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

Automatizando tests en Go con EvoMaster

Tesis de Licenciatura en Ciencias de la Computación

Juan Cruz Basso

Director: Juan Pablo Galeotti

Buenos Aires, 2024

AUTOMATIZANDO TESTS EN GO CON EVOMASTER

En este trabajo se introduce un nuevo Driver de EvoMaster, una herramienta open source para generación automática de testeos impulsada por un algoritmo evolutivo, para aplicaciones web REST implementadas en el lenguaje de programación Go. Esta herramienta cuenta con dos componentes principales: un Core, encargado de llevar a cabo el algoritmo de búsqueda denominado MIO que aplica heurísticas para la optimización de testeos maximizando la cobertura de líneas y branches sobre el código de la aplicación a testear; y un Driver o controlador, encargado de alimentar al Core con las métricas necesarias para la evolución del algoritmo. Se describe el diseño e implementación de un controlador exclusivo para Go encargado de la comunicación con el Core bajo un protocolo ya establecido y la instrumentación de código mediante la aplicación de transformaciones sobre el árbol AST del código fuente original y la inyección del mismo al binario final en compilación. Además, como parte de este trabajo se extiende el core para soportar la escritura de los tests en lenguaje Go y se provee un paquete de Go para la distribución y testeo del nuevo controlador. Por último, se reportan y analizan métricas de cobertura de líneas obtenidas a partir de tests generados automáticamente para casos de prueba diferentes, tanto artificiales y como originales de terceros, disponibles públicamente en GitHub.

Palabras claves: REST, Testing, APIs Web, Algoritmos Genéticos, Metaheurísticas, Go.

GO TESTS AUTOMATION WITH EVOMASTER

In this work, we introduce a new Driver for EvoMaster, an open-source tool for automatic test generation powered by an evolutionary algorithm, specifically designed for REST web applications implemented in the Go programming language. This tool has two main components: a Core, responsible for executing the search algorithm known as MIO, which applies heuristics for test optimization by maximizing line and branch coverage in the application code under test; and a Driver or controller, which feeds the Core with the necessary metrics for the evolution of the algorithm. The design and implementation of an exclusive controller for Go is described, which handles communication with the Core under an established protocol and code instrumentation through the application of transformations on the AST (Abstract Syntax Tree) of the original source code and its injection into the final binary during compilation. Additionally, as part of this work, the core is extended to support test writing in Go, and a Go package is provided for the distribution and testing of the new controller. Finally, line coverage metrics obtained from automatically generated tests for different test cases, both artificial and third-party, publicly available on GitHub, are reported and analyzed.

Keywords: REST, Testing, Web APIs, Genetic Algorithms, Metaheuristics, Go.

AGRADECIMIENTOS

A mi director, JP, por proponerme este tema de tesis que me motivaba y por siempre estar presente para ayudarme.

A la UBA, por formarme académicamente y prepararme profesionalmente. A los profesores y ayudantes que pusieron su grano de arena para que hoy esté acá, de gran calidad humana y académica.

A mi familia. A mis padres por todo su apoyo y ayuda en este largo camino para poder estar donde estoy hoy. A mi madre, por preguntarme cada día sin falta cuándo me iba a recibir y recordarme que estudiara para poder lograrlo. A mi padre, por ayudarme a escoger esta hermosa carrera e interesarse en mi trayectoria en la misma. A mis hermanas, por todo el apoyo moral que me dieron y la ayuda en la casa. A mis abuelos, que a pesar de no entender muy bien mi carrera, siempre estuvieron interesados en mi graduación, en especial a mi abuelo Juan, que desafortunadamente no pudo verme recibirme, pero siempre le apasionó y le interesó la tecnología.

A mi novia, Mariana, que me brindo todo su amor y fuerzas para que pueda estudiar y enfocarme, ayudándome con todo lo que estuviese a su alcance.
A mi hijo de cuatro patas, Benito, por alegrarme mis días.

A mis compañeros de facultad, Brian Bohe, Ignacio Mariotti, Franco Frizzo y Francisco Figari, con quienes compartí muchos grupos de trabajo y me ayudaron incontables veces.
A todos los integrantes del grupo de "Walter y los 40 ladrones", que fueron un motor indispensable para poder avanzar en la carrera con motivación, grupos de estudio e incontables salidas para hacer de la etapa universitaria un gran camino.
A Ignacio Rodríguez y Julieta Luzzi, quienes también fueron una parte muy importante de mi etapa universitaria, con quienes compartí infinidad de buenos momentos, los cuales me ayudaron a tener la voluntad de seguir estudiando.

Índice general

1. INTRODUCCIÓN

Cada vez hay más microservicios que se comunican mediante HTTP APIs, como REST, GraphQL y GRPC. Una parte crítica del proceso es el testeo de estos servicios. Para ayudar a los desarrolladores a mejorar este proceso, existen herramientas que generan tests automáticos de las APIs de manera inteligente. Entre estas herramientas se encuentra EvoMaster.

EvoMaster [1] es la primera herramienta de código abierto impulsada por IA, desarrollada en 2016, que genera automáticamente tests a nivel de sistema para aplicaciones web y empresariales. EvoMaster trabaja abstrayéndose del lenguaje en el cual la API está implementada, lo que facilita su uso. Entre los controladores oficiales de EvoMaster se encuentran implementaciones para Java, C# y JavaScript, así como implementaciones de terceros, como la versión en Python [2].

EvoMaster se divide en dos enfoques: Black-Box y White-Box. El enfoque Black-Box permite generar tests automáticos sin necesidad de acceder al código fuente, mientras que el enfoque White-Box mejora la calidad de los tests utilizando información de cobertura de líneas, declaraciones y ramas. Este trabajo se basa en el enfoque White-Box, generando un Driver para instrumentar y realizar tests White-Box en Go para REST APIs específicamente.

Para realizar los tests White-Box, EvoMaster consta de dos componentes: el Core y los Drivers. El Core es responsable de ejecutar el algoritmo de búsqueda para la optimización de tests, buscando maximizar la cobertura de líneas, declaraciones y ramas. El Driver proporciona estas métricas al Core.

En este trabajo se creará un Driver en Go que utilizará el árbol AST del código fuente para inyectar información que informe si se ha pasado por un punto determinado y, en el caso de una rama, qué tan cerca estuvo de entrar al bloque contrario. Este nuevo código fuente será usado durante la compilación. También se realizarán modificaciones en el Core para generar los tests en Go.

Por último, se compararán los resultados de Black-Box y White-Box en los diferentes Drivers artificiales existentes actualmente, y se validarán en casos reales las mejoras que puede aportar EvoMaster.

2. MARCO TEÓRICO

2.1. REST HTTP APIs en Go

Una API REST (Representational State Transfer) [6] es un conjunto de principios de arquitectura para diseñar servicios web. Fue creado para guiar el diseño y desarrollo de la arquitectura de la Web. REST define un conjunto de restricciones sobre cómo debe comportarse la arquitectura de un sistema hipermedia distribuido a escala de Internet, como la Web. El estilo arquitectónico REST enfatiza interfaces uniformes, despliegue independiente de componentes, la escalabilidad de las interacciones entre ellos y la creación de una arquitectura en capas para promover el almacenamiento en caché, reducir la latencia percibida por el usuario, reforzar la seguridad y encapsular sistemas heredados.

Go, un lenguaje de programación desarrollado por Google, es conocido por su eficiencia y simplicidad, siendo una excelente opción para desarrollar servicios web escalables. La biblioteca estándar `net/http` es la herramienta principal en Go para construir servidores y clientes HTTP. A continuación, se presenta un ejemplo básico de una API REST en Go usando `net/http`:

```go
package main

import (
    "encoding/json"
    "net/http"
    "sync"
    "github.com/gorilla/mux"
)

var (
    items   = []string{}
)

type Response struct {
    Message string `json:"message"`
}

func addItemHandler(w http.ResponseWriter, r *http.Request) {
    var item string
    if err := json.NewDecoder(r.Body).Decode(&item); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    items = append(items, item)
    w.WriteHeader(http.StatusCreated)
}

func getItemsHandler(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(items)
}

func deleteItemHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
```

```
    item := vars["item"]
    for i, v := range items {
        if v == item {
            items = append(items[:i], items[i+1:]...)
            break
        }
    }
    w.WriteHeader(http.StatusNoContent)
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/items", addItemHandler).Methods("POST")
    r.HandleFunc("/items", getItemsHandler).Methods("GET")
    r.HandleFunc("/items/{item}", deleteItemHandler).Methods("DELETE")
    http.ListenAndServe(":8080", r)
}
```

En el ejemplo, se definen tres handlers de solicitudes HTTP que permiten agregar, obtener y eliminar elementos internos del servidor. La función `main` configura el servidor HTTP para escuchar en el puerto `8080` y manejar solicitudes para los siguientes endpoints:

- **POST /items:** Adicionar un item
- **GET /items:** Obtener todos los items
- **DELETE /items/{item}:** Borrar un item

2.1.1. Especificación OpenAPI

No existe una manera fácil de saber todos los endpoints y parámetros que se pueden enviar a un servidor para un cliente de una REST HTTP API. Para resolver esto, existen especificaciones como OpenAPI [5], que proporcionan una forma estándar de describir los servicios RESTful.

OpenAPI permite a los desarrolladores definir todos los endpoints disponibles, los parámetros de entrada y salida, y otra información relevante de manera estructurada y legible tanto para humanos como para máquinas, para que los clientes puedan consultar esta definición para saber cómo utilizar las APIs del servidor.

A continuación, se presenta un ejemplo de especificación OpenAPI para la API REST definida anteriormente:

```
openapi: 3.0.0
info:
  title: Items API
  version: 1.0.0
paths:
  /items:
    post:
      summary: Adicionar un item
      requestBody:
        required: true
        content:
          application/json:
```

```
            schema:
              type: string
      responses:
        '201':
          description: Item adicionado correctamente
        '400':
          description: Request erroneo
    get:
      summary: Obtener todos los items
      responses:
        '200':
          description: Una lista de items
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
        '400':
          description: Request erroneo
  /items/{item}:
    delete:
      summary: Borrar un item
      parameters:
        - in: path
          name: item
          required: true
          schema:
            type: string
          description: El item a borrar
      responses:
        '204':
          description: Item borrado correctamente
        '400':
          description: Request erroneo
```

Esta especificación suele exponerse en algún endpoint como podría ser un `GET /open-api`. Es decir:

```
//go:embed open_api.json
var openAPIyaml []byte

func getOpenAPI(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/yaml")
    w.Write(openAPIyaml)
}

func main() {
...
    r.HandleFunc("/open-api", getOpenAPI).Methods("Get")
...
}
```

En este ejemplo se obtiene una especificación de OpenAPI del archivo `open_api.yaml` y se expone en el endpoint correspondiente.

2.2. EvoMaster

EvoMaster es una herramienta de código abierto que genera automáticamente tests a nivel de sistema para aplicaciones web. Utiliza algoritmos evolutivos y análisis dinámico de programas para crear tests efectivos que maximicen la cobertura y detecten fallas. Se le llama a la Aplicación Web a ser testeada "System Under Test" (SUT).

El primer paso de EvoMaster es entender cuáles son todos los endpoints que el SUT expone. Para ello, necesita que el SUT exponga en algún endpoint la especificación de sus endpoints en el formato de OpenAPI presentado en la sección 2.1.1.

El *Core* de EvoMaster es el encargado de leer la especificación de los endpoints del SUT y, a partir de ellos, llevar a cabo el algoritmo de búsqueda evolutivo para generar los tests.

Dado que la especificación de OpenAPI no provee información sobre la cobertura del SUT de cada test, EvoMaster tiene un modo llamado White-Box, el cual se puede ejecutar sobre SUTs que son instrumentados y controlados por interfaces provistas por EvoMaster para diferentes lenguajes como *Java*, *C#* y *JavaScript*. Pudiéndose extender a otros lenguajes tal como fue hecho en el caso de *Python* [2].

Este modo ofrece una mejora significativa en términos de cobertura y velocidad en comparación con el modo Black-Box, que se utiliza para aplicaciones con código cerrado. Por lo tanto, en este trabajo, nos proponemos integrar el lenguaje *Go* para generar tests en modo White-Box.

2.2.1. Core

El *Core* de EvoMaster es responsable de ejecutar un algoritmo evolutivo para la generación de tests. EvoMaster contiene algunos algoritmos evolutivos, pero el más importante de ellos es el denominado MIO (Many Independent Objective). Este algoritmo busca optimizar la cobertura de código al generar tests que alcanzan diferentes objetivos como líneas de código, ramas de control y códigos de estado HTTP. A continuación, se describe en mayor detalle el funcionamiento del algoritmo MIO.

Algoritmo MIO

Para abordar la optimización de la cobertura de código en sistemas software, el algoritmo Many Independent Objective (MIO)[4] se presenta como una estrategia evolutiva empleada por EvoMaster. Este algoritmo tiene como objetivo principal maximizar la cobertura de distintos objetivos definidos dentro del código, tales como líneas de código, ramas de estructuras de flujo, y códigos de estado devueltos por endpoints.

El algoritmo MIO gestiona múltiples poblaciones de tests, cada una enfocada en alcanzar un objetivo específico de cobertura. Inicia con todas las poblaciones vacías y, en cada iteración, decide si generar un nuevo test aleatorio o mutar uno existente basado en una probabilidad R. Los tests generados son evaluados para determinar qué objetivos de cobertura cumplen. Aquellos tests que cubren un objetivo se incorporan al archivo de individuos optimizados A, y las poblaciones correspondientes se actualizan o eliminan según el desempeño de los tests en cada objetivo.

Para identificar los objetivos alcanzados por un nuevo individuo, el algoritmo usa resultados obtenidos a partir de ejecuciones de tests y mediciones de cobertura, como

el uso de métricas de líneas y ramas cubiertas. Estas métricas se obtienen mediante la instrumentación del código fuente, explicada detalladamente en la siguiente sección (2.2.2).

El proceso incluye una estrategia de "búsqueda enfocada", o focus search, donde se reduce la generación aleatoria de tests y se prioriza la mutación de tests existentes. Este enfoque es similar a técnicas como *Simulated Annealing*, equilibrando la exploración inicial del espacio de búsqueda con la explotación de soluciones prometedoras.

La función de fitness δ juega un papel crucial en el algoritmo MIO. Define la calidad o adecuación de un test en relación con el objetivo específico que se intenta cubrir. En el contexto de MIO, δ evalúa qué tan bien un test particular cumple con los requisitos del objetivo en términos de cobertura de código o cumplimiento de condiciones definidas. Por lo tanto, durante la ejecución del algoritmo, δ guía la selección y mutación de tests, favoreciendo aquellos que contribuyen más eficazmente a la optimización global de la cobertura de código.

Este pseudocódigo muestra la estructura y lógica del algoritmo MIO. Cada iteración evalúa la generación de nuevos tests o la mutación de tests existentes, manteniendo y actualizando poblaciones específicas para cada objetivo de cobertura. Al finalizar, el algoritmo devuelve un conjunto A de individuos que cubren los objetivos especificados.

Algorithm 1 Algoritmo Many Independent Objective (MIO)

Input: Condición de parada C, Función *fitness* δ, Tamaño de población N, Función de mutación m, Probabilidad de mutar individuo R, Comienzo de búsqueda enfocada F
Output: Archive de individuos optimizados A

$Z \leftarrow$ ConjuntoDePoblacionesVacias()
$A \leftarrow \{\}$
while $\neg C$ **do**
  if $R >$ Random$(0, 1)$ **then**
    $p \leftarrow$ GenerarIndividuoRandom()
  else
    $p \leftarrow$ SeleccionarIndividuoRandom(Z)
    $p \leftarrow$ Mutation(m, p)
  end if
  CalcularObjetivos(p, δ)
  for all $t \in$ ObjetivosAlcanzados(p) **do**
    if ObjectivoCubierto(t) **then**
      Actualizar(A, p)
      $Z \leftarrow Z \setminus \{Z_t\}$
    else
      $Z_t \leftarrow Z_t \cup \{p\}$
      if $|Z_t| > n$ **then**
        RemoverPeorTest(Z_t, δ)
      end if
    end if
  end for
  ActualizarParametros(F, R, N)
end while
return A

2.2.2. Driver

El *Driver* es el componente de EvoMaster encargado de instrumentar el código fuente de la aplicación a testear y de comunicar las métricas de cobertura al *Core*. La instrumentación implica modificar el código fuente para insertar puntos de monitoreo que registren en ejecución la cobertura de líneas, statements y branch distance durante los tests.

Otra de las tareas del Driver es exponer un controlador que funcione como una API HTTP REST, mediante la cual EvoMaster pueda comunicarse, obtener el estado interno de la instrumentación y gestionar el SUT. Los endpoints del controlador son los siguientes:

- `GET /controllerInfo`: Expone información general mínima del controlador e instrumentación, utilizada para generar los tests y para detectar que tipo de modo se puede ejecutar (WhiteBox o BlackBox).
- `PUT /runSUT`: Inicializa el SUT de manera correcta con ciertos parámetros.
- `GET /infoSUT`: Expone información del SUT y su instrumentacion que se utilizara para ejecutar el algoritmo evolutivo. Un ejemplo de esta información es la URL de OpenAPI, la URL base del SUT, número total de líneas, número total de branches entre otros.
- `POST /newSearch`: Indica que se está ejecutando un nuevo test que puede consistir de varias acciones, ejemplo: adicionar un item y borrar el item que fue adicionado. También la cobertura de los objetivos se reinicia.
- `PUT /newAction`: Señaliza que en una búsqueda de un test específico estamos ejecutando una nueva acción.
- `GET /testResults`: Retorna los resultados de cobertura, branch distance del test ejecutado y *Taint Analysis*.

Para que estos endpoints puedan gestionar el SUT, se suele exponer una interfaz a ser implementada por el SUT que lleve a cabo las acciones necesarias para poder inicializar y finalizar el SUT de manera correcta.

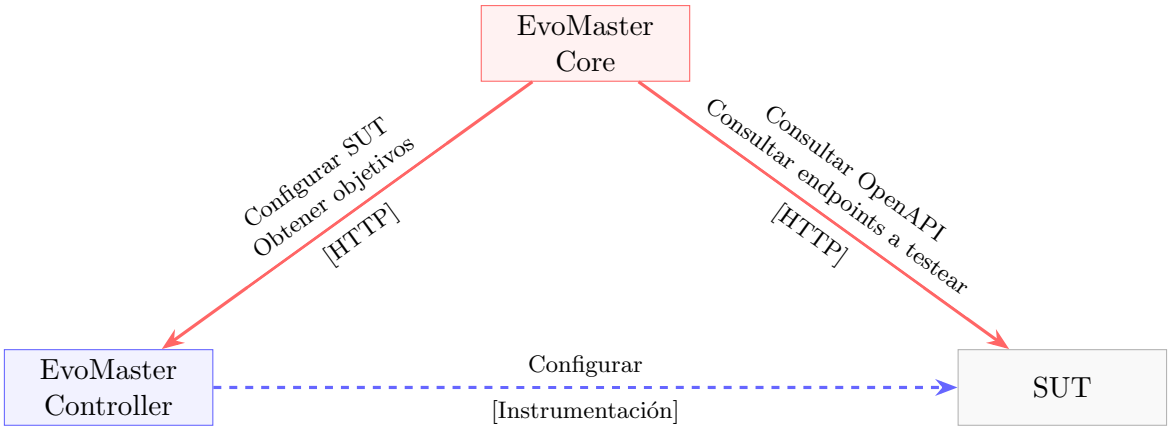

Fig. 2.1: Interacción entre EvoMaster y SUT

Cobertura

Hay 2 coberturas que son importantes para EvoMaster, la cobertura de líneas y la cobertura de *statements*.

La cobertura de líneas detalla qué líneas de código fueron visitadas en cada ejecución de un test. Por otro lado, se agrega la cobertura de *statements* para tener otra granularidad sobre el código que es a nivel de instrucción. En una línea puede haber varias instrucciones, por lo cual hacer esta distinción permite tener un mayor manejo de cobertura.

Branch Distance

El "branch distance" (distancia de rama) es una métrica que se utiliza para cuantificar qué tan cerca está una condición en una estructura de control, como un condicional o un bucle, de cambiar su estado. En otras palabras, mide la proximidad de una condición booleana de evaluar a verdadero o falso.

Esta métrica es particularmente útil para que EvoMaster pueda saber qué tan lejos está de poder entrar a ramas inexploradas, o ramas sobre las cuales querría iterar.

3. IMPLEMENTACIÓN

La integración de EvoMaster en Go fue estructurada en 2 partes desacopladas, el Driver y el Core. En las próximas subsecciones se explicara en detalle como fueron la integración de estos módulos.

3.1. Driver

3.1.1. Instrumentación

Es necesario modificar el SUT para agregar y guardar la información necesaria para obtener la cobertura de código y la distancia de ramas. Podemos dividir el problema en dos partes: primero, cómo agregar esta información en el código fuente, y segundo, cómo ejecutar automáticamente la modificación de ese código fuente.

Considerando que EvoMaster tiene varias integraciones, analizaremos estas integraciones antes de decidir cómo implementarlo en Go.

- `Java:` En función de que Java genera Java bytecode que se ejecuta por la JVM, se pueden realizar modificaciones sobre el mismo utilizando una biblioteca llamada ObjectWeb ASM[7]. Esta biblioteca hace bastante fácil la modificación del código a partir de la implementación de un "ClassVisitor", mientras que la modificación automática utiliza la interfaz "ClassFileTransformer"[8] que puede modificar código a ejecutar antes de crear las clases en la JVM.

- `C# .NET:` La implementación en C# es bastante parecida a la de Java, ya que los dos trabajan con máquinas virtuales y código intermedio que se ejecuta por ellas. C# posee la biblioteca Cecil[9] para modificar el código en formato ECMA CIL[10] que es ejecutada por el entorno de ejecución.

- `JavaScript:` Utiliza la biblioteca Babel[11] para modificar el código trabajándolo como si fuera un árbol AST[31] que lo modifica al momento de compilar.

- `Python:` En este caso se puede ver una estrategia parecida a la de JavaScript modificando el código, trabajándolo como un árbol AST utilizando la biblioteca estándar "ast"[12]. Se utiliza la funcionalidad de "Import Hooks"[13] para modificar el código de manera dinámica.

Teniendo en cuenta estas implementaciones y que Go [14] es un lenguaje estáticamente tipado y compilado, en las siguientes secciones se presentará la implementación realizada para su integración.

Modificar código fuente

Go contiene en su biblioteca estándar el paquete "ast"[15] que sirve para parsear y trabajar con árboles AST (Abstract Syntax Tree [31]). Esto convierte en un buen candidato para poder hacer la modificación del código fuente del SUT[1].

[1] Debido a que para Go es importante los comentarios en compilación debido a sus directivas[28], se utiliza la biblioteca DST[16] que contiene un mejor manejo de ellos que AST.

En las siguientes figuras se pueden apreciar la representación (simplificada) de un árbol AST según la implementación en Go y el código fuente asociado al mismo.

```
1 func compare(a int, b int) bool {
2     if a > b {
3         return true
4     } else {
5         return false
6     }
7 }
```

Fig. 3.1: Código fuente del árbol AST de Ejemplo

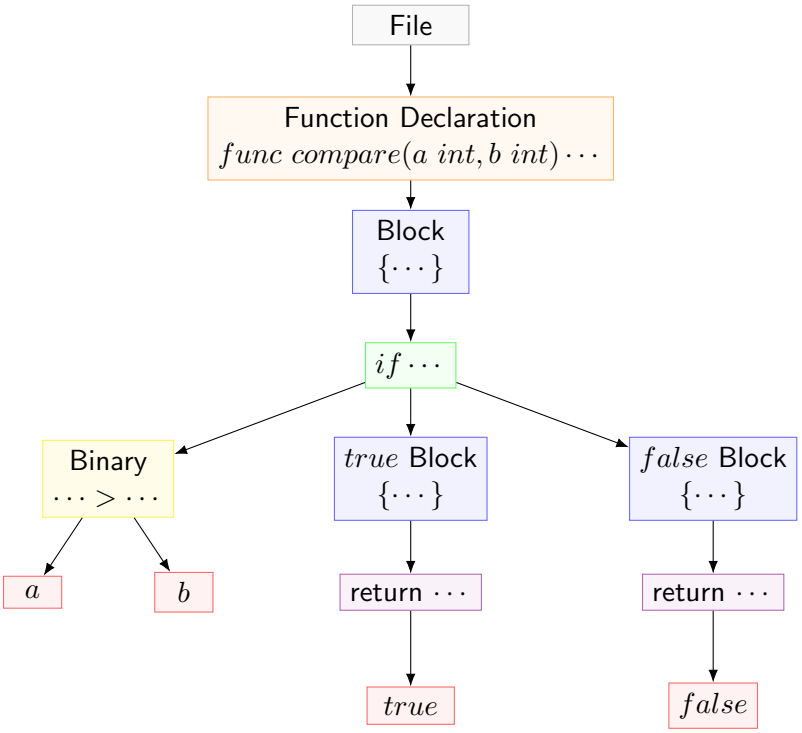

Fig. 3.2: Árbol AST de Ejemplo

Los colores de los nodos del árbol 3.2 corresponden a las palabras claves del mismo color del código 3.1. Es importante entender esta representación debido a que para instrumentar y agregar el seguimiento de Cobertura y Branch Distance reemplazaremos los nodos del árbol por otros que contengan, además del código original, código propio. Es por ello que el primer paso de la instrumentación, será él parseo del código fuente del SUT y, a partir de allí, toda la instrumentación se trabajará directamente con el árbol AST.

Registrar Objetivos:

La primera etapa de la instrumentación es el registro total de los Objetivos. Estos Objetivos son Archivos, Líneas, Sentencias y Ramas que forman parte del código fuente

del SUT. Si en el árbol AST nos encontramos con nodos de tipo *Statement* (como puede ser una asignación, una sentencia de tipo *if*, una sentencia de tipo *return*, un *for loop*, entre otras) se llama a la función RegisterTarget para registrar los objetivos. Esto también se hace para el registro de ramas, al localizar los nodos de comparaciones binarias (ejemplo: $<$, $>$, $==$, *and*) y los nodos de expresiones unarias (específicamente la negación, !). Por último, al detectar un nodo de tipo *File*, este se registrará.

El registro de los objetivos del ejemplo 3.1 es:

```
1  RegisterTarget(file_1_id)
2
3  func compare(a int, b int) bool {
4      RegisterTarget(line_1_id)
5      RegisterTarget(statement_1_id)
6      RegisterTarget(branch_1_id)
7      if a > b { // statement_1
8          RegisterTarget(line_2_id)
9          RegisterTarget(statement_2_id)
10         return true // statement_2
11     } else {
12         RegisterTarget(line_3_id)
13         RegisterTarget(statement_3_id)
14         return false // statement_3
15     }
16 }
```

Fig. 3.3: Ejemplo Registro de Objetivos teórico

La figura 3.3 muestra como, a partir de cada sentencia que va a procesar, se decide registrar el objetivo. Como se necesita tener esta información desde el inicio del SUT y el ejemplo solo sería ejecutado cuando la función es llamada, el registro se hace cuando se inicializa la aplicación (en la práctica significa agregar estas instrucciones en la función *_init_* en Go que se ejecuta en la inicialización). Lo que nos lleva a los registros demostrados en la figura 3.4.

```
1  func _init() {
2      RegisterTarget(file_1_id)
3      RegisterTarget(line_1_id)
4      RegisterTarget(statement_1_id)
5      RegisterTarget(branch_1_id)
6      RegisterTarget(line_2_id)
7      RegisterTarget(statement_2_id)
8      RegisterTarget(line_2_id)
9      RegisterTarget(statement_3_id)
10 }
11
12 func compare(a int, b int) bool {
13     ...
14 }
```

Fig. 3.4: Ejemplo Registro de Objetivos

Cobertura de Objetivos:

Los Objetivos se asocian a un valor entre 1 y 0, donde 1 significa que el objetivo fue cubierto.

La cobertura de los Objetivos se separa en los diferentes tipos de objetivos: Archivos, Líneas y Sentencias (las Ramas se estudian por separado utilizando Branch Distance). Si estamos en presencia de una sentencia se deben hacer las siguientes 3 actualizaciones:

- Guardar que el archivo fue cubierto (valor 1).
- Guardar que la línea fue cubierta (valor 1).
- Guardar la sentencia con un valor de 0.5 antes de ser ejecutada y de 1 luego de ser ejecutada exitosamente.

Las funciones encargadas de esto son EnteringStatement (encargada de hacer las 3 primeras actualizaciones) y CompletedStatement (encargada de actualizar la sentencia como cubierta).

```
1  func compare(a int, b int) bool {
2      EnteringStatement(file_1_id, line_1_id, statement_1_id)
3      CompletedStatement(file_1_id, line_1_id, statement_1_id)
4      if a > b { // statement_1
5          EnteringStatement(file_1_id, line_2_id, statement_2_id)
6          CompletedStatement(file_1_id, line_2_id, statement_2_id)
7          return true // statement_2
8      } else {
9          EnteringStatement(file_1_id, line_3_id, statement_3_id)
10         CompletedStatement(file_1_id, line_3_id, statement_3_id)
11         return false // statement_3
12     }
13 }
```

Fig. 3.5: Ejemplo Cobertura de Objetivos

En el ejemplo de la 3.5, ninguna de las sentencias se puede marcar como parcialmente procesada, pero en el caso de que hubiese una sentencia que no pudiese ser final, entonces se vería envuelta por las dos funciones, como podría ser el caso de una asignación.

Branch Distance:

La distancia de las ramas se define por 2 distancias que determinan que tan lejos se está de entrar a la rama `True` y la distancia de entrar a la rama `False`. Las distancias se normalizan en un valor entre 0 y 1, donde 1 indica que entró a dicha rama. Cabe destacar que uno de estos valores siempre será 1 porque deberá entrar en alguna de las 2 ramas.

Se calcula la distancia de ramas cuando hay un condicional del tipo: $=$, $\neq$, $>$, $\geq$, $<$, $\leq$, $\wedge$, $\vee$, $\neg$. Para poder detectar estos condicionales se instrumentarán los nodos de AST de expresión binaria y los de expresión unaria, validando las operaciones especificadas. Luego se registra la *branch distance* y se continúa con el funcionamiento normal del código.

Se definen las siguientes funciones para instrumentar los condicionales:

- CompareOrderable: Se llama en los nodos de expresión binaria donde el operando requiere que el tipo de dato sea ordenable: $>$, $\geq$, $<$, $\leq$, $\wedge$.
- CompareNonOrderable: Se llama en los nodos de expresión binaria donde el operando no requiere que el tipo de dato sea ordenable: $=$, $\neq$.
- And, Or: Se llama en los nodos de expresión binaria para los operandos $\wedge$ y $\vee$ respectivamente.
- Not: Se llama en los nodos de expresión unaria, donde el operando es la negación ($\neg$).

La distinción entre funciones de tipos ordenables y no ordenables es necesaria para el correcto funcionamiento debido a el fuerte tipado de Go.

Las expresiones que no contengan los operandos especificados, no serán instrumentadas.

Luego en el ejemplo instrumentado con *branch distance* es:

```
1  func compare(a int, b int) bool {
2      EnteringStatement(file_1_id, line_1_id, statement_1_id)
3      CompletedStatement(file_1_id, line_1_id, statement_1_id)
4      if CompareOrderable(a, b, >) { // Branch 1
5          // Branch 1 True
6          EnteringStatement(file_1_id, line_2_id, statement_2_id)
7          CompletedStatement(file_1_id, line_2_id, statement_2_id)
8          return true
9      } else {
10         // Branch 1 False
11         EnteringStatement(file_1_id, line_3_id, statement_3_id)
12         CompletedStatement(file_1_id, line_3_id, statement_3_id)
13         return false
14     }
15 }
```

Fig. 3.6: Ejemplo Instrumentación de Branch Distance

Definimos "*Truthness*" a la distancia a ambas ramas al evaluar una comparación y separamos las heurísticas de distancia en los siguientes tipos de datos:

- `float`, `int` y `uint` con sus tipos para diferentes tamaño de bits. Llamaremos a este grupo de tipos `number` para simplificar *(siendo estos los tipos de datos ordenables)*.
- `string`

Sea:

$$x_1, x_2 \in T$$

$$T \in \{number,\ string\}$$

definimos las siguientes heurísticas:

$$Truthness_{ofTrue}(x_1, x_2, =) = \begin{cases} 1 & \text{si } x_1 = x_2 \\ 1 - norma(distancia_T(x_1, x_2)) & \text{si } x_1 \neq x_2 \end{cases}$$

$$Truthness_{ofFalse}(x_1, x_2, =) = \begin{cases} 0 & \text{si } x_1 = x_2 \\ 1 & \text{si } x_1 \neq x_2 \end{cases}$$

$$Truthness_{ofTrue}(x_1, x_2, <) = \begin{cases} 1 & \text{si } x_1 < x_2 \\ 1 - norma(distancia_T(x_1, x_2)) & \text{si } x_1 \geq x_2 \end{cases}$$

$$Truthness_{ofFalse}(x_1, x_2, <) = \begin{cases} 1 - norma(distancia_T(x_1, x_2)) & \text{si } x_1 < x_2 \\ 1 & \text{si } x_1 \geq x_2 \end{cases}$$

definimos la función `distancia`:

$$distancia_{number}(n_1, n_2) = |n_1 - n_2| \qquad n_1, n_2 \in number$$

$$distancia_{string}(s_1, s_2) = |s_1.length - s_2.length| \cdot C + \sum_{i=0}^{min(s_1.length, s_2.length)} |s_{1_i}.ord - s_{2_i}.ord| \qquad s_1, s_2 \in string$$

Y la función `norma`[34] de la siguiente manera:

$$norma(x) = \frac{x}{x+1} \qquad x \in R$$

Definimos la función auxiliar `invertir` como:

$$invertir(Truthness_{ofTrue}(x_1, x_2, Op)) = Truthness_{ofFalse}(x_1, x_2, Op)$$

$$invertir(Truthness_{ofFalse}(x_1, x_2, Op)) = Truthness_{ofTrue}(x_1, x_2, Op)$$

que nos permite definir el *Truthness* para el resto de las operaciones:

$$Truthness(x_1, x_2, \neq) = invertir(Truthness(x_1, x_2, =))$$
$$Truthness(x_1, x_2, \geq) = invertir(Truthness(x_1, x_2, <))$$
$$Truthness(x_1, x_2, \leq) = invertir(Truthness(x_2, x_1, <))$$
$$Truthness(x_1, x_2, >) = invertir(Truthness(x_1, x_2, \leq))$$

Se aplicará una heurística en las expresiones *unarias* donde simplemente invertirá el valor de *Truthness* de la expresión alcanzada por el operador. En caso de tratarse de una negación de una constante o una variable, no se computarán *branch distances* para dicha operación.

Por último, en el caso de expresiones *binarias*, como son el `And` y el `Or`, se aplicará una heurística definida a partir del *Truthness* de cada uno de los dos operandos.

Sean:

$$x, y \in bool$$

definimos:

$$Truthness_{ofTrue}(x, y, And) = \frac{T_{ofTrue}(x)}{2} + \frac{T_{ofTrue}(y)}{2}$$

$$Truthness_{ofFalse}(x, y, And) = max\{T_{ofFalse}(x), T_{ofFalse}(y)\}$$
$$Truthness_{ofTrue}(x, y, Or) = max\{T_{ofTrue}(x), T_{ofTrue}(y)\}$$

$$Truthness_{ofFalse}(x, y, Or) = \frac{T_{ofFalse}(x)}{2} + \frac{T_{ofFalse}(y)}{2}$$

Al comparar otros tipos de datos, o comparar cadenas de caracteres con valores numéricos por igualdad, la implementación actual no permite calcular un valor de *branch distance* y simplemente se retorna un *Truthness* en base al resultado booleano de la operación, es decir:

$$Truthness_{ofTrue}(x, y, Op) = \begin{cases} 1 & \text{si } Op(x, y) \\ 0 & \text{si } \neg Op(x, y) \end{cases}$$

$$Truthness_{ofFalse}(x, y, Op) = \begin{cases} 0 & \text{si } Op(x, y) \\ 1 & \text{si } \neg Op(x, y) \end{cases}$$

Taint Analysis:

El *Taint Analysis* es el análisis específico en la comparación del tipo de dato *string*. Este análisis se separa en 2 partes:

- Comparación entre un argumento *string* y una constante *string*.
- Comparación entre dos argumentos de tipo *string*.

Se ejecuta solo cuando algún argumento en el endpoint es *string* y cumple la expresión regular: `^(?i)_EM_\d+_XYZ_$`. Un ejemplo que cumpla esta expresión es el *string* `_EM_1234_XYZ_`.

Definimos la función HandleTaintForStringEquals que será llamada cuando se instrumente la comparación por igualdad en los *strings* en el algoritmo 2.

La información del *Taint Analysis* es consultado por el *Core* al *Driver* al llamar al endpoint `GET /infoSUT`, donde también recibe los demás objetivos.

Consideraciones en el tipado:

Go es un lenguaje que en ciertos casos infiere el tipo. Esto ocurre, por ejemplo, con el valor `nil` y los números con sus tipos específicos (`uint`, `int`, `float`).

Algunos ejemplos de inferencia de tipos son los siguientes casos:

$$a = 35$$

Donde dependiendo del tipo de a se infiere el tipo de 35, es decir si a es `uint`, 35 también lo será. Esto es importante ya que la igualdad sobre diferente tipos de datos no está definida en Go.

Algorithm 2 Handle Taint Analysis

Input: Argumento string s_1, Argumento string s_2

```
1: if TAINTED(s1) ∧ TAINTED(s2) then
2:     if s1 = s2 then
3:         return
4:     end if
5:     ADDSTRINGSPECIALIZATION(s1, "EQUAL", s1 + "___" + s2)
6:     ADDSTRINGSPECIALIZATION(s2, "EQUAL", s1 + "___" + s1)
7:     return
8: end if
9: if TAINTED(s1) then
10:    ADDSTRINGSPECIALIZATION(s1, "CONSTANT", s2)
11: else if TAINTED(s2) then
12:    ADDSTRINGSPECIALIZATION(s2, "CONSTANT", s1)
13: end if
```

$$b = nil$$

Por otro lado, al comparar un `struct` b, se infiere `nil` al mismo tipo. Por ejemplo, para el tipo de dato `*log.Logger`, `nil` debe ser inferido como `*log.Logger(nil)`.

Estas inferencias son perdidas al momento de instrumentar y llamar a una función como intermediario a la resolución de las comparaciones, ya que no se puede inferir tipo de dato para el 35 y el `nil`.

Se resolvieron estos casos de la siguiente manera:

- `nil`: Se utiliza la biblioteca `reflect`[22] para saber si los argumentos de la comparación son `nil` sin importar su tipo, y de ser ambos `nil` se retorna `True`, de ser uno solo `False`, y en caso contrario se realiza la comparación estándar.
- `number`: En el caso de los números, de haber una discrepancia entre ellos (por ser inferido uno de ellos al estándar `int` de manera errónea) se castea los mismos al tipo de dato que no es `int`.

Estas particularidades hacen que un código fuente que no compilaba empiece a compilar luego de la instrumentación. Esto no es considerado un error de la instrumentación debido a que es responsabilidad del dueño del código fuente validar que el mismo compile, sin importar la instrumentación.

Modificar automáticamente código fuente

Una vez que se haya modificado el código para permitir la instrumentación, queda resolver cómo realizar la modificación del código fuente de manera automática. Una posible solución es la generación automática de código, una práctica común y popular en Go. De hecho, el propio lenguaje proporciona el comando `go generate` [29] para facilitar este proceso. Algunos ejemplos de herramientas que utilizan generación de código son:

- `uber-go/mock:`[17] Herramienta encargada de generar mocks de interfaces. Al utilizar una directiva como `//go mockgen` $\cdots$ en el código, los usuarios pueden especificar qué interfaces deben ser implementadas por la herramienta para ser utilizadas

como mocks. Dependiendo de su configuración, esta herramienta crea las implementaciones en un nuevo archivo `.go` dentro del paquete donde se encuentra la interfaz.

- `golang/protoc-gen-go:[18]` Herramienta encargada de generar código Go para especificaciones de "Protocol Buffers" [32] e interfaces de servidores y clientes RPC a implementar. Para utilizarlo se usa la directiva `//go:generate protoc` $\cdots$ exportando las interfaces y *structs* en código Go.

Un requerimiento importante de la modificación de código fuente es que se pueda modificar paquetes importados como puede ser el paquete `strings`[23] (paquete para facilitar el trabajo de *strings*) u otros paquetes estándar de Go. Esto da mayor flexibilidad para mejoras de *Branch Distance* para tipos de datos específicos. Por lo tanto, la generación de código no es una solución adecuada para garantizar la extensibilidad.

Una solución más adecuada para este requerimiento es adicionar la instrumentación en tiempo de compilación, permitiendo así instrumentar las bibliotecas importadas. En Go, es posible enviar diferentes opciones o banderas durante la compilación, y una de estas opciones es la bandera `-toolexec` [30]. Al proporcionar un ejecutable como argumento, durante la compilación (comando `go build`), los subcomandos que deben ejecutarse para compilar todos los paquetes se envían a dicho ejecutable, que luego debe encargarse de ejecutarlos. Es decir, al utilizar el comando `go build -toolexec=myTool .`, el ejecutable `myTool` será responsable de ejecutar correctamente los comandos para compilar todas las bibliotecas.

Entonces, podemos crear un ejecutable llamado `instrumenter` que, al recibir un subcomando de compilación (por ejemplo, `compile ./file_1.go ./file_2.go`), primero lea e instrumente todos los archivos del paquete (como se explicó en la sección 3.1.1). Luego, guardará estos archivos instrumentados en `./instrumented/file_1.go` y `./instrumented/file_2.go`, y finalmente, en lugar de compilar los archivos originales, compilará los archivos instrumentados ejecutando el comando:

```
compile ./instrumented/file\_1.go ./instrumented/file\_2.go
```

Se puede apreciar en el algoritmo 3 el pseudocódigo de un instrumentador. El instrumentador seria llamado por cada paquete a compilar al ejecutar la herramienta `go build -toolexec=instrumenter .`, para que así, un usuario final pueda utilizarlo para instrumentar su SUT.

Algorithm 3 Instrumentador

Input: Comando *cmd*, Argumentos del comando $args$
Output: Código de Error

1: $fs' \leftarrow \{\}$
2: **for all** $f \in$ ARCHIVOS($args$) **do**
3: $\quad f_{ast} \leftarrow \text{AST}(f)$
4: $\quad f'_{ast} \leftarrow$ INSTRUMENTAR(f_{ast})
5: $\quad f' \leftarrow$ ESCRIBIRARCHIVO(f'_{ast})
6: $\quad fs' \cup \{f'\}$
7: **end for**
8: $args \leftarrow$ ACTUALIZARARGUMENTOS($args, fs'$)
9: **return** EJECUTAR($cmd, args$)

3.1.2. Controlador

El Controlador es el encargado de que el Core de EvoMaster sea capaz de comunicarse con el SUT ya instrumentado previamente. Este objetivo se consigue levantando una API REST HTTP que exponga los servicios necesarios explicados en el Marco Teórico 2.2.2 y siendo capaz de comunicarse con el SUT instrumentado.

Se busca que la interfaz del controlador a implementar sea lo más básica posible, mientras que al mismo tiempo sea bastante flexible. Se creó la interfaz `SutControllerInterface` que define las siguientes funciones a ser implementadas:

- StartSUT(): Encargado de inicializar la REST API del SUT. Devuelve la URL y el puerto donde fue inicializada la API.
- StopSUT(): Encargado de parar la REST API del SUT.
- ResetStateOfSUT(): Encargado de limpiar correctamente los estados del SUT (como por ejemplo una la base de datos)

Queda entonces en el usuario del SUT la decisión de qué biblioteca utilizar para inicializar la API, pero se alienta a tener una función que genere el server que se pueda reutilizar tanto para el controlador como para la aplicación. En la figura 3.7 se puede ver un ejemplo con un servidor que utiliza la biblioteca `net/http`[19].

```
type AppController struct {
        host   string
        port   int
        server *http.Server
}

func (a *AppController) StartSut() string {
        addr := fmt.Sprintf("%s:%d", a.host, a.port)
        a.server = &http.Server{
                Addr:    addr,
                Handler: CreateServer(),
        }

        go func() {
                a.server.ListenAndServe()
        }()

        return addr
}

func (a *AppController) StopSut() {
        a.server.Shutdown(ctx)
}

func (a *AppController) ResetStateOfSUT() {
        // No state to reset
        return
}
```

Fig. 3.7: Implementación de Controlador

Debido a que el ejemplo utiliza la función CreateServer para crear el servidor y el tipo estándar de Go `http.Server`, este ejemplo puede ser reutilizado de manera muy simple con solo implementar la función.

3.2. Core

El Core de EvoMaster busca abstraerse del SUT y el lenguaje de programación del mismo, pero al momento de finalizar la evaluación de un SUT se quiere generar tests en el mismo lenguaje del SUT, de no hacerse, el costo de configuración inicial y mantenimiento para poder generar los tests sería alto. Es por ello que a continuación se explicará configuración para la escritura de test en Go.

3.2.1. Escritura de tests

EvoMaster expone una serie de clases que definen, dependiendo del tipo de test a escribir, la generación de los test en el lenguaje de salida. Esta configuración es bastante verbosa, con diferentes ramas y casos en su lógica.

Para los test se utilizó un esquema de "test suite" en el cual se definen funciones para correr antes y después de la clase de test (llamada *Suite*) y antes y después de cada test de la misma inicializando y configurando el SUT. El manejo de los test en formato de "suite" es una idea que ya es utilizada en los lenguajes con extensiones. Se utilizó la biblioteca "stretchr/testify"[20], para el manejo de suite en conjunto con la biblioteca estándar de testing de Go[21].

En cuanto a la escritura de cada uno de los tests, se utilizaron las siguientes bibliotecas:

- `net/http`:[19] Utilizada para poder hacer el envío de consultas a las REST API del SUT.
- `stretchr/testify`:[20] Utilizada para ejecutar cada test y poder hacer las aserciones sobre ellos.
- `alibaba/fastjson`:[25] Utilizada para poder trabajar los JSONs de manera fácil como un *objeto* y no un *string*, siguiendo los estándares del Core.

```
// Run Test Suite
func TestEvoMasterTestSuite(t *testing.T) {
    suite.Run(t, new(EvoMasterTest))
}


// Tests Structure
type EvoMasterTests struct {
    suite.Suite
    Controller *AppController
    BaseUrlOfSut string
}

//-- Before the whole suite
func (suite *EvoMasterTests) SetupSuite() {
    suite.Controller = &AppController{}
    suite.BaseUrlOfSut = suite.Controller.StartSut()
}

//-- After the whole suite
func (suite *EvoMasterTests) TearDownSuite() {
    suite.Controller.StopSut()
}

//-- Before each test
func (suite *EvoMasterTests) SetupTest() {
    suite.Controller.ResetStateOfSUT()
}


// Tests
func (suite *EvoMasterTests) Test_1() {
...
}

...
```

Fig. 3.8: Configuracion de tests

Los siguientes tests ejemplifican el diseño de los mismos, definiendo primero la consulta con sus "headers" y el cuerpo de la consulta si es necesario. Al final, se valida el código de respuesta y se comprueba que el cuerpo de la respuesta sea correcto.

```
1  func (suite *EvoMasterSuccessesTest) Test_0() {
2          method := http.MethodGet
3          reqUrl := "http://" + suite.BaseUrlOfSut + "/api/get"
4
5          req, err := http.NewRequest(method, reqUrl, nil)
6          suite.Nil(err, "Request creation error must be nil")
7
8          req.Header.Set("x-EMextraHeader123", "")
9          res_0, err := http.DefaultClient.Do(req)
10         suite.NoError(err, "Request error must be nil")
11         defer res_0.Body.Close()
12
13         suite.Equal(200, res_0.StatusCode)
14         suite.True(strings.HasPrefix(res_0.Header.Get("Content-Type"),
           ↪  "application/json"))
15         body_1, err := io.ReadAll(res_0.Body)
16         suite.NoError(err, "Request body read error must be nil")
17
18         var p fastjson.Parser
19         v_body_1, err := p.ParseBytes(body_1)
20         suite.NoError(err, "Parser body creation error must be nil")
21         suite.NotNil(v_body_1, "Parsed body must not be nil")
22         suite.Equal("ok", string(v_body_1.GetStringBytes("success")))
23 }
```

Fig. 3.9: Test usando método `GET`

```
func (suite *EvoMasterSuccessesTest) Test_1() {
        method := http.MethodPost
        reqUrl := "http://" + suite.BaseUrlOfSut + "/api/push"

        body := " { \"some_body\": 123} "
        req, err := http.NewRequest(method, reqUrl, bytes.NewBufferString(body))
        suite.Nil(err, "Request creation error must be nil")
        req.Header.Add("Content-Type", "application/json")

        res_0, err := http.DefaultClient.Do(req)
        suite.NoError(err, "Request error must be nil")
        defer res_0.Body.Close()

        suite.Equal(200, res_0.StatusCode)
        suite.True(strings.HasPrefix(res_0.Header.Get("Content-Type"),
        ↪  "application/json"))
        body_1, err := io.ReadAll(res_0.Body)
        suite.NoError(err, "Request body read error must be nil")

        var p fastjson.Parser
        v_body_1, err := p.ParseBytes(body_1)
        suite.NoError(err, "Parser body creation error must be nil")
        suite.NotNil(v_body_1, "Parsed body must not be nil")
        suite.Equal(float64(13.0), v_body_1.GetFloat64("counts"))
        suite.Len(v_body_1.GetArray("logs"), 0)
        suite.Equal("ok", string(v_body_1.GetStringBytes("success")))
}
```

Fig. 3.10: Test usando método `POST`

4. EVALUACIÓN

Al evaluar el rendimiento del Driver en Go se tomaron dos enfoques diferentes, tomando como casos de estudio SUTs creados artificialmente para la validación y tomando como casos de estudio aplicaciones Open Source.

Para comparar el rendimiento de los drivers, se evaluará la cobertura total de líneas obtenida en los tests en modo Black Box y en modo White Box (es decir, utilizando la instrumentación).

4.1. Casos de Estudio Artificiales

EvoMaster posee algunos casos de estudio que se utilizaron para poder validar el rendimiento de sus Drivers y compararlos contra el de otros lenguajes. Estos casos de estudio pueden encontrarse en el repositorio "EMB"[3]. Entre los casos de estudio implementados en todos los lenguajes se encuentran: *Numerical Case Study* (NCS) y *String Case Study* (SCS). En el caso de Go se implementaron cumpliendo no solo con su especificación de OpanAPI, sino también intentando mantener la lógica lo más consistente con su contrapartida en los demás lenguajes.

Cada caso de estudio se analizará en la cantidad de líneas de cobertura en sus dos modos para ese mismo lenguaje, es decir:

$$Performance_{lang} = \frac{Coverage(TestsWhiteBox_{lang})}{Coverage(TestsBlackBox_{lang})} - 1 \tag{4.1}$$

Cuanto mayor es $Performance_{lang}$, mejor es el rendimiento que tuvo la generación de test White Box por sobre los Black Box.

Al momento de crear los test se ejecutaron 5 corridas para cada tiempo con diferentes *Seeds* específicas para que sea lo más determinístico posible.

En cada caso de estudio se comparan para todos los lenguajes el rendimiento en un mismo tiempo fijo, comparándolo con la cobertura de los test generados por Black Box *(figuras 4.2 y 4.3)*. En el caso de Go, también se compara el rendimiento de los tests White Box utilizando diferentes tiempos límites *(figura 4.1)*.

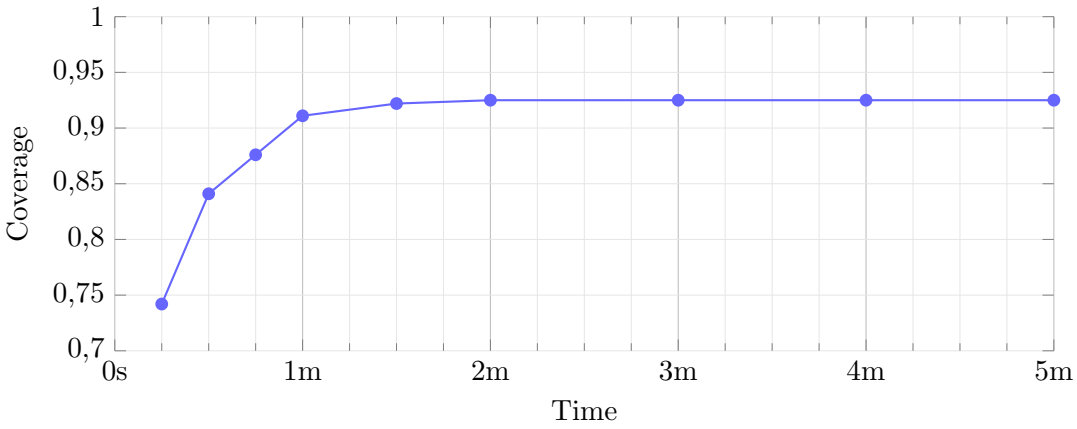

Fig. 4.1: Rendimiento de White Box de NCS en diferentes tiempos

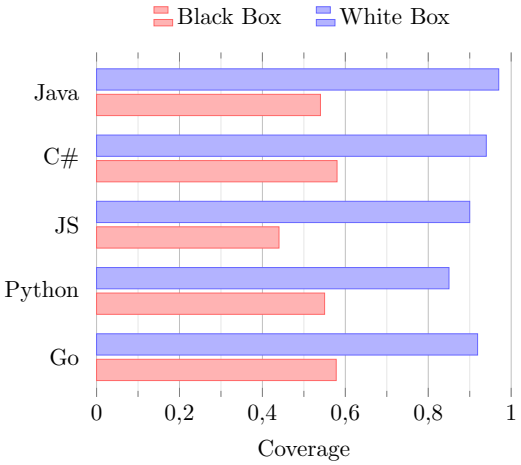

Fig. 4.2: Cobertura NCS

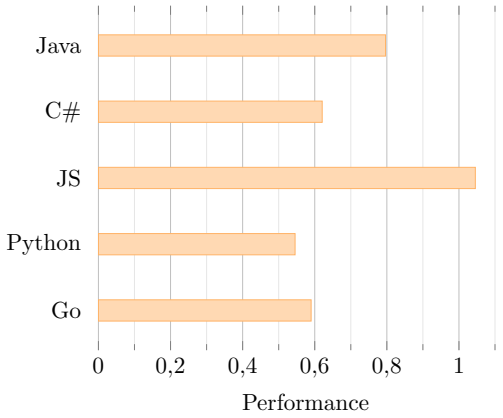

Fig. 4.3: Rendimiento NCS

Se puede ver como, a partir del primer minuto, los cambios en la cobertura de los tests ya no varía demasiado.

Al analizar el rendimiento del SCS, dado que se trata de un caso de estudio de *string*, el *Taint Analysis* juega un papel crucial. Por lo tanto, al evaluar su rendimiento, se compararán los enfoques Black Box y White Box, y además se distinguirán las soluciones de White Box que utilizan *Taint Analysis* de aquellas que no lo utilizan. Esto permitirá entender el impacto del *Taint Analysis* y verificar si funciona de manera correcta.

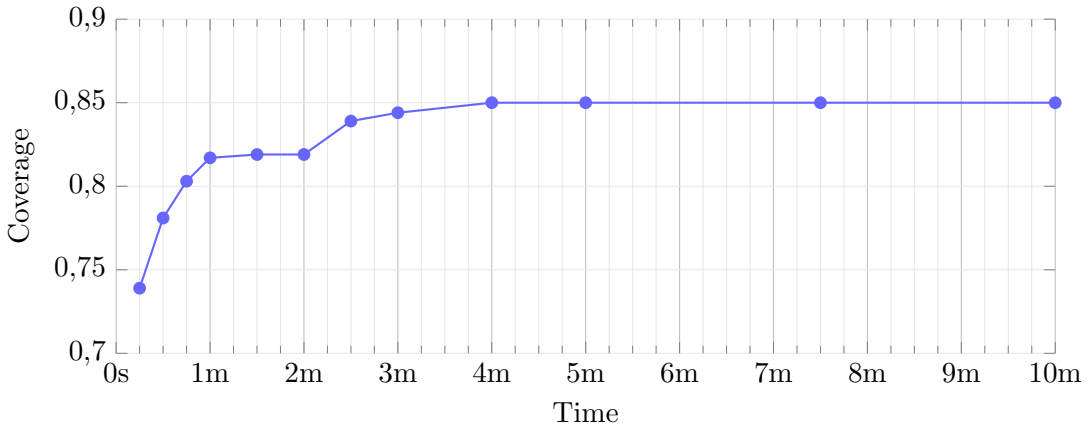

Fig. 4.4: Rendimiento White Box + Taint de SCS en diferentes tiempos

En este caso de estudio se puede ver también una curva muy parecida a la del *NCS*, con un salto a los 2 minutos.

4.2. Casos de Estudio Reales

Al momento de escoger casos de estudio reales se buscaron REST HTTP APIs que fuesen tanto fácil de modificar como de correr, y de estas buscar las APIs con mayor audiencia / usuarios y que posean especificación OpenAPI.

Las APIs escogidas fueron:

- `CocaineCong/TodoList[27]4.2.1`: Aplicación para el manejo de lista de tareas.

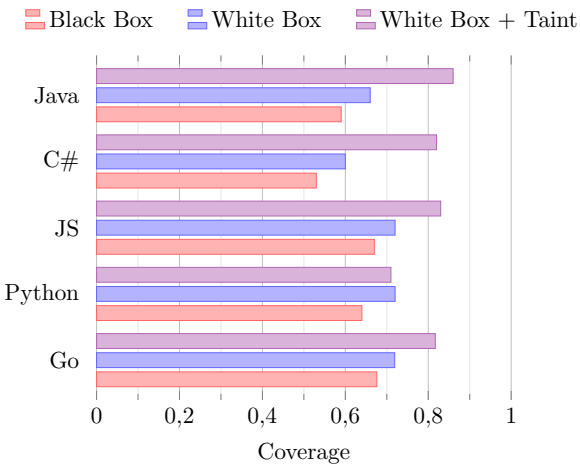

Fig. 4.5: Cobertura SCS

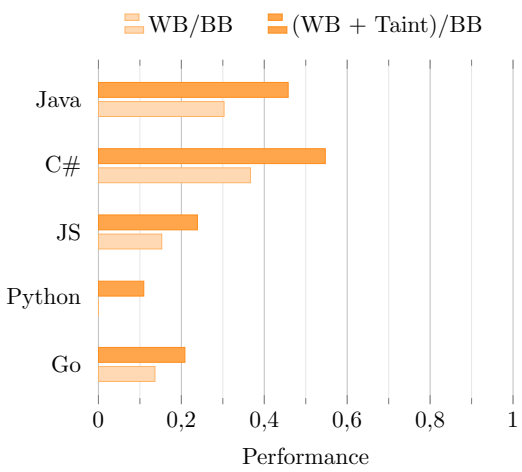

Fig. 4.6: Rendimiento SCS

- `hyperonym/ratus[26]4.2.2`: Aplicación para manejo de cola de tareas asincrónicas.

4.2.1. CocaineCong/TodoList

61 ★ 199

Es un servicio REST simple para hacer seguimiento de *TO DOs*[1]. El servicio cuenta con varios endpoints para poder administrar la lista de tareas en forma de `CRUD`[33]. Para poder guardar la información utiliza la base de datos MySQL y Redis para el manejo de tokens de autenticación.

Los endpoints para administrar las tareas son:

- `POST /task/create`: Crear tarea.
- `PATCH /task/update`: Actualizar tarea dado su *id*.
- `DELETE /task/delete`: Eliminar tarea dado su *id*.
- `GET /task/show`: Obtener tarea dado su *id*.
- `GET /task/list`: Obtener tareas en páginas.
- `POST /task/search`: Buscar tareas a partir de su información.

Durante la generación de los tests, se realizaron 5 ejecuciones para cada tiempo, utilizando diferentes *seeds*.

Como se trata de una aplicación CRUD, la parte más importante no está en la lógica del código, sino en la base de datos. Por esta razón, no se observó una mejora clara en la cobertura, ya que EvoMaster no logró encontrar la combinación de consultas específicas para realizar operaciones de creación y borrado o creación y obtención de información.

De haber implementado la instrumentación con la integración de la Base de Datos se podría haber conseguido mejores resultas en esta caso de estudio.

[1] TO DOs: Lista de tareas pendientes

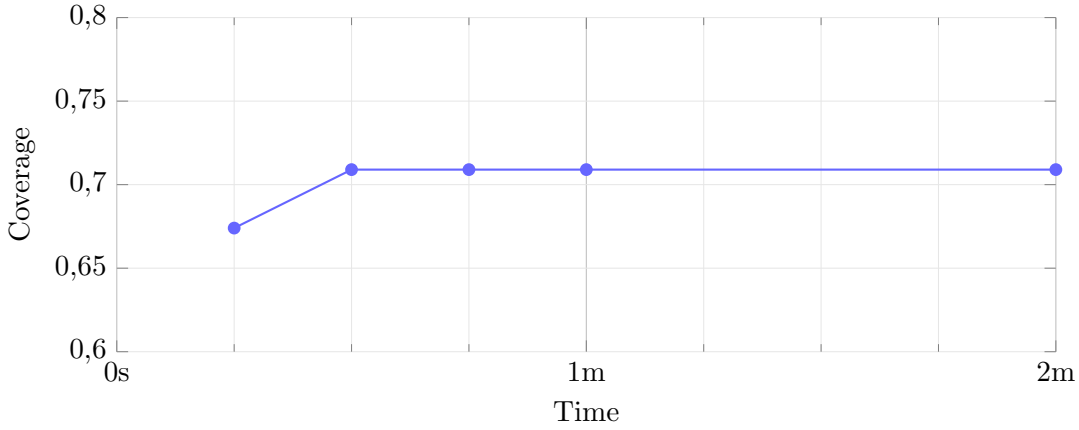

Fig. 4.7: Cobertura White Box de `TodoList` en diferentes tiempos

4.2.2. hyperonym/ratus

7 105

Servidor de cola de tareas asincrónicas basado en REST. Traduce los conceptos de colas de tareas distribuidas en un conjunto de recursos que se ajustan a los principios REST y proporciona una API HTTP coherente para múltiples backends.

La aplicación introduce las siguientes entidades

- *Task:* hace referencia a una unidad de trabajo idempotente que debería ser ejecutada de manera asincrónica.
- *Topic:* conjunto ordenado de *tasks* con la misma propiedad *topic*.
- *Promise:* representa un pedido de *ownership* sobre una tarea activa.

Los endpoints nos ayudarán a trabajar con estas entidades, en donde un uso básico podría ser el siguiente:

1. Creamos una *task* (1) a ser ejecutada en el *topic* `example`.

```
curl -X POST -d '{"payload": "hello world"}' "localhost:8000/topics/example/tasks/1"
```

2. Generar una *promise* de la próxima tarea del *topic*.

```
# Request
curl -X POST "localhost:8000/topics/example/promises?timeout=30s"

# Response
{
        "_id": "1",
        "topic": "example",
        "state": 1,
        "nonce": "e4SN6Si1nOnE53ou",
        "produced": "2022-07-29T20:00:00.0Z",
        "scheduled": "2022-07-29T20:00:00.0Z",
        "consumed": "2022-07-29T20:00:10.0Z",
        "deadline": "2022-07-29T20:00:40.0Z",
```

```
        "payload": "hello world"
}
```

3. Por último informar que la *task* fue ejecutada de manera correcta.

```
curl -X PATCH "localhost:8000/topics/example/tasks/1"
```

Además de los del ejemplo, posee endpoints *CRUD* para todas sus entidades. Para el manejo de ellas admite en su infraestructura trabajar con MongoDB o memoria. A fines de esta prueba se utilizó la configuración de memoria.

Durante la evaluación, se utilizó el mismo esquema que en `CocaineCong/TodoList`4.2.1, ejecutando pruebas para diferentes tiempos con 5 *seeds* y midiendo la cobertura de líneas.

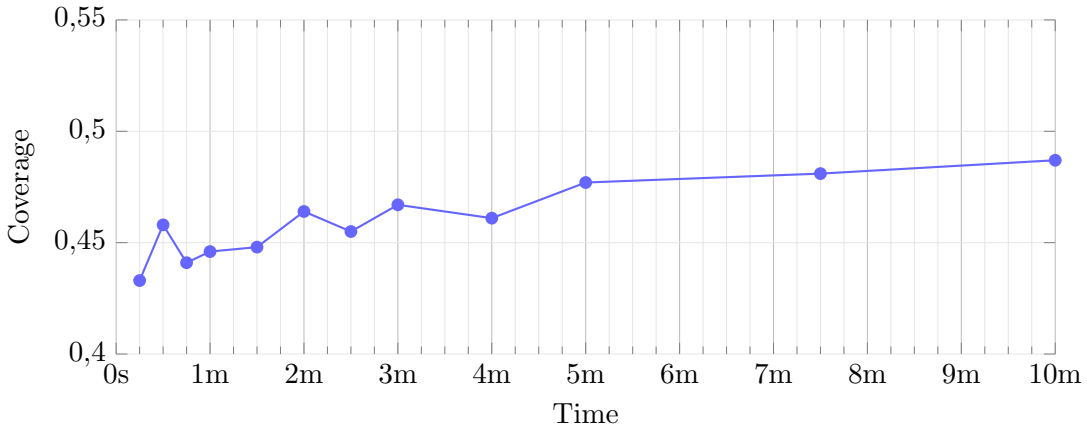

Fig. 4.8: Cobertura White Box de `ratus` en diferentes tiempos

Como *ratus* es una aplicación bastante compleja en comparación con las anteriores, los resultados de cobertura menores tienen sentido.

5. CONCLUSIONES Y TRABAJOS FUTUROS

La integración de Go en EvoMaster cumple muy bien su objetivo, logrando obtener una buena cobertura del código a partir de tests *End to End* al mismo tiempo de mantenerse con una interfaz bastante simple y rápida de implementar.

Los resultados obtenidos en las evaluaciones son acordes a los resultados de los demás lenguajes. Al mismo tiempo se obtuvieron buenos resultados en los casos de estudio reales, demostrando su correcto funcionamiento en los mismos.

A la hora de generar los tests en los lenguajes C# y JavaScript, se descubrió que ya no se mantienen más las mismas. En próximos trabajos sería bueno separar los *Drivers* del *Core* para que sea más fácil su extensibilidad, y también para que esté desacoplado del *Core*.

Como se pudo notar en los casos de estudio reales, la información de la base de datos y como es utilizada juega un rol muy importante para generación de buenos tests de integración. En próximos trabajos se podría realizar la integración de la instrumentación de base de datos.

Algunas aplicaciones modifican bastante su ejecución dependiendo de diferentes configuraciones al iniciar la misma, un ejemplo de esto puede ser utilizar una base de datos en memoria o una base de datos relacional *(como sucede en `hyperonym/ratus4.2.2`)*. En el futuro, sería posible desarrollar una solución que permita modificar estos parámetros de configuración en tiempo de ejecución, cuando el *Core* llama al endpoint `PUT /runSUT` del controlador.

Otra parte complicada de testear son los endpoints que tienen efectos secundarios, como llamadas asincrónicas, ya que la actualización de los objetivos podría ocurrir en cualquier momento. Durante la selección de las aplicaciones reales para evaluar, algunos candidatos fueron descartados debido a este problema.

Finalmente, un último trabajo a futuro puede ser la extensión de la instrumentación a más tipos de datos. Algunos ejemplos incluyen la instrumentación de la biblioteca `time`[24] o la definición de la distancia en los *structs*, por ejemplo, mediante la suma de la distancia de sus atributos.

Bibliografía

[1] EvoMaster. www.evomaster.org.

[2] EvoMaster: Python. https://github.com/axelmaddonni/EvoMaster.

[3] EMB. https://github.com/EMResearch/EMB.

[4] A. Panichella, et al. Many Independent Objective (MIO) Algorithm for Test Generation, 2019. https://arxiv.org/pdf/1901.01541.

[5] Open API. https://swagger.io/resources/open-api/.

[6] Representational State Transfer (REST). https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[7] Biblioteca de Java "ObjectWeb ASM". https://asm.ow2.io/

[8] Documentación de Java "ClassFileTransformer". https://docs.oracle.com/cd/E17802_01/j2se/j2se/1.5.0/jcp/beta1/apidiffs/java/lang/instrument/ClassFileTransformer.html

[9] Biblioteca de C# "Cecil". https://github.com/jbevain/cecil

[10] ECMA CIL. https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf

[11] Biblioteca de JavaScript "babel". https://github.com/babel/babel

[12] Biblioteca de Python "ast". https://docs.python.org/3/library/ast.html

[13] PEP 302 – New Import Hooks. https://peps.python.org/pep-0302/#id21

[14] Go. https://go.dev/

[15] Biblioteca de Go "go/ast". https://pkg.go.dev/go/ast

[16] Biblioteca de Go "dave/dst". https://github.com/dave/dst

[17] Biblioteca de Go "uber-go/mock". https://github.com/uber-go/mock

[18] Biblioteca de Go "golang/protoc-gen-go". https://pkg.go.dev/google.golang.org/protobuf/cmd/protoc-gen-go

[19] Biblioteca de Go "net/http". https://pkg.go.dev/net/http

[20] Biblioteca de Go "stretchr/testify". https://github.com/stretchr/testify

[21] Biblioteca de Go "testing". https://pkg.go.dev/testing

[22] Biblioteca de Go "reflect". https://pkg.go.dev/reflect

[23] Biblioteca de Go “strings”. https://pkg.go.dev/strings

[24] Biblioteca de Go “time”. https://pkg.go.dev/time

[25] Biblioteca de Go “alibaba/fastjson”. https://github.com/alibaba/fastjson

[26] Biblioteca de Go “hyperonym/ratus”. https://github.com/hyperonym/ratus

[27] Biblioteca de Go “CocaineCong/TodoList”. https://github.com/CocaineCong/TodoList

[28] Directivas Go. https://pkg.go.dev/cmd/compile#hdr-Compiler_Directives

[29] Generación de código en Go. https://go.dev/blog/generate

[30] Documentación herramienta go cmd y sus opciones. https://pkg.go.dev/cmd/go#hdr-Compile_packages_and_dependencies

[31] Abstract Syntax Tree (AST). https://www.sciencedirect.com/topics/computer-science/abstract-syntax-tree

[32] Protocol Buffers. https://protobuf.dev/

[33] Martin, James. Managing the Data-base Environment, 1983. p. 381.

[34] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. 2010 Third International Conference on Software Testing, Verification and Validation, 2010. p. 205–214.