detalle sobre la semántica operacional y los detalles particulares de todas las instrucciones implementadas hasta el momento pero en la próxima sección daremos una idea general de que elementos están actualmente soportados.

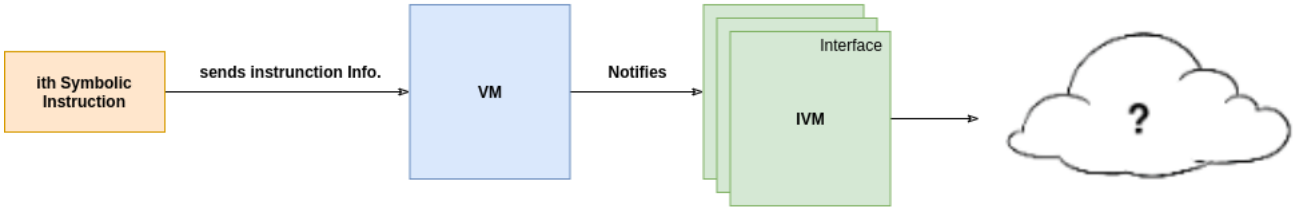

Fig. 4.8: Redirección de la información concreta a la VM simbólica.

Con esto completamos el segundo paso de emulación simbólica, hasta el momento instrumentamos la ejecución para agregar instrucciones de callback hacia las VMs y notificamos a las VMs correspondientes que una instrucción concreta fue ejecutada. A continuación veremos la ultima capa donde representaremos el modelo de memoria, definiendo un entorno simbólico σ junto con la recolección de constraints para generar la PC.

El Entorno Simbólico σ

El objetivo del entorno es replicar de manera simbólica el comportamiento del modelo de memoria de la JVM, esto es, provee creación y mantenimiento de los contenidos simbólicos del interprete simbólico. Recapitulando de la sección 2.2.1, mas en particular de la figura 2.6, el entorno simbólico emulará el comportamiento del heap y del stack de la misma manera que lo hace la ejecución concreta. Como se puede ver en la figura 4.9.

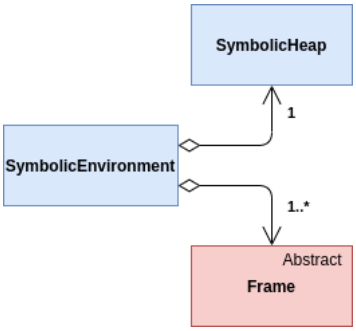

Fig. 4.9: Diagrama de clases del entorno simbólico. En el diagrama, el entorno contiene una instancia del heap simbólico y un conjunto de frames representando el stack de llamadas.

El Stack de Ejecución

No entraremos en detalle sobre las implementaciones de frames particulares pero veremos como se compone la clase abstracta. En particular, modela el stack de operandos y las variables locales que use el scope correspondiente.

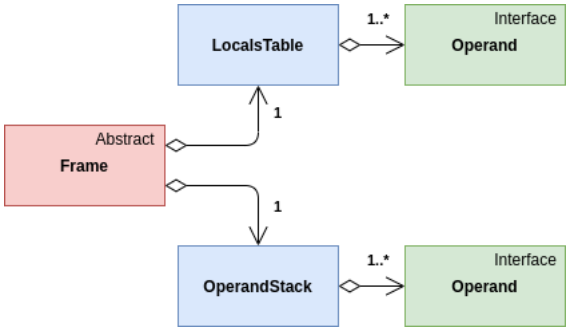

Fig. 4.10: Diagrama de clases de un frame. En el diagrama, un frame contiene una instancia del stack de operandos y de la tabla de locales. A su vez, cada uno contiene uno o más operandos.

Ahora haremos foco en los operandos, recordemos de la sección 2.2.1 que los operandos de la memoria interna pueden caer en dos categorías, los que usan un slot o dos. Los categorizaremos de la misma manera como se ve en la figura a continuación.

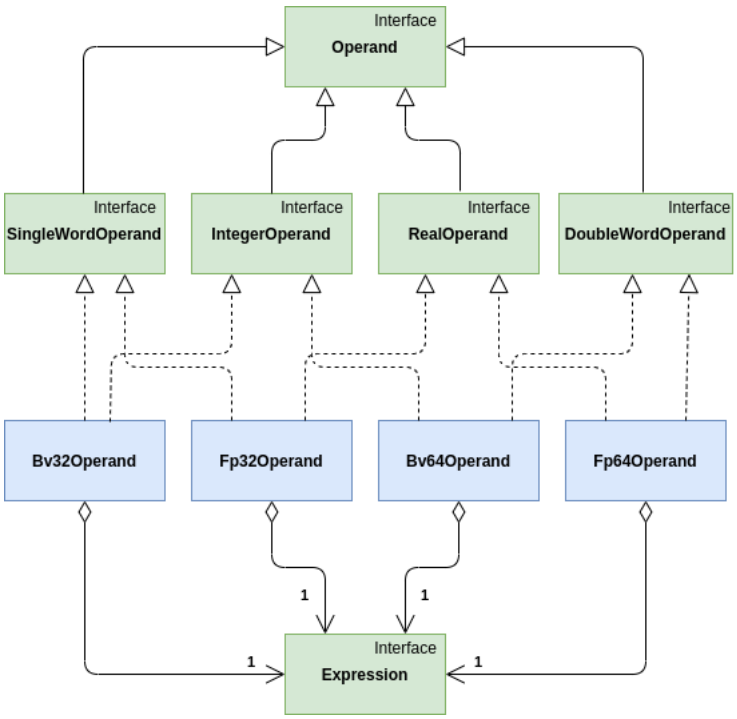

Fig. 4.11: Diagrama de clases de los operandos soportados por el entorno simbólico. En el diagrama, se dividen en dos tipos (Bv = enteros y Fp = de punto flotante)

.

Estos contendrán las expresiones que representaran los valores simbólicos de la ejecución, hablaremos de su implementación mas adelante en la sección 4.1.3.

El Heap Simbólico

El heap esta dividido en dos secciones disjuntas separando los tipos de objetos dinámicos que maneja, *los estáticos* y *los instanciados*. Para el primero solamente se mantiene el estado simbólico de los campos estáticos mientras que para el segundo se mantendrán tres elementos: Las *referencias simbólicas*, los *campos de instancias* y los *Arreglos*. Si bien los últimos dos subyacentemente tienen sus referencias, estos son disjuntos, esto es, los arreglos y las instancias de los objetos no se relacionan dentro del heap. Esta observación sera útil a la hora de modelar de manera simbólica los arreglos tal como explicaremos mas adelante.

Recolección de constraints

A la hora de ejecutar alguna instrucción de brancheo queremos recolectar las constraints asociadas dentro de una PC, para esto creamos un objeto recolector el cual mantendrá el estado de la PC y se ira actualizando desde las VMs en caso de que la instrucción actualice la PC. El objeto **PathConditionColector** mantiene una lista de nodos de brancheo recorridos con las constraints asociadas, el cual es levantado al finalizar la ejecución simbólica para retornar la PC a *ExplorationAlgorithm*.

4.1.3. Gramática Simbólica

En la sección anterior introdujimos el interprete simbólico y desglosamos sus responsabilidades para entender como funciona la infraestructura de nuestro modelo de memoria sin entrar en detalle sobre el manejo de las expresiones simbólicas en sí. Recapitulando de la sección 2.4, las expresiones simbólicas están libres de cuantificadores y pertenecen a alguna teoría T, en particular están expresadas en términos de los inputs. El hecho de que pertenezcan a alguna teoría conocida nos da información sobre las construcciones bien formuladas de las expresiones, con esto podemos definir una gramática formal sobre las mismas. Definir una gramática nos brinda la capacidad de aplicar varias técnicas conocidas para modelarlas dentro del interprete simbólico y, además, generar consistencia entre el modelo que usara el interprete para generar las expresiones y el SMT solver para resolverlas, lo cual será muy útil a la hora de traducir una expresión simbólica a una query SMT.

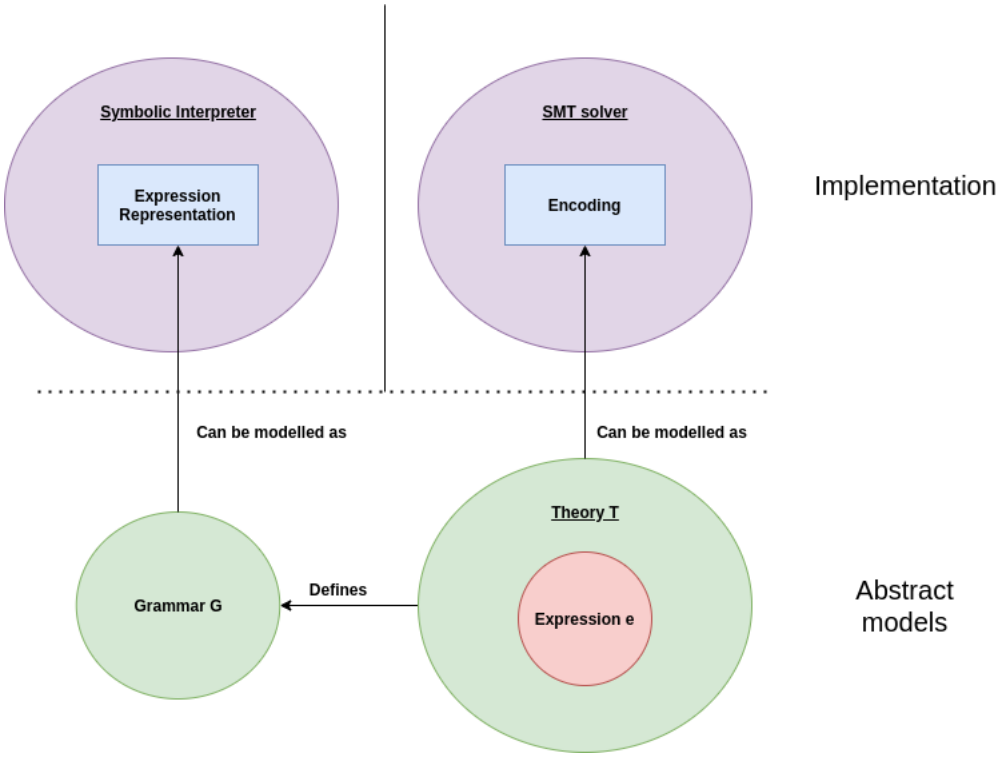

Fig. 4.12: Representaciones de una Teoría y sus implementaciones. En el diagrama, debajo de la linea punteada nos encontramos en el mundo abstracto donde una teoría contiene alguna expresión que se representara a través de una gramática durante la ejecución simbólica. Arriba de la linea punteada nos encontramos en la implementación tanto del interprete simbólico como del SMT solver, donde respectivamente se definirá un encoding para esa teoría y un conjunto de expresiones simbólicas para su gramática asociada.

.

Un simple ejemplo

Recordemos de la sección 2.3.1 que SMT-lib especifica las descripciones sobre las teorías que un SMT solver puede implementar dentro del estándar. A modo de ejemplo, la descripción parcial de la teoría de enteros tiene las siguientes funciones (sin incluir las comparaciones):

- **Unarias:** $-$ y *abs*.
- **Binarias** $-$, $+$, $*$, *div* y *mod*.

Con lo que la gramática reducida usada en Evosuite, queda:

- **No Terminales:** IntegerExpression, IntegerUnaryExpression, IntegerBinaryExpression, Operator, Value
- **Terminales:** IntegerVariable, IntegerConstant, -, +, *, div, mod, abs
- **Símbolo distinguido:** IntegerExpression
- **Producciones:**

IntegerExpression → Value | integerUnaryExpression | IntegerBinaryExpression

integerUnaryExpression → Operator IntegerExpression

IntegerBinaryExpression → Operator IntegerExpression IntegerExpression

Value → IntegerVariable | IntegerConstant

Operator → - | * | + | div | % | abs

La Implementación

Gracias a tener una gramática definida podemos aplicar un patrón *Interpreter*[6] para modelar las expresiones simbólicas como se puede ver en la figura 4.13

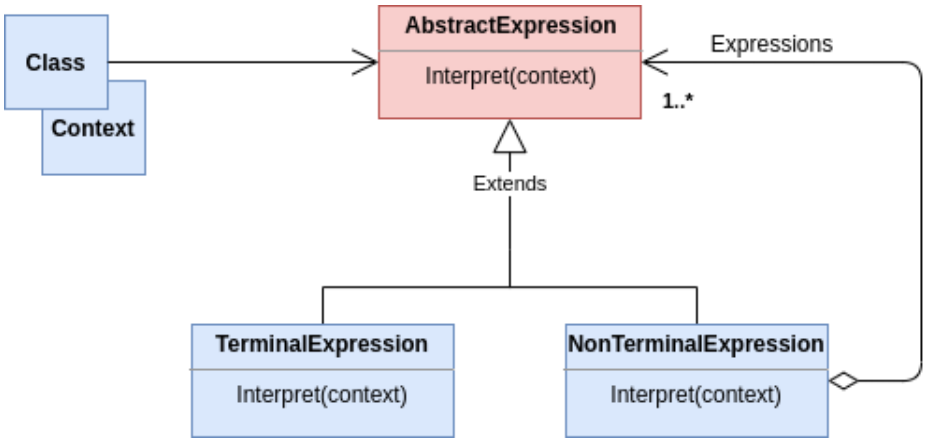

Fig. 4.13: Diagrama UML del patrón interpreter

Esto nos termina definiendo una composición de expresiones que iremos creando a lo largo de la ejecución y guardando como expresiones abstractas en σ, utilizando la gramática correspondiente a cada teoría, esto será muy útil para parsear las expresiones mas tarde. En particular, implementamos en las expresiones un patrón **Visitor**[6], de manera que podamos iterar el árbol de sintaxis subyacente mas fácil mas adelante y, además, guardamos el valor concreto de la ejecución en el objeto de la expresión para llevar un registro del valor concreto asociado. Podemos ver como la composición de expresiones forma el AST correspondiente en la figura 4.14

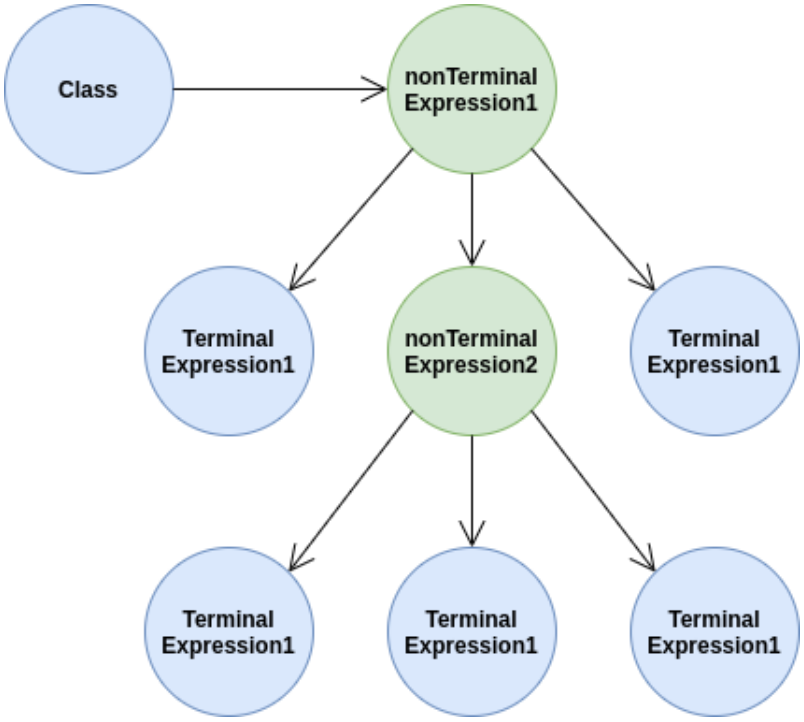

Fig. 4.14: AST asociado al patrón interpreter.

Finalmente, como las teorías suelen ser disjuntas, agregamos una interfaz para cada tipo de dato (Sort en SMT) para clasificar las expresiones que usamos.

En la actualidad se encuentran implementadas la teoría de **enteros**, **reales** y **strings**.

4.1.4. Creación y actualización del estado simbólico

Una distinción importante es el momento donde se crean las variables simbólicas dentro del entorno simbólico. Recapitulando de la sección 2.4, solo queremos predicar sobre los inputs, lo que significa que todo el resto de la ejecución debe generar literales. Para esto dividimos la ejecución concólica del test case en dos momentos: *antes* de la ejecución el SUT y *durante* la ejecución del SUT.

- **Antes:** Todas las variables que estemos construyendo en esta etapa serán las que se usen como inputs para el SUT por lo que deben representar variables en el estado simbólico al iniciar la ejecución del SUT. No entraremos en detalle sobre como realizar esta distinción pero diremos que esta parte pasa a través de una VM distinta a las que vimos antes, llamada *SymbolicObserver*. Esta interpreta las instrucciones de los inputs ejecutados y usa la API de *SymbolicEnvironment* para generar las variables correspondientes.
- **Durante:** Durante la ejecución del SUT, las VMs usan la API de *SymbolicEnvironment* solo para generar literales en el caso de que hubiera una variable nueva, de esta manera todas las expresiones quedan generadas según las variables de los inputs generadas previamente.

Ambos momentos se pueden ver en la figura 4.15.

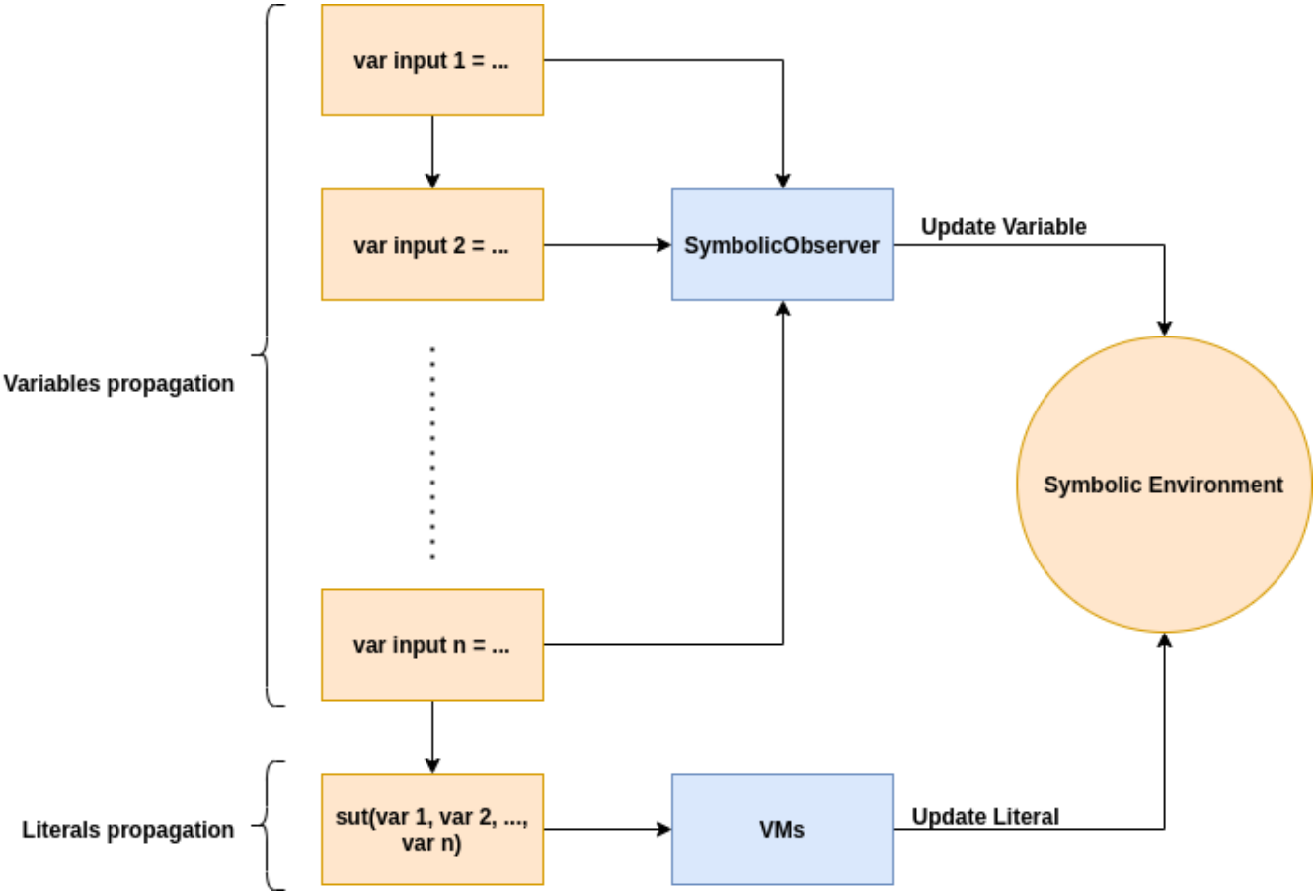

Fig. 4.15: Momentos de ejecución concólica en un test case.

4.2. Arreglos

Hasta el momento dimos la construcción del interprete simbólico y de sus expresiones, en esta sección veremos la extensión del interprete para soportar arreglos de manera simbólica. Primero veremos la teoría soportada por SMT-lib para arreglos, luego daremos su gramática y hablaremos de los detalles de su implementación dentro del heap. Finalmente, daremos una explicación general de su resolución del lado del solver y la actualización de los test cases.

No muchas herramientas soportan arreglos simbólicos en java, en particular DSC[39] y Symbolic PathFinder[47] utilizan la teoría de arreglos provista por SMT-lib. Nosotros haremos algo parecido y luego daremos una implementación alternativa perdiendo un poco de completitud.

4.2.1. Arreglos en Java

Los arreglos tienen un ciclo de vida similar a los objetos convencionales, cuando se quiere crear un nuevo arreglo, se crea un objeto asociado y se usa una referencia para manejarlo. Al ser objetos dinámicos los arreglos viven en el *heap*, sin embargo, a nivel byte-code su uso no es exactamente igual al de un objeto.

Creación

Para crear un nuevo arreglo se utilizan las instrucciones *NEWARRAY*, *ANEWARRAY* o *MULTINEWARRAY* en caso de que se cree un arreglo de tipos primitivos, referencias a objetos o multidimensional respectivamente. Nosotros haremos foco en el primero. En todos los casos, para su creación se necesita un tipo de dato (que corresponderá a su contenido) y la dimensión (o dimensiones para el ultimo caso), esto resultará en una referencia hacia el nuevo arreglo correspondiente.

Uso

Para su uso hay tres operaciones

1. Almacenar usando las instrucciones de la forma XASTORE
2. Seleccionar usando las instrucciones de la forma XALOAD
3. Consultar el largo usando la instrucción ARRAYLENGTH.

Las correspondencias para las instrucciones de almacenamiento y selección son: I: Integer, C: Character, S: Short, A: Reference, L: Long, B: Boolean, F: Float, D: Double. Nosotros haremos foco en todas excepto en AALOAD, ya que solo daremos soporte a los tipos primitivos.

Posibles errores

Las posibles errores de ejecución que se pueden generar son dos:

1. Un intento de acceso (tanto para almacenar como para seleccionar) a una posición fuera del rango del arreglo.
2. Un intento de almacenar un tipo invalido (e.g. IASTORE en un arreglo de floats).

4.2.2. Arreglos en el Heap Simbólico

Dadas las instrucciones que utilizan los arreglos, pasemos al diseño del heap. Como vimos en la sección 4.1.2, la **HeapVM** se encarga de procesar todas las instrucciones relacionadas con el heap, en particular las de los arrays como vimos antes, a esta altura actualizamos el valor del heap simbólico de acuerdo a instrucción que ejecutemos. En particular el heap tendrá una sección especialmente diseñada para los arreglos como se puede ver en la figura 4.16.

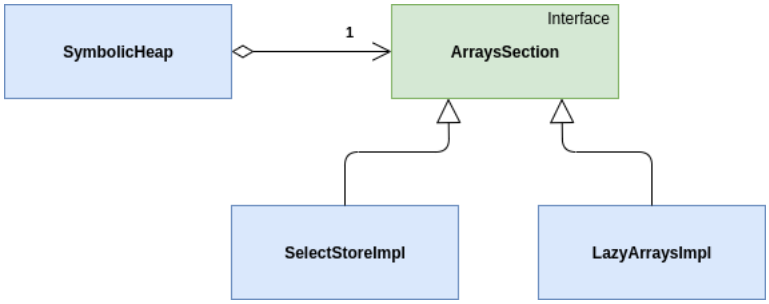

Fig. 4.16: Diagrama de clases para la sección de los arrays del heap

.

Todas las responsabilidades del heap simbólico con respecto a los arreglos (creación y actualización) quedará redirigida hacia *ArraySection* a modo de **API**, de esta manera podemos encapsular diferentes implementaciones de arreglos dentro del heap simbólico. A modo de trabajo futuro queda hacer un rework de la clase *SymbolicHeap* de manera que se asemeje a las VMs simbólicas y pueda separar su API en secciones disjuntas (entre ellas, la de los arreglos). La API conceptualmente queda definida como:

- **CreateVariable:** Crea internamente un nuevo arreglo simbólico representando una variable y devuelve su referencia simbólica.
- **CreateConstant:** Crea internamente un nuevo arreglo simbólico representando un literal y devuelve su referencia simbólica
- **ArrayLoad:** Efectúa una operación de selección sobre un arreglo y un índice dado y devuelve una expresión que lo represente.
- **ArrayStore:** Efectúa una operación de almacenamiento sobre un arreglo, un índice y un valor dados y altera el estado simbólico del arreglo

En particular los últimos dos estarán divididos por tipo de operandos (enteros, reales, strings y referencias). En esta implementación solo contemplamos enteros y reales.

4.2.3. Implementación 1: Teoría de Arreglos

Esta implementación esta situada en la clase *SelectStoreImpl* de la figura 4.16. Consiste en usar la teoría de arreglos ya definida en SMT-lib, extendiendo la gramática simbólica actual para soportarla en el interprete simbólico.

Teoría de Arreglos en SMT-lib

SMT-lib define la teoría de arreglos con un tipo *Array* con dos tipos X e Y, los cuales representan el Tipo del *indice* y del *contenido* a la par de dos axiomas:

1. **Seleccionar:** select (Array X Y) X Y. Esto es, dado un arreglo con tipo de índice X y tipo de contenidos Y, obtener el valor del índice X.
2. **Almacenar:** store (Array X Y) X Y (Array X Y). Esto es, dado un arreglo con tipo de índice X y tipo de contenidos Y, guardar en el índice X el valor Y.

Es importante remarcar que la teoría considera arreglos infinitos con lo que no predica sobre el largo del arreglo, esto tendremos que modelarlo por separado como veremos mas adelante.

Actualización de σ

Agregamos 4 posibles operaciones sobre σ, dos creaciones (literales y variables) y dos actualizaciones (select y store). La semántica de las creaciones simplemente agrega el literal o la variable correspondiente a la memoria, nos interesaremos en las actualizaciones. Para el estado simbólico usaremos la notación vista en la sección 2.4.

- **Select:** En caso de que se efectúe un select, no modificaremos la memoria pero si crearemos una nueva expresión que sera propagada. Esto es, sea arr la variable correspondiente al arreglo, i el valor simbólico correspondiente al índice del acceso, la nueva expresión e propagada será

$$e = select(\sigma[arr], i)$$

- **Store:** En este caso queremos actualizar el estado simbólico asignando un nuevo valor al arreglo. Sea arr la variable correspondiente al arreglo, i el valor simbólico correspondiente al índice del acceso, v el valor simbólico asociado que se quiere guardar en el arreglo y $arr_{concreto}$ el arreglo concreto resultante de la actualización, la nueva expresión e será

$$e = store(\sigma[arr], i, v)$$

  Actualizando σ

$$\sigma' = \sigma + [arr \rightarrow e]$$

  Además, la memoria concreta $\mathbf{M}$ se actualiza tal que

$$\mathbf{M}' = \mathbf{M} + [arr \rightarrow arr_{concreto}]$$

Largo del Arreglo

Dado que el largo del arreglo queda por fuera de la teoría de arreglos, necesitamos modelarlo como un elemento separado en la ejecución simbólica. Para esto necesitamos (1) generar una variable simbólica separada para representar el largo, (2) ligarla al arreglo de alguna manera, (3) encontrar los momentos durante la ejecución donde es relevante, (4) resolverla en el SMT-solver y (5) actualizar el largo en el test case. Veamos cada una por separado.

Para resolver (1) y (2), al momento de crear el arreglo simbólico dentro de *SymbolicObserver* creamos una variable simbólica entera con el nombre reservado

<nombre de la variable simbólica del arreglo asociado>_length_<numero de dimensión>

cuyo valor por defecto es 0. Por ejemplo, si el arreglo simbólico tiene nombre arr_0 y es unidimensional (Por el momento no se soportan arreglos multidimensionales), el nombre del largo simbólico será $arr_0\_length\_0$, donde la palabra reservada **length** define que predica sobre el largo de un arreglo. Esta variable es asociada como campo de instancia al arreglo dentro del heap y luego sera accesible cuando se utilice el arreglo (e.g. en IALOAD, ARRAYLENGTH, etc.).

Para resolver (3) miramos cuando se necesita usar el largo, en el caso de llamar a ARRAYLENGTH simplemente se levanta la expresión simbólica asociada del heap y se la propaga. El segundo punto es ver si se efectuó un acceso fuera de rango en el arreglo asociado al ejecutar un XALOAD o XASTORE, en este punto miramos si se produjo un *IndexOutOfBounds* comparando el valor concreto del índice contra el concreto del largo del arreglo. Que ocurra un acceso fuera de rango lo modelamos como un brancheo en la ejecución entre *error* y *no error*, en este punto extendemos la noción de condición de brancheo que veníamos usando hasta el momento.

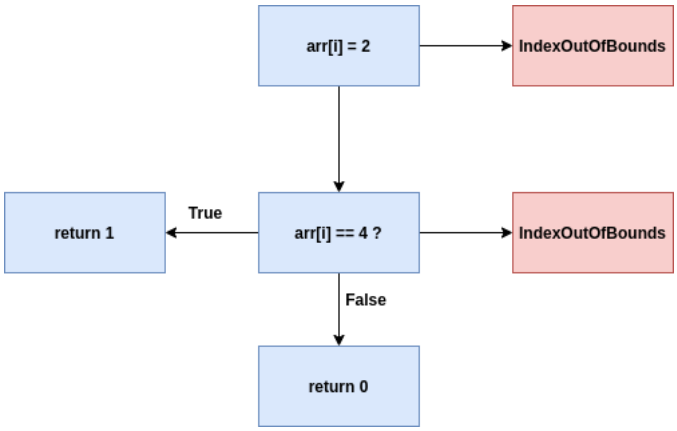

Fig. 4.17: Extensión en la noción de condición de brancheo

.

Como podemos ver en la figura 4.17, extendemos los lugares donde se recolectan constraints para la PC, a partir de ahora acumulamos también las constraints sobre los accesos al arreglo. Esto es útil también para encontrar este tipo de errores durante la exploración concólica. Veremos un ejemplo integrador completo al final de esta sección.

Para resolver (4), como la variable simbólica representa un entero, simplemente se resolverá como cualquier otro entero usando su teoría correspondiente en el SMT solver si aparece en la PC, con lo que no hay necesidad de implementar nada de ese lado.

Para resolver (5), al actualizar el test case con los valores de la solución encontrada, primero miramos si el nombre de la variable simbólica tiene una palabra reservada, en caso de ser el largo de un arreglo, se procesa el nombre simbólico del arreglo del nombre del largo, se levanta su statement asociado en el test case y se actualiza su largo al nuevo valor.

Por ejemplo, sea el algoritmo 8 y su CFG asociado en la figura 4.18.

Algorithm 8: arrayLengthExample

```
   Result: Integer
   Input: int[] x
1  if x[2] == 3 then
2  |  return 0;
3  end
4  return 1;
```

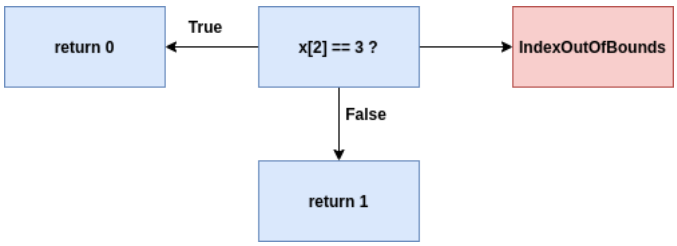

Fig. 4.18: CFG asociado al algoritmo 8

.

Comenzamos la ejecución con

$$\sigma = \{x = x_0\}, \text{ estado concreto: } \{\text{x = new int[0]}\} \text{ y } PC = \emptyset$$

Al ejecutar la linea **1**, nos encontramos con un *IndexOutOfBounds*, lo que se agrega a la PC y se termina la ejecución con el estado

$$\sigma = \{x = x_0\}, \text{ estado concreto: } \{\text{x = new int[0]}\} \text{ y } PC = x_0\_length\_0 \leq 2$$

Negando el único valor de la PC obtenemos

$$PC = x_0\_length\_0 > 2$$

Luego, suponiendo que el solver resuelva $x_0\_length\_0 = 3$, actualizando el test case comenzamos la próxima ejecución con

$$\sigma = \{x = x_0\}, \text{ estado concreto: } \{\text{x = new int[3]}\} \text{ y } PC = \emptyset$$

Lo que luego de ejecutar la linea **1** resultara en

$$\sigma = \{x = x_0\}, \text{ estado concreto: } \{\text{x = new int[3]}\} \text{ y } PC = x_0\_length\_0 > 2$$

A continuación agregaremos la gramática para poder recopilar el constraint correspondiente de x al haber recorrido el branch true en el caso anterior.

Gramática Simbólica

Como explicamos en la sección anterior, daremos la gramática simbólica correspondiente y no entraremos en detalle sobre su implementación. La idea general es separar los tipos de arreglos simbólicos de la misma manera que se separan los datos normales en cuatro tipos: *enteros*, *reales*, *strings* y *referencias*.

Como los arreglos representan de manera subyacente una referencia, el núcleo de la expresión representara una referencia a un arreglo, no daremos toda la gramática de las referencias y solo veremos la extensión propuesta:

- **No Terminales:** ReferenceExpression, IntegerArrayExpression, RealArrayExpression, StringArrayExpression, ReferenceArrayExpression, ...
- **Terminales:** ReferenceConstant, ReferenceVariable, IntegerArrayConstant, IntegerArrayVariable, RealArrayConstant, RealArrayVariable, StringArrayConstant, StringArrayVariable, ReferenceArrayConstant, ReferenceArrayVariable
- **Símbolo distinguido:** ReferenceExpression
- **Producciones:**

  ReferenceExpression → ... | ReferenceValue | IntegerArrayExpression | RealArrayExpression | StringArrayExpression | ReferenceArrayExpression

  IntegerArrayExpression → ...

  RealArrayExpression → ...

  StringArrayExpression → ...

  ReferenceArrayExpression → ...

  ReferenceValue → ReferenceConstant | ReferenceVariable

Haciendo foco en la construcción de los enteros (el resto es análogo)

- **No Terminales:** IntegerArrayExpression, IntegerArrayStore
- **Terminales:** IntegerArrayConstant, IntegerArrayVariable
- **Símbolo distinguido:** IntegerArrayExpression
- **Producciones:**

  IntegerArrayExpression → IntegerArrayValue | IntegerArrayStore

  IntegerArrayStore → IntegerArrayExpression IntegerExpression IntegerExpression

  IntegerArrayValue → IntegerArrayConstant | IntegerArrayVariable

Además, extendemos los valores de la gramática de enteros agregando la selección de un valor de un arreglo de enteros (ya que obtener un valor entero de un arreglo de enteros resulta en un valor entero). La gramática extendida de los enteros resulta en:

- **No Terminales:** IntegerExpression, IntegerUnaryExpression, IntegerBinaryExpression, IntegerArraySelect, Operator, Value

- **Terminales:** IntegerVariable, IntegerConstant, -, +, *, div, mod, abs
- **Símbolo distinguido:** IntegerExpression
- **Producciones:**

  IntegerExpression → Value | integerUnaryExpression | IntegerBinaryExpression | **IntegerArraySelect**

  integerUnaryExpression → Operator IntegerExpression

  IntegerBinaryExpression → Operator IntegerExpression IntegerExpression

  IntegerArraySelect → IntegerArrayExpression IntegerExpression

  Value → IntegerVariable | IntegerConstant

  Operator → - | * | + | div | % | abs

Resolución SMT

Para la resolución en SMT agregamos soporte para los tipos (sorts) de arreglos en el parser del módulo *Solver*, este hace uso del patrón *visitor* para iterar el AST correspondiente y traducir la expresión a la query SMT. A la hora de traducirla se agregaron los axiomas *select* y *store* también vistos previamente. Para parsear la respuesta se extendió el parser asociado para reconstruir un arreglo basándose en la respuesta del solver.

Actualización del test case

Por un lado, el largo se actualiza de la misma manera que vimos antes, para actualizar el arreglo, se iteran los nuevos elementos generados por el solver SMT y se miran las diferencias contra el arreglo anterior (el valor actual del statement asociado), en caso de diferir se actualiza el statement de asignación correspondiente (o crea en caso de no haberse modificado previamente). En particular, la estrategia de construcción del arreglo es en dos partes: (1) un statement de creación y (2) una serie de statements de asignación. En (1) se liga la variable simbólica del largo y en (2) se ligan cada una de las posiciones que se hayan modificado a lo largo de las iteraciones. Ambas partes usan como base común el nombre del arreglo.

Un Ejemplo

Veamos un ejemplo de la ejecución tomando el algoritmo 9 y su CFG asociado 4.19.

Algorithm 9: arrayLengthExample

```
   Result: Integer
   Input: int[] x, int a
1  if x[2] == 3 then
2  |  x[1] = a;
3  |  if x[1] == 4 then
4  |  |  return 0;
5  |  else
6  |  |  return 2;
7  |  end
8  end
9  return 1;
```

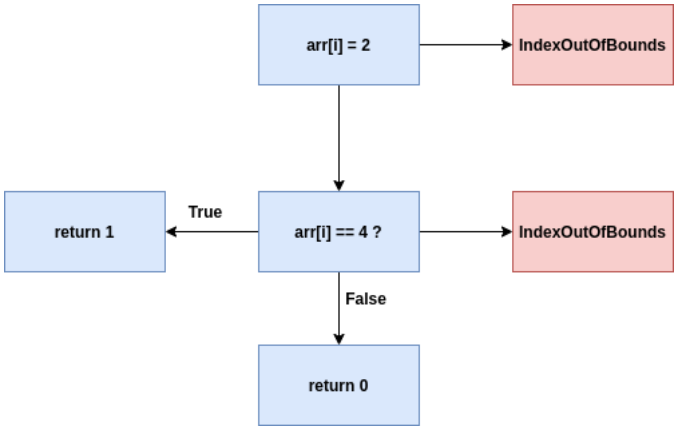

Fig. 4.19: CFG asociado al algoritmo 9

.

Iteración 1: Comenzamos la ejecución con el estado:

$$\sigma = \{x = x_0, a = a_0\}, \text{ estado concreto: } \{\text{x = new int[0], a = 0}\} \text{ y } PC = \emptyset$$

Luego de ejecutar la linea **1**, da *IndexOutOfBounds* y generamos la PC:

$$PC = x_0\_length\_0 \leq 2$$

Iteración 2: Siguiendo los mismos pasos que en el ejemplo del largo pasamos a una ejecución inicial con el siguiente estado

$$\sigma = \{x = x_0, a = a_0\}, \text{ estado concreto: } \{\text{x = new int[3], a = 0}\} \text{ y } PC = \emptyset$$

Luego de ejecutar la linea **1** generamos una selección sobre el arreglo con una comparación y un acceso, generando una constraint como vimos en la sección del store simbólico σ usando *select*:

$$\sigma = \{x = x_0, a = a_0\}, \text{ estado concreto: } \{\text{x = [0, 0, 0], a = 0}\} \text{ y}$$
$$PC = x_0\_length\_0 > 2 \wedge (select\ x_0\ 2) \neq 3$$

Dado que los arreglos concretos contienen valores por defecto según su tipo (e.g. para enteros todos 0s), el arreglo X tendrá 0 en todas las posiciones, llevando a la ejecución concreta por la rama false del branch en la linea **1**. Lo que finalizará la ejecución devolviendo el valor *1*. Luego, como esta PC fue generada a partir de otro de largo 1, solo negamos la ultima constraint lo que nos da la PC (Por bound en algoritmo 4):

$$PC = x_0\_length\_0 > 2 \wedge (select\ x_0\ 2) = 3$$

Suponiendo que el solver nos devuelve los valores:

$$x_0 = (store\ (Array\ Int\ Int\ 0)\ 2\ 3), x_0\_length\_0 = 3$$

Con esto podemos actualizar los valores del arreglo x en la respuesta y agregar un statemnt para agregar un elemento, el setup del test case queda:

Algorithm 10: test case actualizado

```
1 // Array initialization;
2 int[] x = new int[3];
3 x[2] = 3;
4 // integer initialization;
5 int a = 0;
6 // SUT call;
7 sut(x, a);
```

Iteración 3: Empezamos la ejecución del SUT con el estado:

$$\sigma = \{x = x_0, a = a_0\}, \text{estado concreto: } \{\text{x} = [0, 0, 3], \text{a} = 0\} \text{ y } PC = \emptyset$$

Adelantando la ejecución hasta el mismo punto que hasta el fin de ejecutar la linea **1**, el estado nos queda

$$\sigma = \{x = x_0, a = a_0\}, \text{estado concreto: } \{\text{x} = [0, 0, 3], \text{a} = 0\} \text{ y}$$
$$PC = x_0\_length\_0 > 2 \wedge (select\ x_0\ 2) = 3$$

Con lo que tomamos el branch true en la linea **1** y pasamos a ejecutar la linea **2**, lo que resulta en una operación *store* dentro del arreglo, la memoria concreta queda igual solo porque el valor de a era 0, pero en la memoria simbólica actualizamos el valor de x:

$$\sigma = \{x = (store\ x_0\ 1\ a_0), a = a_0\}, \text{estado concreto: } \{\text{x} = [0, 0, 3], \text{a} = 0\} \text{ y}$$
$$PC = x_0\_length\_0 > 2 \wedge (select\ x_0\ 2) = 3$$

Luego pasamos a ejecutar la linea **3** donde hacemos un *select* y creamos una constraint para agregar a la PC, el estado queda:

$$\sigma = \{x = (store\ x_0\ 1\ a_0), a = a_0\}, \text{estado concreto: } \{\text{x} = [0, 0, 3], \text{a} = 0\} \text{ y}$$
$$PC = x_0\_length\_0 > 2 \wedge (select\ x_0\ 2) = 3 \wedge (select\ (store\ x_0\ 1\ a_0)\ 1) \neq 4$$

Esto es, agregamos la condición $(select\ (store\ x_0\ 1\ a_0)\ 1) \neq 4$ a la PC, quiere decir que la selección sobre el valor simbólico actual de x $(store\ x_0\ 1\ a_0)$ en la posición 1 no es igual a 4, en particular el valor actual dependerá del valor simbólico de a. Luego retornamos **2** por haber recorrido la rama false del branch en la linea **3**. Negando el ultimo elemento de la PC obtenemos

$$PC = x_0\_length\_0 > 2 \wedge (select\ x_0\ 2) = 3 \wedge (select\ (store\ x_0\ 1\ a_0)\ 1) = 4$$

Como el valor depende de a, el solver resolverá algo del estilo:

$$x_0 = (store\ (Array\ Int\ Int\ 0)\ 2\ 3), x_0\_length\_0 = 3, a_0 = 4$$

Con lo que luego de actualizar el test case como vimos antes, pasamos a la ultima iteración.

Iteración 4: Finalmente, el estado inicial de la ejecución nos queda

$$\sigma = \{x = x_0, a = a_0\}, \text{estado concreto: } \{\text{x} = [0, 0, 3], \text{a} = 4\} \text{ y } PC = \emptyset$$

Repitiendo lo mismo que en la iteración anterior logramos recorrer la rama true de la linea **3**, devolviendo **0** y finalizando la ejecución.

4.2.4. Implementación 2: Arreglos Vagos

Esta idea transforma al arreglo de una variable simbólica representando el arreglo entero a un conjunto de variables independientes que serán propagadas a lo largo de la ejecución y luego unidas al final.

Idea Principal

Como vimos en la implementación 1, las nociones del uso de arreglos son dos: almacenar, seleccionar. El largo ya lo modelamos de manera separada con lo que reusamos esa implementación (queda implementada de manera ortogonal). En particular las expresiones generadas por la teoría de arreglos van generando expresiones anidadas donde un *store* previo puede haber "pisado" una dirección de memoria, en ese caso un *select* posterior estaría predicando sobre ese valor en particular. En caso de no haber aplicado nunca un *store* a una posición del arreglo en particular, esa posición aun contendrá el valor original al momento de haber ingresado al SUT, con lo que de aplicar un *select* sobre ella en este momento, estaríamos predicando sobre la variable original que es la que queremos averiguar. Definimos:

Definición 4.2.1. Posición Sucia Decimos que una posición de un arreglo es *sucia* si se ha escrito en ella.

En el contexto de la ejecución concólica diremos que una posición del arreglo pasa a ser *sucia* cuando se la escribe dentro del *SUT*. Esta idea propone, en lugar de representar un arreglo como una sola entidad en memoria simbólica, usarlo como locación de memoria para alojar n variables simbólicas independientes.

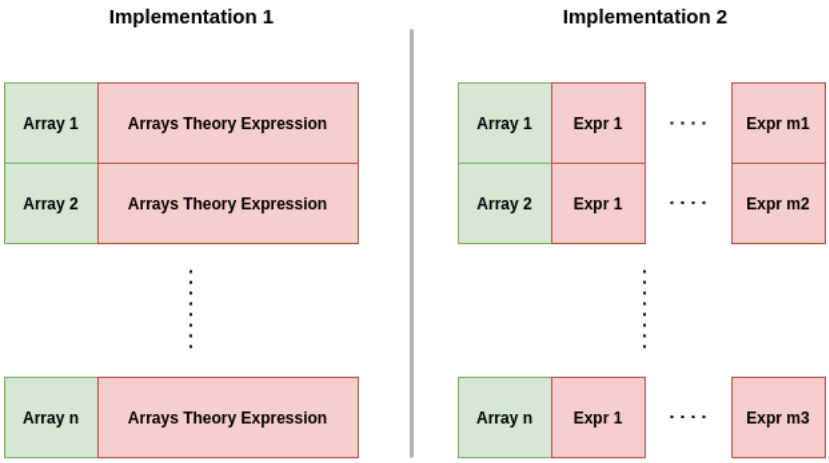

Fig. 4.20: Comparaciones entre modelos de memoria simbólica de arreglos para la implementación 1 vs implementación 2.

.

En la figura 4.20 podemos ver, a la izquierda, el modelo de memoria de la implementación 1 donde para el arreglo i-ésimo guarda una sola expresión simbólica representada con la gramática derivada de la teoría de arreglos. A la derecha, proponemos que el arreglo guarde n variables, cada una representada por la teoría subyacente del tipo del arreglo durante la ejecución concólica. De este modo, si un *store* es aplicado al arreglo, simplemente se reemplazará la expresión en esa posición por la nueva, mientras que si se aplica un *select* hay dos casos: si la posición está sucia o no. En caso de estar sucia, un *store* previo la pisó y devolvemos la expresión almacenada. En caso de no ser sucia, quiere decir que esa posición predica sobre el valor que el arreglo tenía en esa posición al ingresar al SUT (el input que nos interesa encontrar), en ese caso generamos dinámicamente la variable simbólica que represente ese elemento y la devolvemos.

Implementación

Recordemos que el modelo de memoria de los arreglos lo armamos a modo de *API* en *ArraysSection* de modo que al resto del motor le es transparente como manejemos internamente los arreglos siempre y cuando las expresiones que devolvamos sean de tipo válido. Para esto mantenemos como arreglo expresiones de tipo **ReferenceExpression** como ya veníamos usando y recordemos que por como modelamos la gramática en la implementación anterior, las operaciones de *select* tipan con respecto a la teoría del tipo del componente del arreglo, con lo que devolver una expresión de ese tipo basta.

Con esto modelamos esta implementación en la clase **LazyArraysImpl**, esta mantiene el estado interno de la memoria como vimos en la figura 4.20. Mantenemos la distinción entre arrays literales y variables de manera que los literales no puedan generar variables frescas mientras que los variables si. En particular, para ligar las variables simbólicas frescas generadas usaremos el nombre reservado

<nombre de la variable simbólica del arreglo asociado>_content_<numero de índice>

Con esto ligamos la variable a su arreglo asociado y a su posición. Por ultimo, una vez generada una PC, los valores se resolverán en sus teorías respectivas en el SMT solver, con

lo que no necesitamos implementar nada de ese lado. A la hora de actualizar el test case, primero vemos si la variable tiene alguna palabra reservada, en caso de ser el contenido de un arreglo, obtenemos el nombre simbólico del arreglo de su nombre, buscamos su statement de asignación correspondiente y lo actualizamos (o creamos en caso de no haber sido necesario usar esa posición previamente).

Perdida de completitud

Una contra de la técnica es que, al momento de generar la variable fresca, concretizamos el índice al que se accede, en el caso de que se quisiera usar una variable i para acceder a un índice del arreglo y luego se la use para generar una constraint, la resolución del mismo no se vería reflejado en la posición accedida en nuestro arreglo, pudiendo generar una divergencia. Queda como trabajo futuro hacer algún análisis empírico para entender que tan común es este caso y cuanto impacta en la completitud de la técnica.

5. EXPERIMENTACIÓN PRELIMINAR

Recapitulando del capítulo 3, introdujimos un nuevo módulo de DSE así como su algoritmo de exploración. En este capítulo intentaremos describir un estudio empírico realizado para comparar este nuevo algoritmo con el que estaba implementado previamente en Evosuite (Por simplicidad lo llamaremos *algoritmo legacy*). En particular, no entraremos en detalles sobre el funcionamiento del algoritmo legacy y lo pensaremos a modo de caja negra. Muchos elementos de este capítulo están basados en Arcuschin [48]. Por restricciones de tiempos, los artefactos utilizados no fueron seleccionados para probar esta técnica en particular, con lo que los resultados de esta experimentación son a modo preliminar.

5.1. Objetivos

Dado que la meta de este capítulo es comparar los algoritmos, podemos tomar varios acercamientos para investigarlos pero siempre teniendo en cuenta dos elementos importantes: (1) La meta de la generación del test suite es maximizar cobertura y (2) ambos algoritmos cumplen con el esquema de un algoritmo de exploración tradicional en DSE como discutimos en el capítulo 3, esto es, ambos deben generar un input inicial, obtener una PC y luego generar alguna suerte de input sucesor, resolviendo constraints a través de un SMT solver, para continuar explorando. Proponemos orientar la experimentación haciendo foco en estos dos elementos guiándonos por las siguientes preguntas generales:

- **Global:** ¿Cómo se comparan los algoritmos en términos de cobertura generada dada una cota sobre el tiempo de exploración permitido?
- **Desglose:** ¿Cómo se diferencian las ejecuciones realizadas entre estos algoritmos en relación con los tiempos de uso del solver respectivamente contra la ejecución concólica, uso de la cache, y tamaño de las PC?

Como variantes generales utilizamos dos configuraciones sobre la ejecución, dejando todo el resto por defecto en la herramienta:

- **Tiempo de Exploración:** 180s.
- **Algoritmo de Exploración:** Búsqueda generacional, legacy.
- **Implementación de Arreglos simbólicos:** Teoría de arreglos.

5.1.1. Pregunta global

Dado que solo nos interesa ver la cobertura en relación con el tiempo, tomamos como única métrica la cobertura por ramas generada por la ejecución.

5.1.2. Pregunta desglosada

Dado que nos interesa hilar fino en los tiempos de ejecución de los algoritmos y en algunas de las maneras en las que funcionan, agregamos varias métricas:

- **Cache de contra-ejemplos:** Cantidad de llamados y hit rate,
- **Tiempos de uso:** Ejecución de test cases de manera concólica, ejecución del demostrador de teoremas.
- **Path conditions:** Tamaño mínimo, tamaño máximo, tamaño promedio, cantidad de caminos explorados.

Por un lado, nos interesa analizar que tanto uso hace el algoritmo de la cache de contraejemplos además de los tamaños que tendrán las PC resultantes de las ejecuciones y por otro, como esto impacte en los tiempos de ejecución de la técnica en diferentes momentos.

5.2. Sujeto

Para la experimentación se selecciono el conjunto de 38 clases usado en Galeotti et al. [3]. De dichos sujetos solo se utilizaron 16 debido a diversos factores externos, siendo el mas común la desaparición del proyecto de la web o que la clase ya no se encontraba en ese proyecto. El detalle de los sujetos seleccionados se puede ver en la Tabla 5.1. Todos los sujetos fueron utilizados en la experimentación.

Project	Class	LOC	Branches
Chemeval	org.openscience.cdk.index.CASNumber	23	15
Conzilla	se.kth.cid.identity.ResourceURL	22	12
Conzilla	se.kth.cid.identity.URI	66	51
Conzilla	se.kth.cid.identity.URIClassifier	48	30
Conzilla	se.kth.cid.identity.PathURN	17	10
Conzilla	se.kth.cid.identity.URN	17	12
Conzilla	se.kth.cid.identity.MIMEType	33	12
GSV05	stempeluhr.validation.TimeChecker	29	11
WIFE	com.prowidesoftware.swift.model.IBAN	103	46
WIFE	com.prowidesoftware.swift.model.BIC	149	103
Commons CLI	org.apache.commons.cli.CommandLine	90	53
JDom	org.jdom.Attribute	135	65
Commons Codec	org.apache.commons.codec.language.DoubleMetaphone	546	498
JodaTime	org.joda.time.DateTime	336	168
JodaTime	org.joda.time.DateTimeFormat	369	459
JGraphT	org.jgrapht.alg.BellmanFordIterator	103	45
Commons Math	org.apache.commons.math3.transform.FastFourierTransformer	256	101
NanoXML	net.n3.nanoxml.XMLElement	317	166

Tab. 5.1: Benchmark de clases utilizado. LOC(lines of uncomented code) fueron calculadas usando JavaNCSS[49]. Los branches fueron calculados usando Evosuite y son calculados a nivel bytecode.

Los artefactos (y sus decompilados), ejecutable, y scripts usados para correr los experimentos se pueden encontrar en el repositorio de github sobre esta experimentación [50] bajo el directorio *Experimentacion/Benchamarks/clases_del_paper/repos/jars*

5.3. Procedimiento

Dada la naturaleza aleatoria de los algoritmos y dado que pueden ocurrir errores/timeouts del lado del solver de forma esporádica se decidió correr cada combinación *variante-sujeto* 10 veces. El tiempo total de ejecución teórica fue de 18 *clases* x 2 *variantes* x 10 *repeticiones* x 180 *segundos* c/u = 64800 *segundos* = 18*hs* de cómputo estimado. Es decir, la experimentación total estimada fue aproximadamente en total de 0.75 días de ejecución.

Para ejecutar la técnica se utilizó una computadora con un procesador Intel Core i7-6600U (de 4 cores) y 8GB de RAM, corriendo Ubuntu 18.04.

Para cada par de variables A, B y métrica objetivo, utilizamos el test no paramétrico *U de Mann-Whitney* (En R, dados los datasets resultantes X e Y, se los puede comparar usando *wilcox.test(X, Y)*) para analizar la hipótesis nula: dada una ejecución de cada variante, es igual de probable que A sea mejor que B para el objetivo seleccionado que lo contrario. Esto es, la hipótesis nula dice que no hay diferencia estadística entre la efectividad de las dos variantes con respecto a la métrica objetivo elegida.

Para las variantes que mostraron una diferencia estadística, utilizamos la medida no paramétrica A_{12} de *Vargha y Delaney* para determinar si hay alguna magnitud significativa. A modo de ejemplo para su interpretación, dadas dos variantes A y B, un valor $A_{12} = 0{,}7$ significa que uno obtendrá mejores resultados 70 % del tiempo utilizando A (En R, dados los datasets resultantes X e Y, se los puede comparar usando *VD.A(X, Y)*. Las diferencias entre variantes son caracterizadas como pequeñas, medianas y largas cuando A_{12} supera 0.56, 0.64 y 0.71 respectivamente.

La formula para calcular la medida es:

$$A_{12} = (R_1/m - (m+1)/2)/n$$

Donde m es la cantidad de observaciones para la primera muestra de datos, y n la cantidad para la segunda. Para nuestro experimento, $m = n$, dado que corrimos la misma cantidad de veces todas las variantes. R_1 es la suma del ranking del primer grupo de datos bajo comparación. Por ejemplo, tomemos los datos $X = 42, 11, 7$ e $Y = 1, 20, 5$. Entonces, X tendría rankings $6, 4, 3$, cuya suma es 13, mientras que Y tendría rankings $1, 5, 2$. La suma de rankings es un componente básico del test *U de Mann-Whitney*, por lo que la mayoría de las herramientas lo proveen. En nuestro caso, los resultados estadísticos se calcularon utilizando el lenguaje de programación *R*.

5.4. Resultados

En la práctica, la mayoría de las clases resultaron no contener métodos estáticos con inputs de tipo primitivos o arreglos, con lo que la ejecución real resultó en un total de aproximadamente 3*hs* de ejecución, donde solo 3 clases generaron cobertura para el algoritmo *legacy* y 5 clases para la *búsqueda generacional*.

Para obtener una idea general de los resultados, se pueden ver las clases relevantes con sus promedios de cobertura en las tablas 5.2 y 5.3 para la búsqueda generacional y el algoritmo legacy respectivamente.

Clase	Cobertura Promedio
com.prowidesoftware.swift.model.IBAN	0,06521739130434782
org.apache.commons.math3.transform.FastFourierTransformer	0,039603960396039604
org.joda.time.format.DateTimeFormat	0,461220043572985
org.openscience.cdk.index.CASNumber	0,3333333333333333
se.kth.cid.identity.URIClassifier	0,646666666666667

Tab. 5.2: Clase y cobertura promedio de la ejecución de búsqueda generacional sobre el benchmark dado.

Clase	Cobertura Promedio
com.prowidesoftware.swift.model.IBAN	0.06521739130434782
org.joda.time.format.DateTimeFormat	0,618494311304769
se.kth.cid.identity.URIClassifier	0,536666666666667

Tab. 5.3: Clase y cobertura promedio de la ejecución del algoritmo legacy sobre el benchmark dado.

En particular la clase *IBAN* genero constantemente la misma cobertura para ambos algoritmos. Dado que solo las clases **DateTimeFormat** y **URIClassifier** dieron resultados distintos (el resto resulto en cobertura 0), haremos foco en ellas.

5.4.1. Pregunta general

En la tabla 5.4 se pueden ver los *p-valores* obtenidos del test *U de Mann-Whitney* junto con su medida A_{12} de *Vargha y Delaney*. Para determinar si se puede tomar o no la hipótesis nula vamos a tomar nivel de significancia $\alpha = 0{,}05$.

Clase	Mann-Whitney	Vargha y Delaney
org.joda.time.format.DateTimeFormat	0,0004371	0
se.kth.cid.identity.URIClassifier	0,0001796	0,95

Tab. 5.4: Clase, p-valor y A_{12} comparando algoritmo legacy vs busqueda generacional.

Podemos observar que ambos p-valores apoyan la hipótesis alternativa de que los algoritmos difieren, además la medida de efecto fue significativamente diferente, para *DateTimeFormat*, búsqueda generacional resulto 0 % de veces mejor que al algoritmo legacy como podemos ver en la figura 5.1

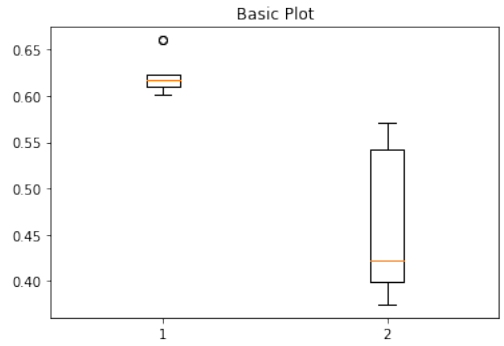

Fig. 5.1: Comparación de coberturas para la clase DateTimeFormat. En la figura, los valores de la izquierda representan las coberturas generadas para el algoritmo legacy, a la derecha para la búsqueda generacional.

.

Por otro lado, búsqueda generacional fue mejor en *URIClassifier* un 95 % de los casos como se puede ver en la figura 5.1.

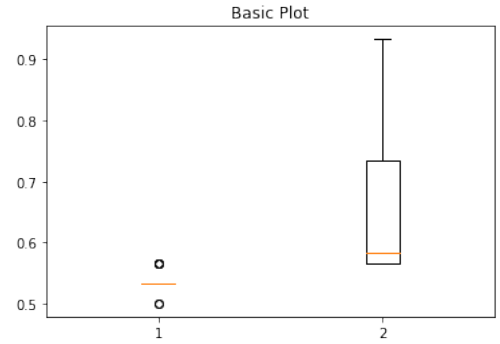

Fig. 5.2: Comparación de coberturas para la clase UriClassifier. En la figura, los valores de la izquierda representan las coberturas generadas para el algoritmo legacy, a la derecha para la búsqueda generacional.

.

A simple vista podemos ver que la varianza de la búsqueda generacional es considerablemente mas alta que la de la búsqueda legacy. Suponemos que se debe a la búsqueda generacional utiliza *cobertura por ramas incremental* como función de puntaje, la cual puede varias considerablemente el orden de las PC a navegar dependiendo de como se vayan resolviendo las ejecuciones concólicas, mientras que el algoritmo legacy realiza una búsqueda mas guiada (algo similar a un DFS), con lo que búsqueda resulta un poco mas determinista a lo largo de las sucesivas ejecuciones. A continuación haremos un desglose para entender un poco mejor porque se dan estos resultados.

5.4.2. Pregunta desglosada

En la pregunta anterior vimos que en un caso búsqueda generacional fue mejor y en otro no. A continuación veremos el mismo análisis en cada clase por separado. Veremos 4 puntos: (1) tamaño de las PC y cantidad de caminos explorados, (2) Uso de la cache de

contra ejemplos, (3) tiempos de ejecución y, por ultimo, (4) los resultados de las consultas hechas al SMT solver.

UriClassifier

Path Conditions: En la figura 5.3 se puede ver la cantidad de PC recorridas en cada clase. Se puede ver que búsqueda generacional recorre, en promedio, el doble que el algoritmo legacy.

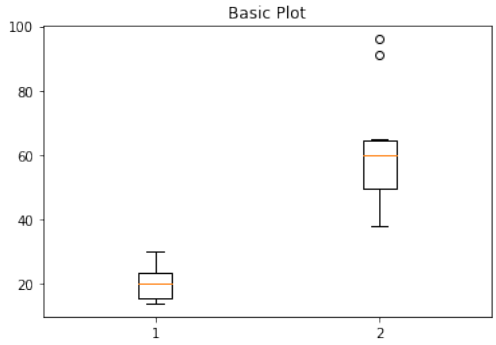

Fig. 5.3: Comparación de coberturas para la clase UriClassifier. En la figura, los valores de la izquierda representan las coberturas generadas para el algoritmo legacy, a la derecha para la búsqueda generacional.

.

Además, el tamaño de las PC en términos de su cantidad de constraints no difiere en gran escala, la cantidad mínima para ambos en 1, mientras que el promedio se mantiene entre 6 y 8 como se puede ver en la figura 5.4, además la máxima cantidad de constraints se comporta de manera parecida, siendo un poco mas alta para búsqueda generacional como se ve en la figura 5.5.

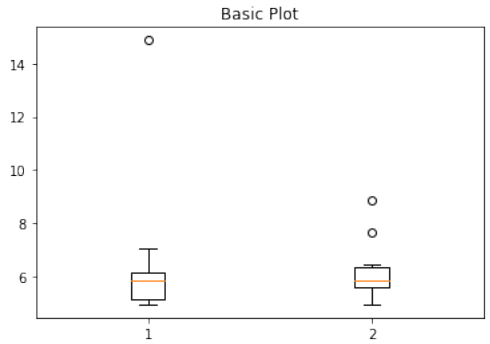

Fig. 5.4: Comparación del largo promedio de las PC para la clase UriClassifier.

.

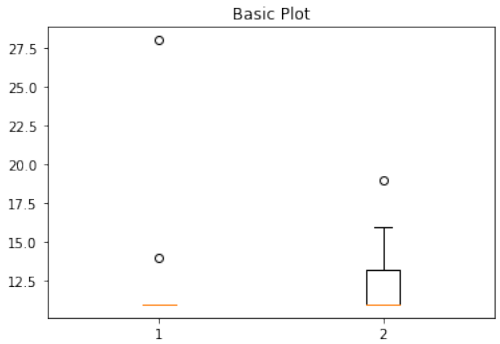

Fig. 5.5: Comparación del largo máximo de las PC para la clase UriClassifier.

.

Cache: Viendo el uso de la cache podemos ver como búsqueda generacional es significativamente menos eficiente en su uso donde el algoritmo legacy tiene, en promedio, un hit rate de alrededor del 53 % vs un promedio de alrededor del 25 % para búsqueda generacional en la figura 5.16.

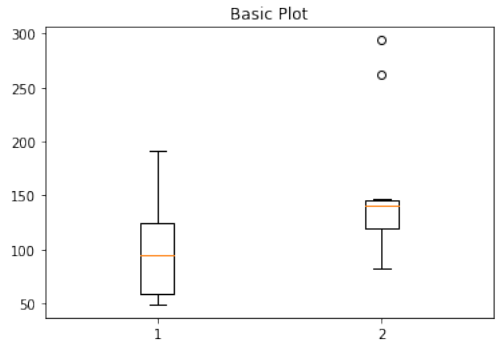

Fig. 5.6: Comparación de cantidad de llamadas a la cache para la clase UriClassifier.

.

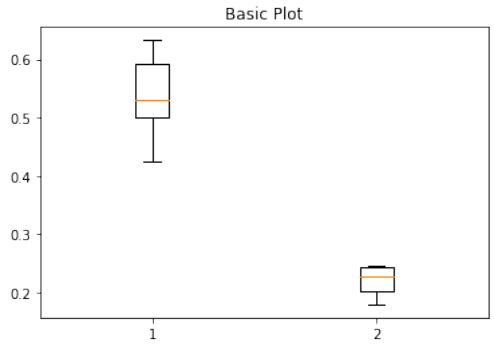

Fig. 5.7: Comparación del hit rate de la cache para la clase UriClassifier.

.

Tiempos de ejecución: Podemos ver como los tiempos de ejecución de la búsqueda generacional son mas altos en general tanto para la ejecución concólica como para la resolución de constraints.

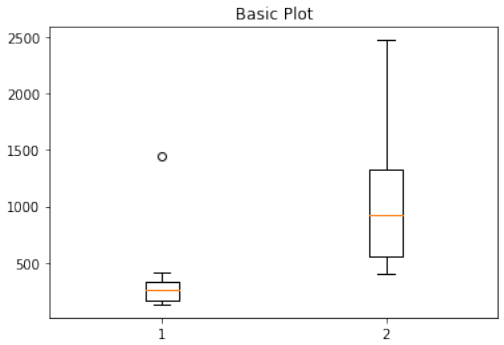

Fig. 5.8: Comparación sobre la cantidad de tiempo ejecutando el SMT solver para la clase UriClassifier.

.

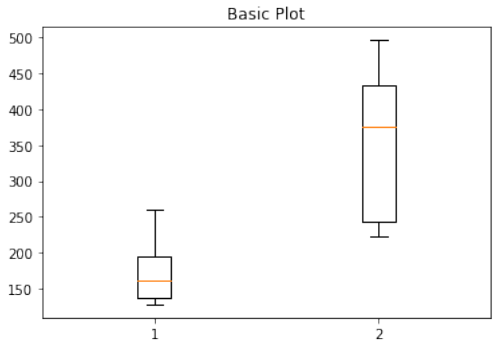

Fig. 5.9: Comparación sobre la cantidad de tiempo ejecutando concólicamente test cases para la clase UriClassifier.

.

Respuestas del SMT solver: En las figuras 5.10 y 5.11 podemos ver como la cantidad de consultas crece de manera proporcional tanto en las satisfacibles como en las no satisfacibles. Ninguno de los dos algoritmos reportó timeouts o errores como respuesta a las consultas SMT.

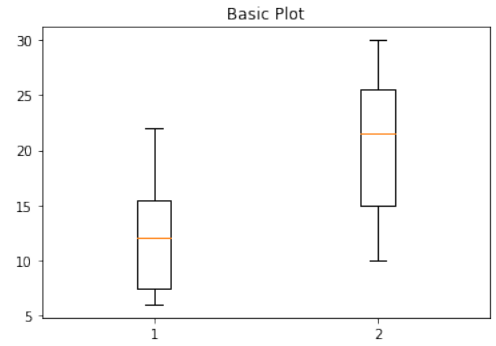

Fig. 5.10: Comparación sobre la cantidad de consultas que resultaron satisfacibles por el SMT solver para la clase UriClassifier.

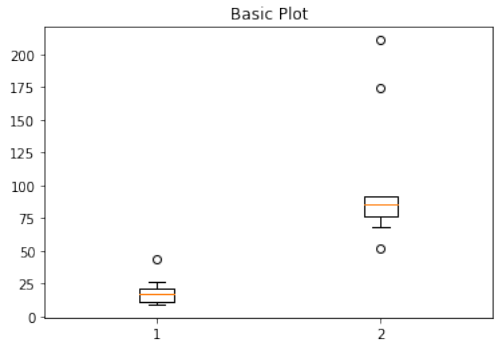

Fig. 5.11: Comparación sobre la cantidad de consultas que resultaron no satisfacibles por el SMT solver para la clase UriClassifier.

Dada la poca complejidad de la clase en términos de branches (Tiene tan solo 30 como se puede ver en la tabla 5.1) y los resultados vistos se podría sugerir que una leve ventaja por haber cubierto algunos branches mas le da la ventaja a la búsqueda generacional en este caso.

DateTimeFormat

A diferencia de la clase anterior, *DateTimeFormat* posee 450 branches, con lo que a simple vista su complejidad pareciera mucho mayor. A continuación repetiremos los pasos del análisis anterior.

Path Conditions: En la figura 5.12 se puede ver como búsqueda generacional explora una cantidad significativamente menor (casi 3 veces menos) de PC en comparación con el algoritmo legacy.

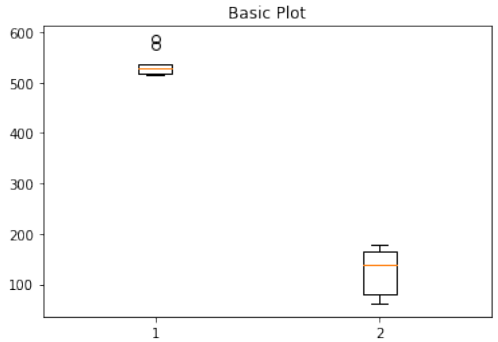

Fig. 5.12: Comparación de coberturas para la clase DateTimeFormat. En la figura, los valores de la izquierda representan las coberturas generadas para el algoritmo legacy, a la derecha para la búsqueda generacional.

Además, el tamaño de las PC en términos de su cantidad de constraints difiere (la cantidad mínima para ambos fue 1), mientras que el promedio tiene una varianza relati-

vamente alta para búsqueda generacional, esto no se ve en el algoritmo legacy, atribuimos esto a la diversidad de la búsqueda por la heurística de maximizar cobertura incremental por ramas de la técnica. Esto le daría prioridad a PC de mayor largo en la ejecución.

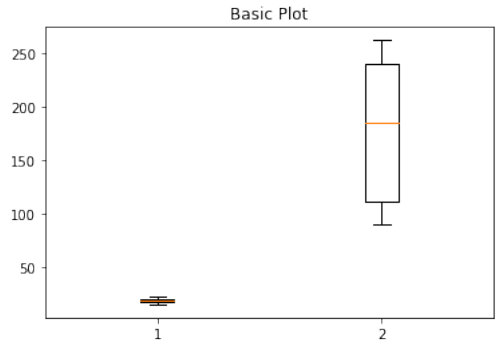

Fig. 5.13: Comparación del largo promedio de las PC para la clase DateTimeFormat.

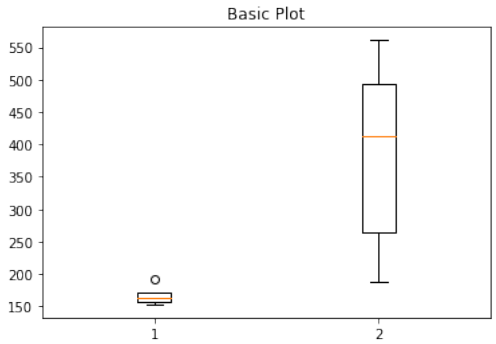

Fig. 5.14: Comparación del largo máximo de las PC para la clase DateTimeFormat.

Cache: Viendo el uso de la cache podemos ver como búsqueda generacional es aún menos eficiente que en *UriClassifieren* donde su hit rate fue 0 % vs 20 % apróximadamente del algoritmo legacy. Los resultados sugieren que, como el espacio de búsqueda de la clase es considerablemente mayor al de la clase anterior, las PC exploradas difieren bastante (por elegir siempre la que provee mayor coverage incremental) y esto lleva a no encontrar soluciones previas en la cache.

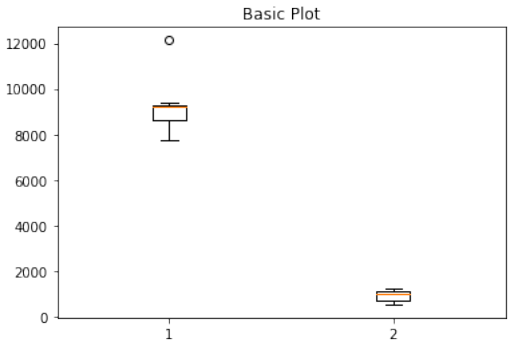

Fig. 5.15: Comparación de cantidad de llamadas a la cache para la clase DateTimeFormat.

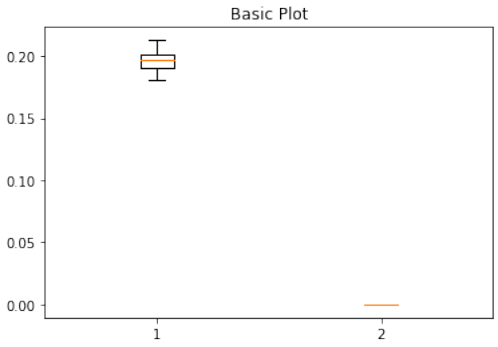

Fig. 5.16: Comparación del hit rate de la cache para la clase DateTimeFormat.

Tiempos de ejecución: En las figuras 5.17 y 5.18 podemos ver como los tiempos de ejecución concólica se reducen drásticamente en búsqueda generacional para dar lugar a una cantidad significativa de tiempo invertido en resolver constraints en el SMT solver.

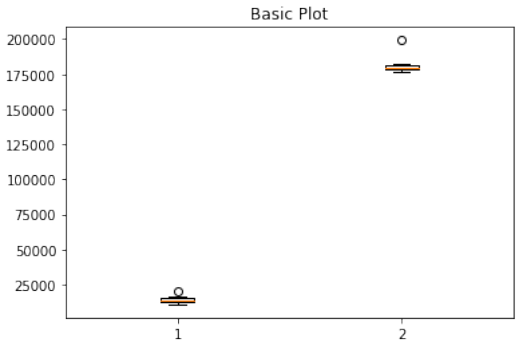

Fig. 5.17: Comparación sobre la cantidad de tiempo ejecutando el SMT solver para la clase DateTimeFormat.

.

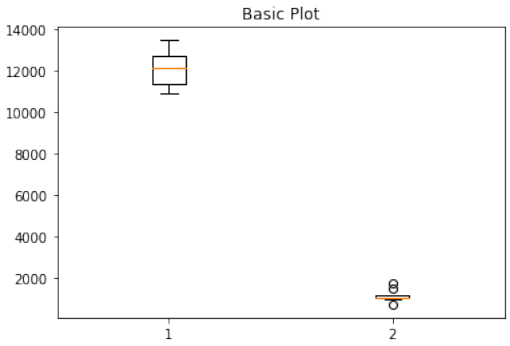

Fig. 5.18: Comparación sobre la cantidad de tiempo ejecutando concólicamente test cases para la clase DateTimeFormat.

.

Respuestas del SMT solver: Siguiendo el hilo de los puntos anteriores, podemos ver en las figuras 5.10 y 5.11 como la cantidad de consultas tanto satisfacibles como no satisfacibles son mucho menores en búsqueda generacional. Esto se puede ver como un efecto secundario a tener PC largas, lo que lleva a las consultas SMT a llevar mucho tiempo en resolverse como vimos en la figura 5.17. Además, al ser largas se generan varios timeouts / errores como se puede ver en la figura 5.21.

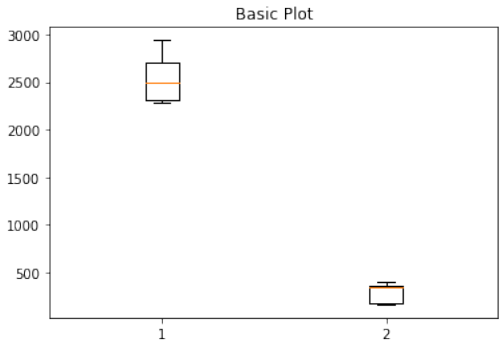

Fig. 5.19: Comparación sobre la cantidad de consultas que resultaron satisfacibles por el SMT solver para la clase DateTimeFormat.

.

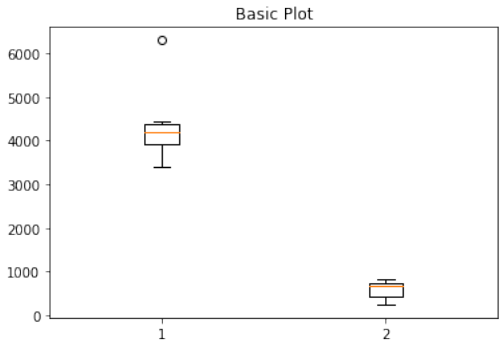

Fig. 5.20: Comparación sobre la cantidad de consultas que resultaron no satisfacibles por el SMT solver para la clase DateTimeFormat.

.

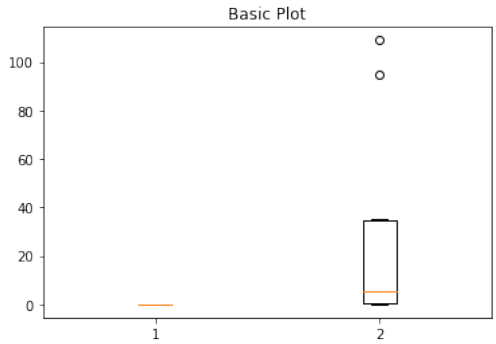

Fig. 5.21: Comparación sobre la cantidad de consultas que resultaron en error o timeout por el SMT solver para la clase DateTimeFormat.

.

Dada la complejidad de la clase y a la luz de los resultados anteriores, se puede ver como la búsqueda generacional recorre PC largas (ya que prioriza cobertura incremental), lo que lleva a tener una cantidad significativa de constraints a resolver sumado a tener poco hit rate en la cache por la variedad de PC recorridas. Sin embargo, el nuevo algoritmo logra una cantidad de cobertura relativamente considerable en comparación con el algoritmo legacy basándonos en la poca cantidad de PC recorridas, esto sugiere que la técnica está sujeta a un fuerte desarrollo en las optimizaciones de constraints, las cuales todavía no se encuentran implementadas en la herramienta. Esto resulta en una técnica con potencial para crecer, pero aún no escalable a programas mas complejos. Dado todo esto podemos decir que, sobre el corpus experimentado y a la espera de futura experimentación, pareciera ser mas eficiente el algoritmo legacy por el momento.

6. CONCLUSIONES

En este trabajo, por un lado separamos el módulo de ejecución simbólica dinámica del algoritmo genético existente en la herramienta además de implementar un nuevo algoritmo de búsqueda sobre el mismo. Por otro lado extendimos el poder expresivo del interprete simbólico para poder predicar sobre arreglos dando dos implementaciones sobre estos. Pudimos ver con la experimentación preliminar como la búsqueda generacional, si bien poderosa en la literatura, queda sujeta a fuertes optimizaciones del lado de las constraints por ser poco adaptativa al modelo de cache que propusimos y las optimizaciones actualmente implementadas. Por otro lado, las dos implementaciones de arreglos muestran la contracara entre un modelo de memoria preciso, pero costoso a la hora de resolver por un SMT solver contra uno con pérdida de completitud pero conceptualmente mas simple y fácil de implementar.

6.1. Trabajo Futuro

Como trabajo futuro, creemos que aun quedan muchas mejoras que se le pueden realizar a la herramienta, además de preguntas abiertas sobre las técnicas propuestas en este trabajo:

1. Extender el motor para predicar sobre objetos complejos. Esto se puede basar en la implementación provista en DSC[39]. Además de preguntas como ¿Qué impacto tiene en el tiempo de ejecución de la técnica?

2. Hacer un rework del algoritmo de búsqueda para intercambiar con algún criterio las estrategias durante la ejecución. Por lo que vimos en la literatura parece ser un área no explorada en lo que respecta a los algoritmos de búsqueda, esto trae varias preguntas como ¿Qué criterios se pueden tener en cuenta para intercambiar las estrategias? ¿Cómo impacta en la performance de la búsqueda el tipo de estrategia que se use? ¿Existen estrategias óptimas para casos particulares?

3. Optimizaciones de constraints:

   a) Completar los casos faltantes en **Counter-Example Cache**

   b) Agregar estrategias nuevas como **Constraint subsumption**

   c) ¿Qué optimizaciones mejorarían la performance de la búsqueda generacional?

4. Hacer un rediseño del heap simbólico para que se asemeje al de las VMs.

Bibliografía

[1] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 99, 01 2012.

[2] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20–27, January 2012.

[3] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369, 2013.

[4] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering (2. ed.)*. 01 2003.

[5] W. E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[9] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[12] Sang Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. pages 380–394, 05 2012.

[13] Nikolai Tillmann and Peli de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008.

[14] Dot net memory model documentation. `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref`. Accessed: 2020-07-08.

[15] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, page 419–423, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. Tesma and catg: Automated test generation tools for models of enterprise applications. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2:717–720, 2015.

[17] Karthick Jayaraman, David Harvison, and Vijay Ganesh. jfuzz: A concolic whitebox fuzzer for java.

[18] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. Jdart: A dynamic symbolic analysis framework. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[19] W. Visser, K. Havelund, G. Brat, and Seungjoon Park. Model checking programs. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 3–11, 2000.

[20] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Softw. Engg.*, 20(3):611–639, June 2015.

[21] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[22] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 147–158, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *Proceedings of the 2011 11th International Conference on Quality Software*, QSIC '11, page 31–40, USA, 2011. IEEE Computer Society.

[24] A. Arcuri, G. Fraser, and R. Just. Private api access and functional mocking in automated unit test generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 126–137, 2017.

[25] Edsger W. Dijkstra. Software engineering techniques. In *NATO Science Committee*, page 16, 1969.

[26] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[27] Java 8 specification. `https://docs.oracle.com/javase/8/`. Accessed: 2020-05-09.

[28] Java instrumentation api documentation. `https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html`. Accessed: 2020-05-10.

[29] Asm framework documentation. `https://asm.ow2.io/documentation.html`. Accessed: 2020-05-10.

[30] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Concolic fuzzing. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-12-21 16:38:57+01:00.

[31] Smt-lib website. `http://smtlib.cs.uiowa.edu/harp/language-reference/keywords/ref`. Accessed: 2020-07-08.

[32] Z3 website. `https://github.com/Z3Prover/z3`. Accessed: 2020-07-13.

[33] Cvc4 website. `https://cvc4.github.io/`. Accessed: 2020-07-13.

[34] Yices website. `https://yices.csl.sri.com/`. Accessed: 2020-07-13.

[35] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. November 2008.

[36] Ting Chen, Xiaodong Lin, Jin Huang, Abel Bacchus, and Xiaosong Zhang. An empirical investigation into path divergences for concolic execution using crest. *Security and Communication Networks*, 8:n/a–n/a, 06 2015.

[37] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 310–315, 2011.

[38] Soot website. `https://github.com/soot-oss/soot`. Accessed: 2020-07-13.

[39] Mainul Islam and Christoph Csallner. Dsc+mock: a test case + mock class generator in support of coding against interfaces. In *WODA '10*, 2010.

[40] Falk Howar, Fadi Jabbour, and Malte Mues. *JConstraints: A Library for Working with Logic Expressions in Java*, pages 310–325. 06 2019.

[41] Uml wikipedia. `https://en.wikipedia.org/wiki/Unified_Modeling_Language`. Accessed: 2020-07-13.

[42] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 362–369, 2013.

[43] Yu Liu, Xu Zhou, and Wei-Wei Gong. A survey of search strategies in the dynamic symbolic execution. *ITM Web of Conferences*, 12:03025, 01 2017.

[44] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29:1758–1773, 09 2013.

[45] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[46] Java statements example. `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html`. Accessed: 2020-05-09.

[47] Aymeric Fromherz, Kasper Luckow, and Corina Păsăreanu. Symbolic arrays in symbolic pathfinder. *ACM SIGSOFT Software Engineering Notes*, 41:1–5, 01 2017.

[48] Ivan Arcuschin Moreno. Una evaluación empírica del enfoque de sapienz para la generación automática de casos de test para aplicaciones android. master's thesis. departamento de computación, facultad de ciencias exactas y naturales, universidad de buenos aires, 2018.

[49] Javancss website. `https://www2.informatik.hu-berlin.de/swt/intkoop/jcse/tools/JavaNCSS%20-%20A%20Source%20Measurement%20Suite%20for%20Java.html#usage`. Accessed: 2020-08-20.

[50] Thesis extras github repository. `https://github.com/ilebrero/Tesis-LaFHIS/tree/master`. Accessed: 2020-05-05.