Departamento de Sistemas Informáticos y de Computación

Universitat Politècnica de València

Mejorar la selección de acciones en TESTAR con técnicas de Inteligencia Artificial

TRABAJO FINAL DE MASTER (TFM)

Master MUITSS

Autor: Borja Davó Gelardo

Tutor: Tanja Ernestina Jozefina Vos

Curso 2020-2021

Resumen

TESTAR es una herramienta de testing de caja negra que realiza secuencias automatizadas de pruebas a otras aplicaciones, para así encontrar fallos o incoherencias en su comportamiento. La labor de este trabajo es modificar la forma en la que esta herramienta toma decisiones, con el objetivo de reducir la aleatoriedad y conseguir una ejecución más inteligente.

Palabras clave: Pruebas automatizadas, Interfaz Gráfica de Usuario, Inteligencia Artificial, Aprendizaje por Refuerzo, TESTAR, pruebas de caja negra.

Resum

TESTAR és una ferramenta de testing de caixa negra que realitza seqüències automatitzades de proves a altres aplicacions, per a trobar errors o incoherències en el seu comportament. La tasca d'este treball és modificar la forma en què esta ferramenta pren decisions, amb l'objectiu de reduir l'aleatorietat i aconseguir una execució més intel·ligent.

Paraules clau: Proves automatitzades, Interfície Gràfica d'Usuari, Intel·ligència Artificial, Aprenentatge per Reforç, TESTAR, proves de caixa negra.

Abstract

TESTAR is a black box testing tool that performs automated sequences of tests to other applications, in order to find bugs in their behavior. The task of this work is to modify the way in which this tool makes decisions, with the aim of reducing randomness and achieving a more intelligent execution.

Keywords: Automated testing, Graphical User Interface, Artificial Intelligence, Reinforcement Learning, TESTAR, Black box testing

Lista de imágenes

1.1 El ciclo de ejecución de TESTAR 11
1.2 Diagrama de casos de uso de TESTAR 12

3.1 El estado de una GUI. 28
3.2 El estado de una GUI como un árbol de widgets. 29
3.3 Clases etiquetables `State`, `Widget` y `Action` 30
3.4 Capas de los diferentes protocolos de TESTAR 32
3.5 Integración de Reinforcement Learning en TESTAR 33
3.6 Secuencia de RL en TESTAR 35
3.7 Integración de Reinforcement Learning y funciones de recompensa en TESTAR 36

5.1 Proceso de un experimento 47
5.2 El proceso de recolección de datos y extracción de conocimiento. 48

6.1 Tabla de Excel de la cobertura de instrucciones en Rachota con selección aleatoria y agrupación de widgets, al cabo de 200 acciones 57
6.2 Cobertura de instrucciones en Rachota (200 acciones) 58
6.3 Cobertura de instrucciones en Spaghetti (300 acciones) 59
6.4 Cobertura de instrucciones en SwingSet2 (300 acciones) 60
6.5 Cobertura de ramas en Rachota (200 acciones) 61
6.6 Cobertura de ramas en Spaghetti (300 acciones) 61
6.7 Cobertura de ramas en SwingSet2 (300 acciones) 62

B.1 Progreso de la cobertura de instrucciones en Rachota usando selección aleatoria 77
B.2 Progreso de la cobertura de instrucciones en Spaghetti usando selección aleatoria 78
B.3 Progreso de la cobertura de instrucciones en SwingSet2 usando selección aleatoria 78
B.4 Progreso de la cobertura de instrucciones en Rachota usando agrupación de widgets 79
B.5 Progreso de la cobertura de instrucciones en Spaghetti usando agrupación de widgets 79
B.6 Progreso de la cobertura de instrucciones en SwingSet2 usando agrupación de widgets 80
B.7 Progreso de la cobertura de instrucciones en Rachota usando diferencia de widgets 80
B.8 Progreso de la cobertura de instrucciones en Spaghetti usando diferencia de widgets 81
B.9 Progreso de la cobertura de instrucciones en SwingSet2 usando diferencia de widgets 81
B.10 Progreso de la cobertura de instrucciones en Rachota usando diferencia de imágenes 82

B.11 Progreso de la cobertura de instrucciones en Spaghetti usando diferencia de imágenes . . . 82
B.12 Progreso de la cobertura de instrucciones en SwingSet2 usando diferencia de imágenes . . . 83
B.13 Progreso de la cobertura de ramas en Rachota usando selección aleatoria . 83
B.14 Progreso de la cobertura de ramas en Spaghetti usando selección aleatoria 84
B.15 Progreso de la cobertura de ramas en SwingSet2 usando selección aleatoria 84
B.16 Progreso de la cobertura de ramas en Rachota usando agrupación de widgets 85
B.17 Progreso de la cobertura de ramas en Spaghetti usando agrupación de widgets . . . 85
B.18 Progreso de la cobertura de ramas en SwingSet2 usando agrupación de widgets . . . 86
B.19 Progreso de la cobertura de ramas en Rachota usando diferencia de widgets 86
B.20 Progreso de la cobertura de ramas en Spaghetti usando diferencia de widgets 87
B.21 Progreso de la cobertura de ramas en SwingSet2 usando diferencia de widgets . . . 87
B.22 Progreso de la cobertura de ramas en Rachota usando diferencia de imágenes 88
B.23 Progreso de la cobertura de ramas en Spaghetti usando diferencia de imágenes . . . 88
B.24 Progreso de la cobertura de ramas en SwingSet2 usando diferencia de imágenes . . . 89

Lista de tablas

3.1 Ejemplos de widgets de los que puede estar compuesto un GUI . . . 28

A.1 *Test settings* de Rachota . . . 72
A.2 *Test settings* de Spaghetti . . . 73
A.3 *Test settings* de SwingSet2 . . . 75

Lista de algoritmos

1 Funcionamiento de la técnica QLearning 13
2 Algoritmo de RL de AutoBlackTest 19
3 Algoritmo de RL aplicado a apps Android 21
4 Algoritmo de RL aplicado a TESTAR 22
5 Algoritmo No-RL aplicado al testeo de apps Android mediante agrupación 24
6 Algoritmo del enfoque de agrupación de widgets 40
7 Función $learn(s, s', a, reward, \gamma, \alpha)$ en el enfoque de agrupación de widgets 40
8 Algoritmo del enfoque de diferencia de widgets 41
9 Función $calculateReward(s, a, s')$ en el enfoque de diferencia de widgets . 42
10 Algoritmo del enfoque de diferencia visual 44
11 Función $calculateReward(s, a, s')$ en el enfoque de diferencia visual 45

Contenidos

Lista de imágenes 1
Lista de tablas 2
Contenidos 5
1 Introducción 7
1.1 Motivación . . . 8
1.2 Conceptos previos . . . 9
1.2.1 Testing de software . . . 9
1.2.2 TESTAR . . . 10
1.2.3 Reinforcement Learning . . . 12
1.2.4 Q-learning . . . 13
1.3 Objetivos . . . 14
1.4 Impacto Esperado . . . 15
1.5 Colaboraciones . . . 15
1.6 Metodología . . . 16
1.7 Análisis de riesgos . . . 16
1.8 Estructura de este documento . . . 17
2 Estado del arte 19
2.1 AutoBlackTest [24] . . . 19
2.2 Adamo et al. [13] . . . 20
2.3 Esparcia et al. [22] . . . 22
2.4 Cao et al. [20] . . . 23
2.5 Observaciones del estado del arte . . . 25
3 Análisis del problema 27
3.1 Análisis del funcionamiento de TESTAR . . . 27
3.1.1 Estados del GUI en TESTAR . . . 27
3.1.2 Representación de estados y acciones . . . 30
3.1.3 Actividad de TESTAR en tiempo de ejecución . . . 31
3.2 Análisis del marco para Reinforcement Learning de TESTAR . . . 32
3.3 Análisis del entorno de desarrollo . . . 35
3.4 Identificación y análisis de soluciones posibles . . . 36
4 Soluciones propuestas 39
4.1 Enfoque de selección aleatoria . . . 39
4.2 Enfoque de agrupación de widgets . . . 39
4.3 Enfoque de diferencia de widgets . . . 41
4.4 Enfoque de diferencia visual . . . 44
5 Validación 47
5.1 Preguntas de investigación . . . 48
5.2 Contexto . . . 49
5.3 Hipótesis . . . 49
5.4 SUTs . . . 49
5.5 Variables . . . 50
5.5.1 Variables independientes . . . 50

5.5.2 Variables dependientes 50
5.6 Diseño 51
5.7 Instrumentación 51
5.7.1 Las máquinas virtuales y los workers 51
5.7.2 Los *test settings* de TESTAR y los SUTs 52
5.7.3 JaCoCo 53

6 Resultados **55**
6.1 Cobertura temporal 55
6.2 Características del test U 56
6.3 Aplicación del test U 57

7 Conclusiones **63**
7.1 Relación del trabajo desarrollado con los estudios cursados 63
7.2 Trabajos futuros 64

Bibliografía **65**

A *Test settings* de los SUTs utilizados **71**

B Progreso de la cobertura de ramas e instrucciones **77**
B.1 Cobertura de instrucciones con selección aleatoria 77
B.2 Cobertura de instrucciones con agrupación de widgets 79
B.3 Cobertura de instrucciones con diferencia de widgets 80
B.4 Cobertura de instrucciones con diferencia de imágenes 82
B.5 Cobertura de ramas con selección aleatoria 83
B.6 Cobertura de ramas con agrupación de widgets 85
B.7 Cobertura de ramas con diferencia de widgets 86
B.8 Cobertura de ramas con diferencia de imágenes 88

CAPÍTULO 1
Introducción

TESTAR[10] es una herramienta open source[9] que implementa una aproximación sin scripts para la generación de pruebas completamente automatizadas para aplicaciones web y de escritorio de Windows. La herramienta está desarrollada por el grupo STaQ (Software Testing and Quality) del centro de investigación PROS de la UPV.

TESTAR está basado en agentes que implementan diversos mecanismos de selección de acciones y oráculos de pruebas. Los principios subyacentes son muy simples: generar secuencias de prueba de pares (estado, acción) iniciando el sistema bajo prueba (SUT) en su estado inicial y seleccionando continuamente una acción para llevar el SUT a otro estado. La **selección de acciones** representa el desafío fundamental de los sistemas inteligentes: qué hacer a continuación[1]. La complejidad aparece al tratar de optimizar la selección de acciones para encontrar fallos y reconocer un estado defectuoso cuando se encuentra, usando oráculos. Los estados defectuosos no se limitan a errores en la funcionalidad, también se pueden detectar infracciones de otras características de calidad, como accesibilidad o seguridad, mediante la inspección del estado.

TESTAR cambia totalmente el paradigma de las pruebas de GUI: desde el desarrollo de scripts hasta el desarrollo de agentes inteligentes habilitados para Inteligencia Artificial (IA).

El desarrollo de TESTAR empezó durante el proyecto FITTEST[3, 34], liderado por la UPV, desde el 2010 hasta el 2013. Posteriormente, el desarrollo continuó mientras se probaba la herramienta en varias empresas en proyectos ERASMUS+, como SHIP [32], y varios más, financiados a nivel nacional por los gobiernos de España y Valencia. En 2017, el desarrollo continuó en el contexto de los proyectos TESTOMAT[11] ITEA3, DECODER[2] de la UE H2020, iv4XR[4] de la UE H2020 y IVVES[5] ITEA3.

Durante los últimos 4 años, TESTAR ha sido desarrollado y ampliado significativamente, y muchas partes internas han sido cambiadas. El primer artículo general se publicó [33] en 2015, y en 2021 se publicará el más reciente en STVR [31].

Con este trabajo TFM se logra **extender TESTAR con diferentes mecanismos de selección de acciones basados en la Inteligencia Artificial**, para que la selección de acciones se lleve a cabo de forma más inteligente.

En este primer capítulo se pretende contextualizar el área de trabajo sobre la cual se desarrolla el proyecto. Para ello, se explicará cómo se ha llegado a escoger el tema, qué se busca ofrecer, de qué forma se ha organizado el desarrollo y qué otros agentes influyen

en él.

Adicionalmente, se incluye al final de este documento un **glosario de términos técnicos**, que pretenden ser de ayuda al lector para esclarecer conceptos que puedan no ser de ámbito general.

1.1 Motivación

Para finalizar el Máster es necesaria la realización de un trabajo final, tutorizado por personal docente de la universidad. La forma en la que se ha elegido este proyecto es explicada a continuación.

En primer lugar, se tuvieron en cuenta algunas **propuestas** de trabajos. Estas fueron anunciadas por profesores, quienes expusieron información básica sobre la temática de sus ideas de proyecto. De entre todas ellas, se **descartaron** aquellas que eran de menor interés, por razones de ámbito, tecnología, temática, etc. Después se **destacaron** aquellas que llamaron más la atención, pasando a ser opciones candidatas a ser escogidas.

Entre las propuestas más relevantes se encontraban varias que hacían referencia a la herramienta **TESTAR**, desarrollada por el grupo STaQ (Software Testing and Quality) del centro de investigación PROS de la UPV. El recorrido de dicha herramienta, así como su funcionamiento, generaron interés en ella.

En ese momento existían cuatro sugerencias para la realización del Trabajo de Fin de Máster en relación con TESTAR:

- Conseguir la detección automática de **diferencias entre versiones** de software
- Desarrollar un plugin TESTAR para el testing de **aplicaciones móviles** en Android o iOS
- Usar Inteligencia Artificial y Reconocimiento de Imágenes para **detectar acciones** a ejecutar en una interfaz gráfica de usuario
- **Mejorar la selección de acciones** en TESTAR con técnicas de Inteligencia Artificial

De entre todas ellas, se prefirió elegir la última. La razón es que el ámbito de la IA es un tema que había sido de **interés** desde hacía mucho tiempo, y esta sería una buena oportunidad para dar los primeros pasos. Este trabajo pretende ser un **empujón adicional** al continuo perfeccionamiento al que está sometida la herramienta anteriormente mencionada.

Otra de las razones por las que se prefirió optar por trabajar en TESTAR es el hecho de que este software tiene una **utilidad clara** y promete ser mejor con cada nueva actualización que lleve a cabo en él. Su **proyección de futuro** es estable, con una larga vida en el ámbito del testing.

Además, las **tecnologías** utilizadas entraban dentro del conocimiento actual, por lo que la desenvoltura y la confianza en el código no eran escasas. El desarrollo se llevó a cabo utilizando Java, usados en diversas ocasiones con anterioridad.

1.2 Conceptos previos

1.2.1. Testing de software

En el desarrollo de software se busca que la calidad del producto sea lo más alta posible [23]. Es por ello que se pretende cubrir en la medida de lo posible los requisitos correspondientes. El **testing** es la fase del ciclo de vida del desarrollo de software que permite ofrecer al equipo de programación información relevante sobre criterios como el funcionamiento de los programas [26]. Se basa en actividades que evalúan características del software, que pueden ser llevadas a cabo en distintos momentos del desarrollo.

Por un lado, pueden ser verificados **requisitos no funcionales**, relacionados estrictamente con propiedades ajenas a la funcionalidad del programa, como la usabilidad, el rendimiento, la escalabilidad o la disponibilidad.Por otro lado, los análisis llevados a cabo en el testing pueden poner el foco en **requisitos funcionales**, que permiten comprobar si las funcionalidades del software son completas y correctas. Para probar este tipo de requisitos es común considerar los distintos **niveles** de abstracción del programa. De este modo, en primer lugar se realizan pruebas en el nivel más bajo, y posteriormente se van probando los niveles superiores hasta llegar al más alto. El orden común que se sigue es el siguiente:

- **Pruebas unitarias:** Se prueban distintas unidades de código, que corresponden a elementos como clases o procedimientos.
- **Pruebas de integración:** Verifican que los elementos unitarios anteriormente probados funcionan correctamente de forma conjunta.
- **Pruebas de sistema:** Son un conjunto de pruebas enfocadas en analizar el comportamiento global del software y su interacción con sistemas externos.
- **Pruebas de humo:** Se busca revisar de forma rápida que no existan fallos en el programa.
- **Pruebas alpha:** Examinan prototipos que normalmente son de bajo coste, para decidir en fase de desarrollo cuál es el mejor camino a tomar.
- **Pruebas beta:** Estas pruebas son realizadas por usuarios finales, y permiten encontrar errores que no se habían detectado en el desarrollo.
- **Pruebas de aceptación:** Se llevan a cabo antes de la salida a producción del software. El cliente prueba el programa compruebas si se cumple lo que espera de él.

Es posible realizar algunas de estas pruebas a nivel de **GUI**, es decir, considerando las características de la interfaz gráfica de usuario. Un ejemplo de ello son las pruebas de sistema, con las que a través de la interfaz se puede validar el comportamiento del sistema. También podemos incluir las pruebas de aceptación, porque la persona que prueba el funcionamiento del software lo hace interactuando directamente con la interfaz.

Continuando a nivel de GUI, podemos diferenciar dos tipos de testing, en función de la forma en la que se empleen scripts en el proceso de validación de software. Por un lado, el **testing con scripts** permite tener en cuenta un conjunto de instrucciones anotadas en

un documento, para interpretarlas y devolver la información correspondiente. Esta es una buena forma de probar el sistema a fondo, cubriéndose así el mayor número de casos de prueba. En contraste, el **scriptless testing** evita el uso de estos scripts, y en su lugar analiza el programa directamente desde la GUI. En este caso, las características y funcionalidades pueden ser verificadas de una forma más rápida y sencilla.

Una de las principales ventajas del scriptless testing es la reducción de los **costes**. En el caso del testing con scripts, los documentos de test deben ser actualizados a medida que el código del programa va cambiando. Esto implica un trabajo de **mantenimiento**, algo que no es necesario en el caso del scriptless testing. Además, el hecho de no usar scripts permite analizar la interfaz gráfica de usuario de forma rápida, dinámica y automática.

Este TFM gira entorno **TESTAR**, un software de scriptless testing que permite evaluar elementos de la GUI de distintas aplicaciones. Realiza pruebas de forma automática, en base a una selección aleatoria y unos oráculos implícitos, y recoge la información obtenida. El proyecto pretende ampliar esta herramienta para incluir características que permitan dotarle de una mayor inteligencia.

1.2.2. TESTAR

En la actualidad, la mayor parte del testing se lleva a cabo teniendo en cuenta los requisitos del software a probar, y scripts en los que estructurar las instrucciones de las pruebas a realizar. Sin embargo, TESTAR plantea otro punto de vista. Esta es una herramienta de testing que puede ser clasificada como **scriptless**, es decir, que no usa scripts para realizar las pruebas. Como se ha visto anteriormente, esto evita costes de mantenimiento. Además, permite **empezar los tests sin necesidad de requisitos**, dado que se basa en testing aleatorio y oráculos implícitos.

En el **testing aleatorio** se realizan pruebas de robustez de tipo **out of the box**. Esto quiere decir que se seleccionan acciones de forma aleatoria en la aplicación a probar. De este modo se sobrepasan los límites del testing de caja negra o blanca, fundamentados en el código o los requisitos, y se trabaja más allá de la caja.

Mediante los **oráculos implícitos** es posible encontrar fallos relacionados con requisitos no funcionales de la aplicación. De este modo, se puede detectar cuándo el programa se bloquea, se cuelga o aparecen ventanas con títulos de error.

En relación a cómo trabaja TESTAR, podemos organizar su **funcionamiento** con los siguientes pasos (reflejado en la Figura 1.1):

1. **Iniciamos el Sistema Bajo Test (SUT)**.
2. **Se obtiene el estado actual** del software analizado. Dicho estado está organizado en un árbol en el que aparecen los widgets disponibles. El árbol se genera automáticamente a partir del estado, por lo que no es necesario ningún mantenimiento.
3. **Se comprueba qué acciones están disponibles** en ese punto de la ejecución del programa analizado.
4. **Se selecciona una de dichas acciones** de forma aleatoria.
5. **La acción seleccionada es ejecutada**.

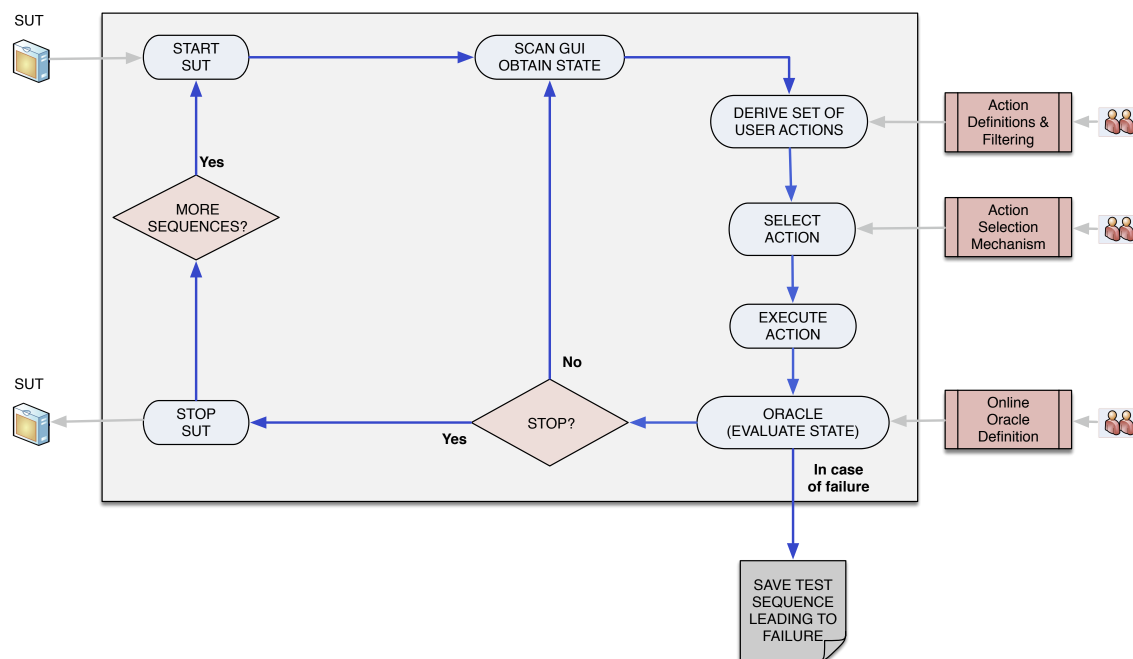

Imagen 1.1: El ciclo de ejecución de TESTAR

6. **Se verifica que el estado es correcto**. Si no lo es, se almacena la secuencia de test que ha llevado a error y que podrá ser ejecutada posteriormente.
7. **Si el estado no es correcto, se detiene la ejecución del test**.
8. **Si el estado es correcto, se vuelve al paso 2**.

TESTAR dispone de cinco modos de ejecución, que ofrecen distintas funcionalidades en relación con la realización y gestión de tests. Su representación en un **diagrama UML de casos de uso** es en la Figura 1.2.

- **SPY**: Permite inspeccionar los widgets existentes de la GUI.
- **GENERATE**: Se realizan tests automatizados siguiendo el ciclo visto anteriormente.
- **REPLAY**: Permite seleccionar secuencias de test realizadas anteriormente, para reproducirlas.
- **VIEW**: Para visualizar secuencias de test realizadas.
- **RECORD**: Permite tomar el control de la ejecución y llevar a cabo acciones manuales durante un test.

En comparación con el testing con scripts, en TESTAR se dedica aproximadamente el mismo tiempo a **especificar casos de test**. Tampoco ha diferencias significativas a la hora de **especificar oráculos**. Sin embargo, la etapa de **desarrollar scripts** no está presente al usar TESTAR. Por esta razón, al usar esta herramienta, la fase de **mantenimiento** es significativamente menos costosa. Además, en este caso, la **ejecución de test automatizada** es mucho más notable.

TESTAR ha sido **desplegado exitosamente en diversas empresas**, como ClaveiCon [17], Softeam [18], Indenova [25], Cap Gemini/Prorail [21], Ponsse [14], ProDevelop [28]. El

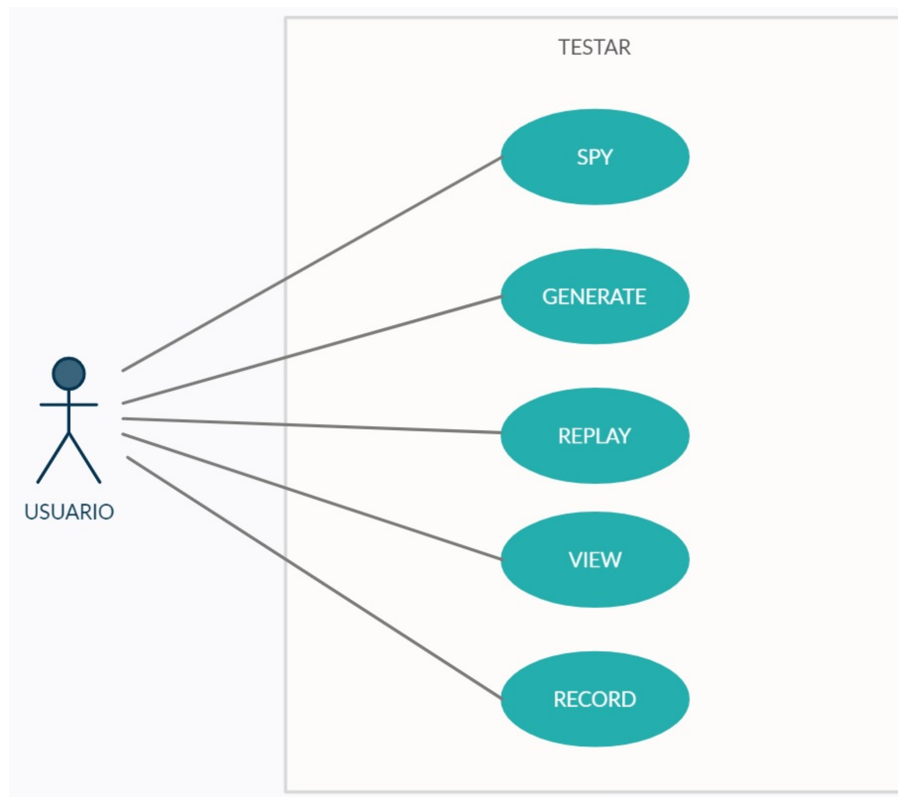

Imagen 1.2: Diagrama de casos de uso de TESTAR

rendimiento de la herramienta ha sido muy bueno, teniendo en cuenta que realiza testing de tipo out of the box. Además, con la configuración de requisitos específicos y con la adición de nuevos oráculos, TESTAR permite adaptarse mejor al sistema que está evaluando. De este modo, se añaden nuevos testwares de forma incremental.

1.2.3. Reinforcement Learning

En el área de la Inteligencia Artificial, la disciplina del **Aprendizaje Automático** pretende lograr un aprendizaje por parte de las máquinas. En concreto, se busca que un elemento, llamado comúnmente agente, mejore su toma de decisiones mediante la experiencia, con el fin de lograr un objetivo [30]. El **Reinforcement Learning** (RL), conocido como Aprendizaje por Refuerzo, se basa en los principios del Aprendizaje Automático, y pone su foco en elementos como la toma de acciones, la transición entre estados y la existencia de recompensas.

Entre los componentes siempre presentes en en un escenario de RL, consideramos en primer lugar la figura del **agente**, capaz de percibir su entorno y llevar a cabo entrenamientos para lograr un objetivo concreto. En segundo lugar, diferenciamos el **entorno**, cuya representación abarca las posibles alternativas que el agente puede tomar. Proporciona la información necesaria en relación al contexto, para que pueda ser posible el aprendizaje. Además, los **estados** son el conjunto de situaciones en las que se puede encontrar el agente. En un estado determinado existen una serie de **acciones** disponibles, que de ser tomadas podrían llevar al agente a un estado distinto.

RL tiene un **procedimiento** determinado. En cada instante de tiempo, ocurre lo siguiente:

1. El agente observa su entorno y genera una **representación del su estado**.
2. Se comprueban **qué acciones están disponibles en el estado actual**.
3. **Se selecciona una acción** de entre las disponibles.
4. **La acción seleccionada se ejecuta**.
5. El entorno devuelve al agente una **recompensa** numérica.

Cuanto mayor sea el valor, más positiva será la recompensa. De este modo, a medida que se vayan llevando a cabo secuencias de acciones, el agente tomará mejores decisiones, procurando así obtener el mayor número de recompensas positivas. Las recompensas se obtienen al ejecutar una acción en un estado. Como ejemplo básico, puede entenderse el valor 1 como una recompensa positiva; -1, como una negativa; y 0, como una recompensa neutra, fruto de una decisión que ni acerca ni aleja al agente del objetivo final. El propósito es que el agente encuentre una secuencia de acciones que le permitan llegar a su destino con la **recompensa acumulada** más alta posible.

1.2.4. Q-learning

Existen varios tipos de RL, como la Evaluación Directa, la Diferencia Temporal o el Deep Reinforcement Learning. Sin embargo, la técnica llamada **Q-learning** destaca por encontrar la forma de maximizar la recompensa total que recibe el agente, teniendo en cuenta los estados por los que va pasando. Fue inventado por Watkins [35].

Este tipo de RL utiliza una tabla numérica llamada **Q-table**, que proporciona información acerca de cuál es la mejor acción a tomar en cada estado en que se encuentre el agente. Al inicio, esta tabla contiene valores por defecto, los cuales se actualizan a medida que se vayan llevando a cabo entrenamientos. De este modo, para una acción en un estado se calculan las máximas recompensas futuras que se espera recibir.

El algoritmo general de Q-learning (1) se puede ver a continuación.

Algoritmo 1 Funcionamiento de la técnica QLearning

Require: γ ▷ factor de descuento
Require: α ▷ tasa de aprendizaje
1: $initializeQValues()$
2: **for each** $sequence$ **do**
3: $s \leftarrow getStartingState()$
4: $s' \leftarrow s$
5: **repeat**
6: $availableActions \leftarrow getAvailableActions(s)$
7: $a \leftarrow selectAction(availableActions)$ ▷ **Seleccionar** una acción
8: $performAction(a)$
9: $s' \leftarrow getReachedState()$
10: $reward \leftarrow calculateReward(s, a, s')$ ▷ **Recompensa** de la acción
11: $learn(s, s', a, reward, \gamma, \alpha)$ ▷ **Aprender** de la experiencia
12: $s \leftarrow s'$
13: **until** s′ is the last state of the sequence
14: **end for**

El algoritmo depende de dos inputs:

- γ: Es el factor de descuento, y representa el grado decrecimiento de las recompensas de las acciones tras ser ejecutadas.
- α: Es la tasa de aprendizaje, y determina alrededor de qué cantidad se actualizan los valores. Normalmente se elige una cifra pequeña o se reduce en cada iteración.

En primer lugar, se proporcionan estos dos parámetros al algoritmo. A continuación, se inicializan los valores de Q a una cifra predeterminada (línea 1).

Conceptualmente, s representa el estado anterior del sistema, y s', el estado actual. Para cada secuencia, se inicializan s y s' al primer estado (líneas 3 y 4). Después, se llevará a cabo una secuencia de acciones hasta que s' sea el último estado de la secuencia (linea 13).

En cada iteración del bucle **Repeat** de la linea 5, se elige la acción que será ejecutada (línea 7). Esta decisión viene dada por la **política** que se siga y que estará implementada en *selectAction*.

Luego se ejecuta la acción seleccionada (línea 8) y el valor de s' se actualiza, pasando a ser el estado alcanzado tras ejecutar la acción (línea 9).

Además, se calcula el valor de recompensa que se ha obtenido (línea 10), usando para ello una **función de recompensa** que estará implementada en *calculateReward*. Esta depende de transiciones de estado previas, y permite que las recompensas disminuyan su valor a medida que el agente ejecute acciones.

En la línea 11 tiene lugar el aprendizaje, guiado por el algoritmo específico usado. Este proceso de aprendizaje estará basado en la **Q-table** que se actualizará a partir del estado anterior, el actual, la acción ejecutada, la recompensa obtenida, el factor de descuento (γ) y el factor de aprendizaje (α). De este modo, los valores de Q correspondientes son actualizados para dirigir al sistema hacia los objetivos establecidos. En Watkins [35] la actualización del Q en $learn(s, s', a, reward, \gamma, \alpha)$ está definida como:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(reward + \gamma\ max_{a' \in getAvailableActions(s')} Q(s',a') - Q(s,a)) \quad (1.1)$$

Finalmente, el estado anterior toma el valor del estado actual (línea 12), y se vuelve a realizar el mismo proceso desde la línea 5, en caso de que no se cumpla la condición de parada.

1.3 Objetivos

TESTAR siempre ha realizado la selección de acciones principalmente de forma aleatoria. Existen solo un estudio que pretenden acercar esta herramienta a **técnicas de Inteligencia Artificial**, en concreto al **Reinforcement Learning** [22]. Con este trabajo se pretenden crear más aproximaciones RL que incorporen conceptos de estos ámbitos de forma sólida. De este modo, se conseguiría crear unos criterios sobre los que se fundamentaría la **selección de acciones**, para que no sea totalmente dependiente del azar.

Se busca mejorar la herramienta y hacer que su crecimiento continuo se vea mucho más enriquecido. En esta mejora constante de funcionalidades en la herramienta TESTAR,

este trabajo procura ser un empujón más hacia adelante y contribuir a su óptimo funcionamiento.

1.4 Impacto Esperado

Para un usuario de TESTAR, las nuevas actualizaciones que se llevan a cabo con este trabajo suponen un incremento en la **eficacia** del testing con dicha herramienta. Al iniciar la ejecución de las pruebas, el sujeto debe poder ver que no se realizan redundancias en la selección de acciones. Por esta razón, asumirá con mayor ímpetu que TESTAR trata de abarcar el mayor espacio funcional posible de las aplicaciones que evalúa, y así llevar a cabo su cometido de una forma más acertada.

1.5 Colaboraciones

Como hemos mencionado anteriormente, TESTAR empezó a desarrollarse en el proyecto FITTEST (Future Internet Testing) de la Unión Europea, que estuvo activo desde el año 2010 hasta el 2013. Después, se continuó su desarrollo por la Universitat Politècnica de València (UPV), Utrecht University y la Open University of The Netherlands, en el contexto de distintas iniciativas de financiación nacionales y europeas.

La herramienta se fue probando en varias empresas y proyectos ERASMUS+, como SHIP. En 2017, el desarrollo continuó dando lugar a los siguientes proyectos:

- **TESTOMAT**: El objetivo de este proyecto es ayudar a los equipos de desarrollo de software aumentando la velocidad de desarrollo sin sacrificar la calidad, lograr el equilibrio adecuado. Ofrece modelos de mejora en términos de automatización de pruebas, prestando atención a las áreas más delicadas.
- **DECODER**: Esta plataforma crea un entorno de desarrollo integrado (IDE) que combina información de diferentes fuentes a través de modelos formales y semiformales. El objetivo es ofrecer mecanismos inteligentes para acortar la curva de aprendizaje de los programadores cuando se entreguen código mutuamente, y así aumentar su productividad. Se conocerá inmediatamente qué se ha hecho, cómo y con qué herramientas.
- **iv4XR**: Este proyecto tiene como objetivo construir una nueva tecnología de verificación y validación para sistemas XR (Realidad Extendida) basada en técnicas de IA. Los desarrolladores de XR pueden implementar poderosos agentes de pruebas para explorar y probar automáticamente sus entornos.
- **IVVES**: Con el objetivo de abordar los desafíos en la verificación y validación de la IA y los sistemas evolutivos, IVVES desarrolla sistemáticamente enfoques de inteligencia artificial para garantizar robustez y completitud en el ámbito industrial. Esto involucra el aprendizaje automático para el control de sistemas complejos y críticos evolutivos.

Durante la realización de este proyecto, habrá colaboraciones con otras personas que, a través de estos proyectos, están participando en el equipo TESTAR.

1.6 Metodología

Para llevar a cabo este trabajo, en primer lugar es necesario conocer TESTAR de primera mano. Para ello, debe poderse **manejar dicha aplicación con soltura**. En esta tarea es de gran ayuda el conocimiento proporcionado por la tutora del proyecto, así como por colaboradores con experiencia en el uso y desarrollo de la herramienta. Adicionalmente, son de interés recursos como guías de uso y de funcionamiento, y otros documentos relacionados presentes en sitios web oficiales.

Para lograr una comunicación sólida entre los integrantes del equipo del proyecto, es imperativo **evaluar periódicamente** el trabajo realizado, así como los próximos pasos a dar. Todo ello debe estar alineado con unas metas y objetivos comunes, a los cuales se pretende llegar en las etapas finales del trabajo.

Acto seguido, se debe comprender **cómo funciona internamente TESTAR**. Aunque no sea necesario entender la función de cada elemento presente en el código, es imprescindible familiarizarse con aquellas secciones que tengan una clara notoriedad en el desarrollo del proyecto. Este aprendizaje viene guiado por trabajadores con una considerable trayectoria en el desarrollo de la aplicación.

Conociendo el funcionamiento interno de TESTAR y aspectos claves de su código, conviene **analizar propuestas** que podrían ofrecer una solución al problema que se plantea. Tras barajar sus ventajas e inconvenientes, es el momento de poner en práctica los enfoques considerados. Para ello, es necesario realizar las actualizaciones pertinentes en al código del programa, en una rama de desarrollo dedicada al actual proyecto.

A medida que se va **desarrollando el trabajo**, es necesario que los miembros del equipo den el visto bueno a aquellas características que consideren acertadas para la consecución de los objetivos. Del mismo modo, si algunas particularidades no tienen una relevancia justificable o no son del todo correctas, se debe prescindir de ellas y redireccionar el rumbo del proyecto.

Al final se **validará el trabajo** con experimentos controlados.

1.7 Análisis de riesgos

Antes de enfocarse en el desarrollo de las características que atañen a este proyecto, es recomendable analizar distintos riesgos que podrían influir en el proceso. Podrían o no manifestarse en la realidad, por lo que detectarlos y evaluarlos puede permitir conocer mejor los escenarios adversos, así como formular posibles soluciones. Analizaremos tres riesgos principales que podrían presentarse:

- **Riesgo tecnológico**:

  En esta categoría destacamos la posibilidad de que se desconozca cómo llevar a cabo el proyecto con alguna herramienta tecnológica. Esto incluye no haber usado nunca, o haberlo hecho de forma escasa, lenguajes de programación, entornos, frameworks u otras herramientas de trabajo.

  En el caso de que este riesgo ocurra, no se podrá avanzar en el desarrollo. Las funcionalidades que se pretenden añadir a TESTAR deben ser escritas acorde a una

serie de metodologías y lenguajes específicos que, de no conocerse con la suficiente soltura, harían de ello una ardua tarea.

Para mitigar el riesgo, en caso de que se manifieste, es necesario realizar un aprendizaje en aquellas áreas tecnológicas imprescindibles o cuya necesidad sea clara.

- **Riesgo de integración**:

  En este trabajo se pretende modificar código existente, y también añadir más. Un riesgo que debe tenerse en cuenta es la posibilidad de que se produzca algún problema de integración. Las nuevas líneas podrían no seguir los procedimientos estructurales o conceptuales deseados, o interferir negativamente en código anterior.

  De producirse este riesgo, el código final no podría ser ejecutado o implantado como debería. El software no podría realizar su función hasta que el inconveniente fuera resuelto de forma exitosa.

  Con tal de eliminar estas adversidades, es necesario contrastar el modo de trabajo con el que se venía llevando a cabo hasta el momento. Además, debe analizarse cuidadosamente si el nuevo código interfiere de alguna forma inesperada con otros sectores del código. Adicionalmente, es muy recomendable consultar con otras personas que hayan colaborado con el código con anterioridad, o que dispongan de conocimiento suficiente para entender los incidentes.

- **Riesgo sanitario**:

  En los últimos tiempos, la difícil situación sanitaria ha afectado a la sociedad de todo el mundo. En España, se llevaron a cabo una serie de restricciones que limitaron notablemente las interacciones sociales, por lo que el impacto económico, social y psicológico ha sido notable. Esto podría ocasionar dificultades para acceder a una atención de calidad en caso de que el equipo informático de trabajo sufra un problema y deba ser reparado o cambiado.

  El impacto que esto podría producir es hacer imposible la realización del trabajo pertinente. Sin los recursos necesarios, y sin la disponibilidad de otros medios, el proceso se ve detenido.

  Entre las medidas que se pueden tomar para reducir el impacto se encuentra disponer de equipos informáticos fiables o suficientes, y llegar a un acuerdo con las personas responsables de la gestión de los Trabajos de Fin De Máster.

1.8 Estructura de este documento

Este documento está organizado entorno a una serie de capítulos, las cuales sirven de base para enunciar características del proyecto y de aquello relacionado con él.

Primero, se comentará el **estado del arte** (capítulo 2). Para ello, se realizará una crítica al estado del arte para conocer algunos trabajos que comparten contextos o características similares. Del mismo modo, se expondrá cuál es la propuesta que pretende distinguir este trabajo de los demás.

Después, se llevará a cabo el **análisis del problema** (capítulo 3). Esto incluye evaluar aspectos como la seguridad, la eficiencia algorítmica, la ética y los riesgos. Adicionalmente, se identificará y analizarán las soluciones posibles.

A continuación, se identificará la **solución más acertada** (capítulo 4) y se expondrán sus características más relevantes.

Luego, se profundizará en la **validación** de las estrategias planteadas (capítulo 5), ejecutando experimentos controlados y almacenando información de interés.

Seguidamente, midiendo los **resultados** (capítulo 6) con una serie de métricas, con el fin de determinar si mejoran el testeo con TESTAR.

Finalmente, se **concluirá** el documento (capítulo 7) mencionando la relación del trabajo realizado con los estudios que han sido cursado en la universidad. Además, se comentarán trabajos futuros situados en la línea de desarrollo de TESTAR.

Asimismo, se incluye al final de este documento un **glosario** (capítulo 7.2) con algunos **términos técnicos o siglas** que podrían ser desconocidos para el lector, y que permiten ofrecer un mayor conocimiento de los conceptos a los que hacen referencia.

CAPÍTULO 2

Estado del arte

Existen aproximaciones en el ámbito del testing que actualmente pretenden acercar elementos Inteligencia Artificial a su operatividad. Algunas de ellas se mencionan a continuación.

2.1 AutoBlackTest [24]

AutoBlackTest es una técnica de generación automática de tests a nivel de GUI [24]. Para simular las funcionalidades de la aplicación a evaluar, e interactuar de forma no aleatoria, utiliza Reinforcement Learning. En concreto, se basa en el algoritmo de Q-learning 1 visto con anterioridad, para plantear la propuesta que podemos ver en el algoritmo 2.

Algoritmo 2 Algoritmo de RL de AutoBlackTest

Require: γ ▷ factor de descuento: 0,9
Require: α ▷ tasa de aprendizaje: 1
Require: ϵ ▷ valor de probabilidad: 0,8
1: $initializeQValues()$
2: **for each** $sequence$ **do**
3: $s \leftarrow getStartingState()$
4: **repeat**
5: $availableActions \leftarrow getAvailableActions(s)$
6: $selectionWay \leftarrow getSelectionWay()$
7: $selectedAction \leftarrow \begin{cases} selectMax(availableActions) & \text{if } selectionWay \text{ is } Maximum \\ selectRandom_\epsilon(availableActions) & \text{if } selectionWay \text{ is } Random \end{cases}$
8: $a \leftarrow selectedAction$ ▷ **Seleccionar** una acción
9: $performAction(a)$
10: $s' \leftarrow getReachedState()$
11: $reward \leftarrow diff(s, s')$ ▷ **Recompensa** de la acción
12: $Q(s, a^*) \leftarrow reward + \gamma\ max_{a \in A_{s'}} Q(s', a)$ ▷ **Aprender** de la experiencia
13: $s \leftarrow s'$
14: **until** s′ is the last state of the sequence
15: **end for**

En primer lugar, AutoBlackTest recibe tres valores de entrada: el factor de descuento γ, el cual vale 0,9; la tasa de aprendizaje α, que es 1; y, ϵ, que tiene un valor de 0,8. α, al tener el valor mencionado, no tiene efecto en el proceso algorítmico.

La **política** de selección de acciones que se sigue es ϵ-greedy, por lo que se selecciona en cada estado una acción aleatoria, con probabilidad ϵ, o se selecciona la acción con el valor de Q más alto. Estas decisiones se llevan a cabo mediante un planificador interno. En el algoritmo podemos apreciar la obtención de las acciones disponibles (línea 5), así como del modo de selección que se usará (línea 6). Partiendo de esta información, se selecciona una acción, que es asignada a a (líneas 7 y 8).

Tras seleccionar la acción, esta es ejecutada (línea 9), y s' pasa a representar el nuevo estado alcanzado (línea 10).

La **función de recompensa** que usa AutoBlackTest se puede ver en la línea 11, y está definida para dar más recompensa a las acciones que provocan más cambios en los elementos que componen la interfaz gráfica. Consiste en una heurística en la que se valoran positivamente las acciones que producen efectos relevantes, y penaliza a aquellas que no lo hacen. Está definida de la siguiente forma:

$$reward(s, a, s') = diff(s, s')$$

- *diff(s,s')* devuelve el grado de cambio entre los estados s y s'.

  Al ejecutar una acción s en un estado, obtenemos el estado siguiente s'. A mayor diferencia entre el árbol de widgets de cada estado, mayor será la recompensa de la acción. Esta idea mejora notablemente el testing, al premiar de una mejor forma a aquellas acciones que producen estados menos similares entre sí y, en definitiva, que favorecen una mayor cobertura del SUT testeado.

Una vez que se ha calculado el valor de recompensa, este es usado para actualizar el valor de Q de la acción que ha sido ejecutada. De este modo, como podemos ver en el algoritmo 1, el sistema lleva a cabo un aprendizaje ($learn(s, s', a, reward, \gamma, \alpha)$). Este proceso se muestra en la línea 12, haciendo uso de la **función de Q** que se incluye a continuación:

$$Q(s, a^*) = reward + \gamma \; max_{a \in A_{s'}} Q(s', a)$$

- *reward* es la recompensa que se ha calculado.
- γ es el factor de descuento.
- $max_{a \in A_{s'}} Q(s', a)$ es el valor de Q máximo que existe en el estado s', y que está asociado a la acción a del conjunto de acciones $A_{s'}$.

2.2 Adamo et al. [13]

Las técnicas de RL también han sido aplicadas en otros entornos, como en las GUI de **apps Android**. El estudio realizado por Adamo et al. [13] ofrece una perspectiva basada en el Algoritmo 1 de Q-learning para evaluar la mejora en el testeo en dicho tipo de aplicaciones. La propuesta se puede ver en el algoritmo 3:

Algoritmo 3 Algoritmo de RL aplicado a apps Android

Require: γ ▷ factor de descuento: 1
Require: α ▷ tasa de aprendizaje: 1
1: $initializeQValues()$
2: **for each** *sequence* **do**
3: $s \leftarrow getStartingState()$
4: **repeat**
5: $availableActions \leftarrow getAvailableActions(s)$
6: $a \leftarrow selectMaxAction(availableActions)$ ▷ **Seleccionar** una acción
7: $performAction(a)$
8: $s' \leftarrow getReachedState()$
9: $numOfActions \leftarrow getNumOfActions(s')$
10: $\gamma \leftarrow 0,9\ e^{-0,1\ (numOfActions-1)}$
11: $reward \leftarrow \frac{1}{x_a}$ ▷ **Recompensa** de la acción
12: $Q(s,a) \leftarrow reward + \gamma\ max_a Q(s',a)$ ▷ **Aprender** de la experiencia
13: $s \leftarrow s'$
14: **until** s′ is the last state of the sequence
15: **end for**

En esta propuesta, el input α no influirá. Sin embargo, el valor de gamma será inicialmente de 1, y se modificará cada vez que se alcance un nuevo estado.

La **política** que se sigue establece que se debe seleccionar aquella acción cuyo valor de Q sea igual al valor máximo de Q existente en el estado en el que se encuentre. Por esta razón, el testeo irá guiado por la ejecución de las acciones con mayor recompensa. Podemos ver esta selección de acciones en la línea 6 del algoritmo.

Tras haber seleccionado una acción, esta se ejecuta (línea 7) y se alcanza un nuevo estado (línea 8).

El factor de descuento γ tiene un valor que se calcula en función del número de acciones que hay en cada estado. Por ello, es necesario determinar este valor (línea 9) y usarlo en la **función de gamma** (línea 10), como podemos ver a continuación:

$$\gamma = 0,9\ e^{-0,1\ (numOfActions-1)}$$

La **función de recompensa** que se sigue está especificada en la línea 11, y se define así:

$$reward(s,a,s') = \frac{1}{x_a}$$

- a es la acción que se ha ejecutado.
- s es el estado del GUI antes de ejecutar a.
- s' es el estado del GUI tras ejecutar a.
- x_a es el número total de veces que se ha ejecutado a en el sistema.

El aprendizaje del sistema ($learn(s,s',a,reward,\gamma,\alpha)$) se lleva a cabo en la línea 12. Se basa en la siguiente **función de Q**:

$$Q(s,a) = reward + \gamma\ max_a Q(s',a)$$

- *Q(s, a)* es el valor correspondiente al Q de la acción a en el estado s.
- *reward* es la recompensa inmediata al ejecutar la acción a en el estado s.
- $max_a Q(s', a)$ es el máximo valor de Q en el estado s' que se obtiene al ejecutar la acción a en el estado s.
- γ es el factor de descuento.

2.3 Esparcia et al. [22]

También podemos destacar el trabajo de Esparcia et al. [22], en el que se investiga el impacto del uso del RL en TESTAR. El algoritmo planteado es el siguiente:

Algoritmo 4 Algoritmo de RL aplicado a TESTAR

Require: R_{max} ▷ recompensa máxima: $R_{max} > 0$
Require: γ ▷ factor de descuento: $0 < \gamma < 1$
Require: α ▷ tasa de aprendizaje: 1
1: $initializeQValues()$
2: **for each** *sequence* **do**
3: $s \leftarrow getStartingState()$
4: **repeat**
5: $availableActions \leftarrow getAvailableActions(s)$
6: $a \leftarrow selectMaxAction(availableActions)$ ▷ **Seleccionar** una acción
7: $performAction(a)$
8: $s' \leftarrow getReachedState()$
9: $reward \leftarrow \begin{cases} R_{max} & \text{if } x_a = 0 \\ \frac{1}{x_a} & otherwise \end{cases}$ ▷ **Recompensa** de la acción
10: $Q(s, a^*) \leftarrow reward + \gamma \ max_{a \in A_{s'}} Q(s', a)$ ▷ **Aprender** de la experiencia
11: $s \leftarrow s'$
12: **until** s′ is the last state of the sequence
13: **end for**

En esta propuesta, se consideran tres inputs: R_{max}, γ y α. El último de ellos tiene un valor de 1, por lo que no afectará al transcurso del algoritmo. Por su parte, gamma (γ) representará un valor comprendido entre 0 y 1; cuanto más pequeño sea, más rápido decrecerá el valor de Q de las acciones seleccionadas. Además, la recompensa máxima (R_{max}) es un valor superior a cero que determina la recompensa inicial que tienen las acciones que no han sido exploradas con anterioridad.

La selección de acciones se rige por la **política** usada, la cual determina que en cada estado se debe seleccionar aquella acción que tenga el máximo valor de Q (línea 6). La representación matemática es la siguiente:

$$a^* \leftarrow max_a\{Q(s, a) \mid a \in A_s\}$$

De este modo, de entre las acciones a que pertenezcan al conjunto de acciones A_s del estado s, se seleccionará la que tenga un mayor valor de Q.

Luego, se ejecuta la acción seleccionada (línea 7) y se pasa al siguiente estado (línea 8).

La **función de recompensa** que se usa en esta propuesta puede verse en la línea 9. Si la acción ha sido ejecutada por primera vez, la recompensa que le corresponde es R_{max}. Por el contrario, si la acción ya ha sido explorada previamente, recibirá una recompensa inversamente proporcional al número de veces que esta ha sido ejecutada:

$$reward = \begin{cases} R_{max} & \text{if } x_a = 0 \\ \frac{1}{x_a} & otherwise \end{cases}$$

- R_{max} es el valor de recompensa inicial de las acciones.

- x_a es el número de veces que se ha ejecutado la acción a.

Cuantas más veces se haya ejecutado una acción, menor recompensa recibirá. Esta heurística premia a las acciones nuevas que hayan surgido en el sistema, y penaliza aquellas que se hayan ejecutado más veces.

Tras haberse calculado la recompensa correspondiente, es el momento de actualizar el valor de Q de la acción. Esto se lleva a cabo en la línea 10 del algoritmo, tomando ejemplo de la definición de la función de aprendizaje $learn(s, s', a, reward, \gamma, \alpha)$. La **función de Q** utilizada es la siguiente:

$$Q(s, a^*) = reward + \gamma\ max_{a \in A_{s'}} Q(s', a)$$

- γ es el factor de descuento.

- $max_{a \in A_{s'}} Q(s', a)$ es el valor de Q máximo que existe en el estado s', y que está asociado a la acción a del conjunto de acciones $A_{s'}$.

- $reward$ es la recompensa calculada.

2.4 Cao et al. [20]

Existe, además, la propuesta de Cao et al. [20], la cual se basa también en el testeo automático de aplicaciones Android a partir de su GUI. Sin embargo, se considera en este caso la agrupación de widgets para la búsqueda de mejores resultados, y no se utiliza RL. En este estudio no se hace uso del Q-learning. En su lugar, se sigue el siguiente algoritmo:

Algoritmo 5 Algoritmo No-RL aplicado al testeo de apps Android mediante agrupación

```
1:  for each sequence do
2:      s ← getStartingState()
3:      repeat
4:          availableActions ← getAvailableActions(s)
5:          for each action do
6:              actionGroup ← action.getGroup()
7:              if actionGroup ∉ previouslySelectedGroups then
8:                  nonVisitedCurrentGroups.add(actionGroup)
9:              end if
10:         end for
11:         if nonVisitedCurrentGroups == ∅ then
12:             a ← selectRandomAction(availableActions)
13:         else
14:             g ← selectRandomGroup(nonVisitedCurrentGroups)
15:             a ← selectRandomActionOf(g)
16:         end if
17:         performAction(a)
18:         s′ ← getReachedState()
19:         previouslySelectedGroups.add(a.getGroup())
20:         nonVisitedCurrentGroups.clear()
21:         s ← s′
22:     until s′ is the last state of the sequence
23: end for
```

Para cada uno de los estados de cada secuencia, se itera sobre las acciones disponibles para obtener el grupo al que pertenece cada una. Esto se lleva a cabo mediante la función *getGroup()* que podemos ver en la línea 6. De este modo, averiguaremos si las acciones corresponden a barras de desplazamiento, pestañas, botones o cualquier otro tipo de widgets.

Existe una lista a la que se van añadiendo los grupos de las acciones que son ejecutadas, y recibe el nombre de *proviousSelectedGroups*. Conocer esta información nos permitirá escoger acciones que no estén relacionadas con dichos grupos, dado que sus características resultan nuevas y ofrecen una exploración más amplia del SUT.

En la línea 7 se comprueba, para cada acción del estado actual, si es del mismo tipo que alguna acción que se ejecutó con anterioridad. En caso negativo, interesa tener en cuenta esta acción; se añade su grupo a una nueva lista llamada *nonVisitedCurrentGroups*, la cual contiene los grupos que corresponden a acciones del estado actual y no a acciones previamente ejecutadas. Matemáticamente su representación es:

$$NVCG = CG \setminus PG = \{g \mid g \in CG \land x \notin PG\}$$

- $NVCG$ (*nonVisitedCurrentGroups*) son los grupos actuales no visitados.
- CG (current groups) hace referencia a los grupos de acciones que existen en el estado actual.
- PG (previous groups) simboliza los grupos de acciones que fueron ejecutadas.
- g representa cada grupo que pertenece a los grupos actuales (CG), pero no a los de acciones ejecutadas (PG).

La **política** de selección de acciones que se usa es la siguiente: En caso de que *nonVisitedCurrentGroups* está vacío (en el primer estado), se seleccionará una acción de entre todas las disponibles de forma aleatoria (línea 12). Por el contrario, si sí existen elementos en dicha lista, se seleccionará aleatoriamente uno de los grupos que contiene, y a continuación se seleccionará una acción que pertenezca a dicho grupo (línea 15).

Luego, se ejecuta la acción seleccionada y se alcanza un nuevo estado (líneas 17 y 18). El grupo al que pertenece es añadido a la lista *previouslySelectedGroups* (línea 19), y se vacía la variable *nonVisitedCurrentGroups* 20.

2.5 Observaciones del estado del arte

Los trabajos anteriormente mencionados apuestan de forma clara por ofrecer nuevas soluciones en el ámbito del testing automatizado, generalmente en el campo del RL. Exploran la aplicabilidad de estas técnicas a distintas plataformas, como entornos de escritorio o sistemas Android.

Sin embargo, no existe un gran número de investigaciones al respecto, y por ello es necesario contribuir al estudio aportando nuevas soluciones y abriendo nuevos caminos.

CAPÍTULO 3

Análisis del problema

3.1 Análisis del funcionamiento de TESTAR

Tal y como se mencionó en la introducción, TESTAR es una herramienta open source que lleva a cabo pruebas automatizadas sin la necesidad de scripts, clasificándose como una herramienta de testing de tipo smart monkey. Implementa un enfoque sin scripts, lo que significa que los casos de test no tienen que ser definidos antes de las ejecuciones de test. En lugar de eso, cada paso de test se genera durante la ejecución, basándose en las acciones disponibles en el momento y estado específicos del GUI.

El principio que subyace en TESTAR es sencillo: generar secuencias de test de *pares (estado, acción)* arrancando la ejecución del SUT (System Under Test) en su estado inicial y seleccionando continuamente una acción para llevar al SUT a otro estado. La selección de acciones representa el problema más básico de los sistemas inteligentes: *qué hacer a continuación*. El objetivo complejo y desafiante es optimizar la **selección de acciones** para encontrar fallos y reconocer estados indeseados cuando se encuentran con un **oráculo**.

El flujo de TESTAR a alto nivel se ilustró en la Imagen 1.1. Los elementos contenidos en el recuadro gris están automatizados. Además, las tres actividades a la derecha representan actividades que podrían ser mejoradas por los testeadores si fuera necesario (por ejemplo definición y selección de acciones, filtros, y oráculos). Después de iniciar el SUT, la herramienta entra en el bucle de seleccionar y ejecutar una **acción**, para así llevar al SUT de un **estado** a otro, hasta que se cumple un criterio de parada. Tras ello, se cierra el SUT.

3.1.1. Estados del GUI en TESTAR

En un GUI existe un conjunto widgets, los cuales pueden ser agrupados según sus características. Algunos de ellos se pueden verse en la tabla 3.1:

Dependiendo del estado en el que se encuentre el GUI, existirán unos widgets u otros. Al llevar a cabo una acción en el sistema, a menudo aparecen o desaparecen widgets de la GUI. Como ejemplo, podemos observar la apariencia de la aplicación Rachota en un estado determinado.

Ventanas	Menús	Controles
• Ventana principal • Ventanas secundarias • Ventanas emergentes • Ventanas de diálogo	• Barras de menú • Menús desplegables • Menús sensibles al contexto	• Botones • Cajas de texto • Enlaces • Radio buttons • Checkboxes • Cuadros de selección desplegables • Deslizadores • Pestañas • Barras de desplazamiento

Tabla 3.1: Ejemplos de widgets de los que puede estar compuesto un GUI

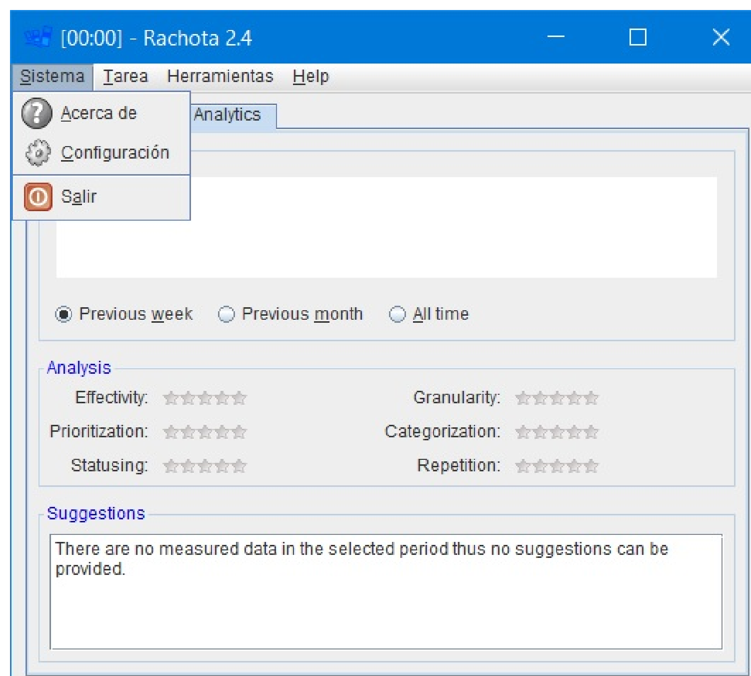

Imagen 3.1: El estado de una GUI.

La imagen anterior representa un estado del SUT Rachota. En él existen cuatro elementos de menú: *Sistema*, *Tarea*, *Herramientas* y *Help*. Además, distinguimos la pestaña *Analytics* y tres elementos de tipo radio button, con los nombres *Previous week*, *Previous month* y *All time*.

Estos widgets se estructuran en una jerarquía llamada **árbol de widgets**. La Imagen 3.2 muestra un ejemplo, basado en el estado de Rachota mencionado.

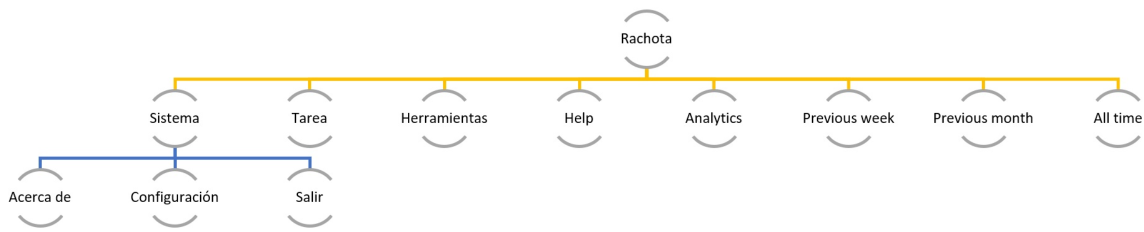

Imagen 3.2: El estado de una GUI como un árbol de widgets.

Cada nodo corresponde a un widget visible y contiene unas propiedades específicas, como su tipo, posición, tamaño, título, si está habilitado, etc. Estos árboles y sus propiedades se pueden obtener automáticamente de varias formas.

Podemos usar **APIs de accesibilidad** – que permiten el uso de ordenadores a gente con discapacidades – al nivel del Sistema Operativo (por ejemplo UIA Automation para Windows, ATK/SPI para Linux o NSAccesibilty para MacOS). Estas APIs de accesibilidad nos permiten recoger información acerca de los widgets visibles de una aplicación, así como dar a TESTAR los medios para consultar los valores de sus propiedades. En concreto, la API UIA da acceso a alrededor de 170 atributos o propiedades [12], lo que nos permite obtener información detallada como:

- El **rol** de un widget. Con esto se podrá conocer si se trata de un botón, un checkbox, un cuadro de selección desplegable, etc.
- La **profundidad** a la que se encuentra un widget en la representación en forma de widget tree de un estado.
- El **tamaño**. Este atributo describe el rectángulo de un widget, necesario para los clicks y otras acciones de ratón.
- Si el widget está **habilitado**. Conocer esto permitirá identificar los widgets deshabilitados para no ejecutar acciones sobre ellos.
- La **ruta** que tiene el widget en la pila de widgets, es decir, en el widget tree.
- Si un widget está **enfocado al teclado**, para saber cuándo se puede escribir en campos de texto.
- **Título, ayuda** y otros atributos descriptivos. Son muy importantes para distinguir unos widgets de otros y proporcionarles una identidad.

Todas estas propiedades y sus valores son almacenados en el *widget tree*. De esta forma, estos árboles capturan el *estado actual* s del GUI como en el ejemplo de la Imagen 3.2. El rol (atributo `Role`) y la profundidad (atributo `ZIndex`) de los widgets se tendrán en cuenta en posteriores capítulos del documento, sirviendo de respaldo para estructurar las soluciones.

Imaginemos que tenemos un árbol de widgets que representa un estado específico s. Los nodos del *widget tree* son los widgets que están visibles en el GUI, en particular en el estado s. Denotaremos este conjunto de nodos de la forma $W(s) = \{w_1, w_2, \ldots, w_k\}$ (por ejemplo, botones, deslizadores, campos de texto, menús, etc).

Las ramas del árbol reflejan las relaciones padre-hijo: cada widget hijo es mostrado dentro del área de la pantalla ocupada por su widget padre. Denotaremos este conjunto

de ramas como $E(s)$. Además, existe una rama directa $(w_i, w_j) \in E(s)$ cuando $w_i \in W(s)$ es el widget padre de $w_j \in W(s)$ en el estado s.

Los valores de todas las propiedades que tienen los widgets también definen el estado. Para un widget $w \in W(s)$ usaremos $P(w,s)$ para denotar el conjunto de todas las propiedades $\{w.p_1, w.p_2, \ldots, w.p_m\}$ (por ejemplo, **rol**, **título**, **posición**, **habilitado**, etc) para ese widget w en el estado s.

Todas las propiedades $P(w,s)$ obtenidas por TESTAR en el estado s para los widgets en $W(s)$ mediante una API o un framework de automatización están asociadas a la representación de TESTAR de las instancias de `State`, `Widget` y `Action`. Esto se lleva a cabo mediante `Tags`, y está representado en la Imagen 3.3.

Las clases etiquetables implementan la interfaz `Taggable`, por lo que los `Tags` pueden ser añadidos a sus instancias. En TESTAR, las clases `Estado`, `Widget` y `Action` son etiquetables y los `Tags` son pares de la forma: (nombre_de_propiedad, valor).

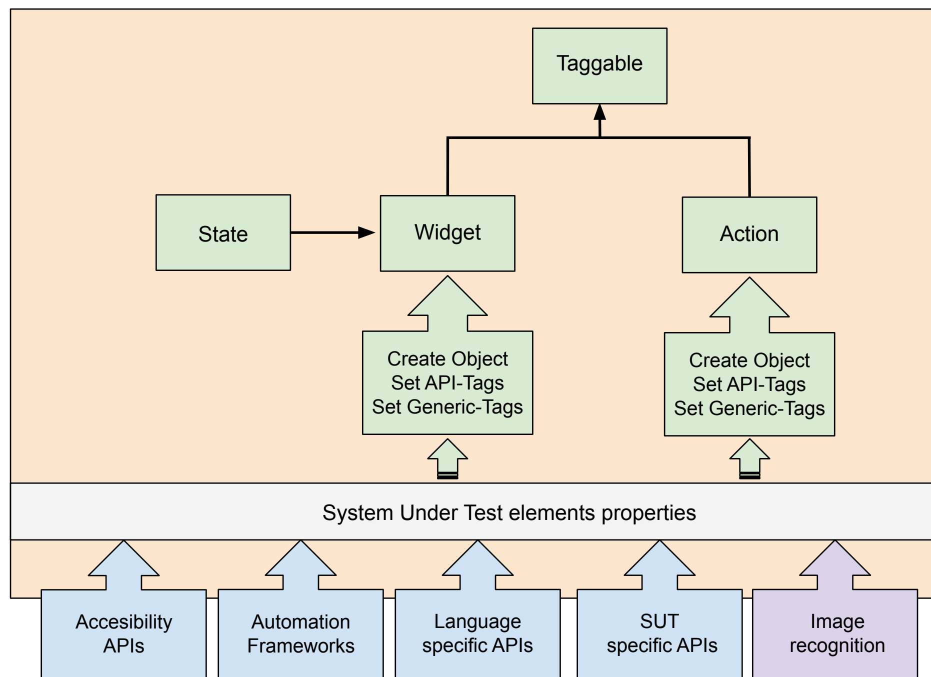

Imagen 3.3: Clases etiquetables `State`, `Widget` y `Action`

3.1.2. Representación de estados y acciones

Para poder reconocer y comparar estados y acciones, necesitamos asignar un identificador único y estable a cada uno de ellos. Para ello, podemos usar atributos, propiedades o valores que vienen con cada widget dentro del árbol de widgets en un estado específico s. Si usamos todas las propiedades, obtenemos lo que llamamos un identificador *concreto*. Sin embargo, no necesitamos usar todos ellos. Podemos seleccionar un subconjunto y usar en su lugar identificadores *abstractos*.

Al seleccionar propiedades para el identificador abstracto, deberíamos tener en cuenta aquellas que son relativamente estables. Por ejemplo, el **título** de una ventana no es normalmente un valor estable (al abrir nuevos documentos en un editor de texto se cambiará el título de la ventana principal), mientras que su texto de ayuda asociado es

menos propenso a cambiar. El **rol**, en cambio, es una propiedad más estable.

Por tanto, para identificar un estado s de la GUI, tomamos todos los widgets $w \in W(s)$ y consideramos simplemente un subconjunto A de propiedades estables de todas las propiedades de todos los widgets en pantalla. Este subconjunto A de **propiedades abstractas** define lo que llamamos una *función de abstracción*.

$$P_{\mathrm{A}}(w,s) \subseteq P(w,s)$$

Es decir:

$$P_{\mathrm{A}}(w,s) = \{w.p \mid w \in W(s) \wedge p \in \mathrm{A}$$

Dado que podría haber un gran número de valores de propiedades a tener en cuenta, solamente guardaremos un valor hash generado a partir de estos valores. TESTAR calcula recursivamente un hash único para cada widget, basándose en la concatenación de los atributos mencionados. Luego, combina los hashes de los widgets y los usa para calcular un hash único para el estado. Evidentemente, esto podría llevar a colisiones. Sin embargo, por simplicidad asumimos que esto es poco probable y no afecta significativamente al proceso de optimización.

3.1.3. Actividad de TESTAR en tiempo de ejecución

El punto de entrada en tiempo de ejecución de TESTAR es la clase Java `Main`. Esta clase tiene acceso al archivo de configuración `test.settings`, definido por el testeador. Además de ajustes como `number_of_sequences`, `number_of_actions` y `SuspiciousTitles`, el testeador puede definir su propia clase de **protocolo** de TESTAR que se necesite usar en el testeo. Este archivo puede estar enfocado a un SUT, a un tipo de test o solamente para un testeador específico. Un protocolo de TESTAR es una clase Java que es responsable de ejecutar diferentes partes del bucle de la secuencia de test, como se muestra en la Imagen 1.1. El código de la clase del protocolo se compila en tiempo de ejecución, también conocido como *runtime execution*.

Los protocolos de TESTAR específicos del SUT, del test o del testeador están en el fondo de un árbol de herencia, tal y como se muestra en la Imagen 3.4.

La clase `DefaultProtocol` contiene todo el código que ejecuta realmente las secuencias de test. Implementa una interfaz, tal y como se definió en la clase `AbstractProtocol`, la cual contiene métodos para ejecutar las diferentes partes del bucle de la secuencia de test (representado en la Imagen 1.1), así como otras cuatro secciones fundamentales:

- `getState()`
- `deriveActions()`
- `selectAction()` y `executeAction()`
- `getVerdict()`

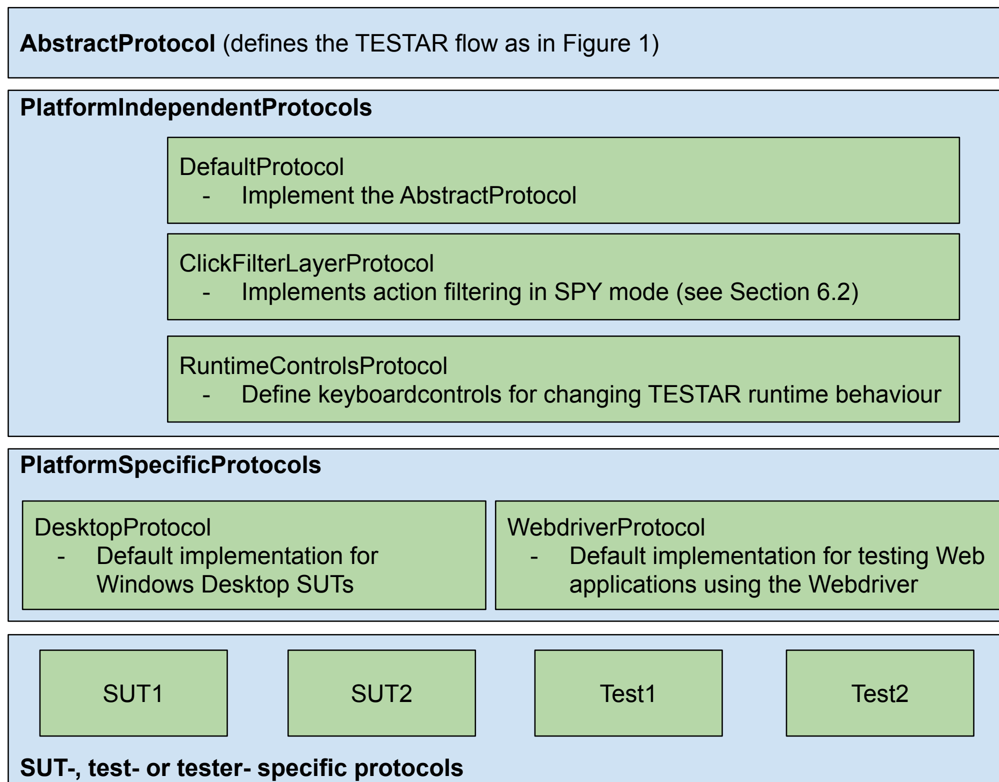

Imagen 3.4: Capas de los diferentes protocolos de TESTAR

`DesktopProtocol` y `WebdriverProtocol` añaden una implementación por defecto para plataformas específicas. La clase `ClickFilterLayerProtocol` lleva a cabo el filtrado de acciones.

Finalmente, existe la clase `RuntimeControlsProtocol`, la cual ofrece controles que permiten la manipulación de los modos en tiempo de ejecución que tiene TESTAR, cuando estos son ejecutados. De entre los principales modos de ejecución, cuatro de ellos representan de *runtime execution*:

- El modo SPY puede ser usado para inspeccionar los widgets del SUT y ver toda la información que TESTAR es capaz de extraer.
- En el modo GENERATE, se ejecuta el ciclo de test, representado en la Imagen 1.1.
- El modo RECORD se puede usar para interactuar manualmente con el SUT almacenar las acciones en secuencias de test.
- El modo REPLAY permite reproducir una secuencia de test existente.

3.2 Análisis del marco para Reinforcement Learning de TESTAR

Para lograr integrar algoritmos de Reinforcement Learning al funcionamiento de TESTAR, explicado en el capítulo 1.2.2, se necesita modificar la estrategia de selección de acciones. TESTAR cuenta con un módulo que permite incorporar diferentes funciones de recompensa, de actualización de Q y políticas de selección de acciones.

Como se muestra en el diagrama de clases de la Imagen 3.5, el módulo *ReinforcementLearning* contiene cuatro submódulos:

- Rewards
- QFunctions
- ActionSelectors
- Policies

Los submódulos *Rewards, QFunctions y ActionSelectors* contienen las interfaces *RewardFuncion, QFunction y Policy*, respectivamente, que definen el funcionamiento que deberán tener todas las funciones de recompensa, las actualizaciones de Q y las políticas de selección que se integren al módulo.

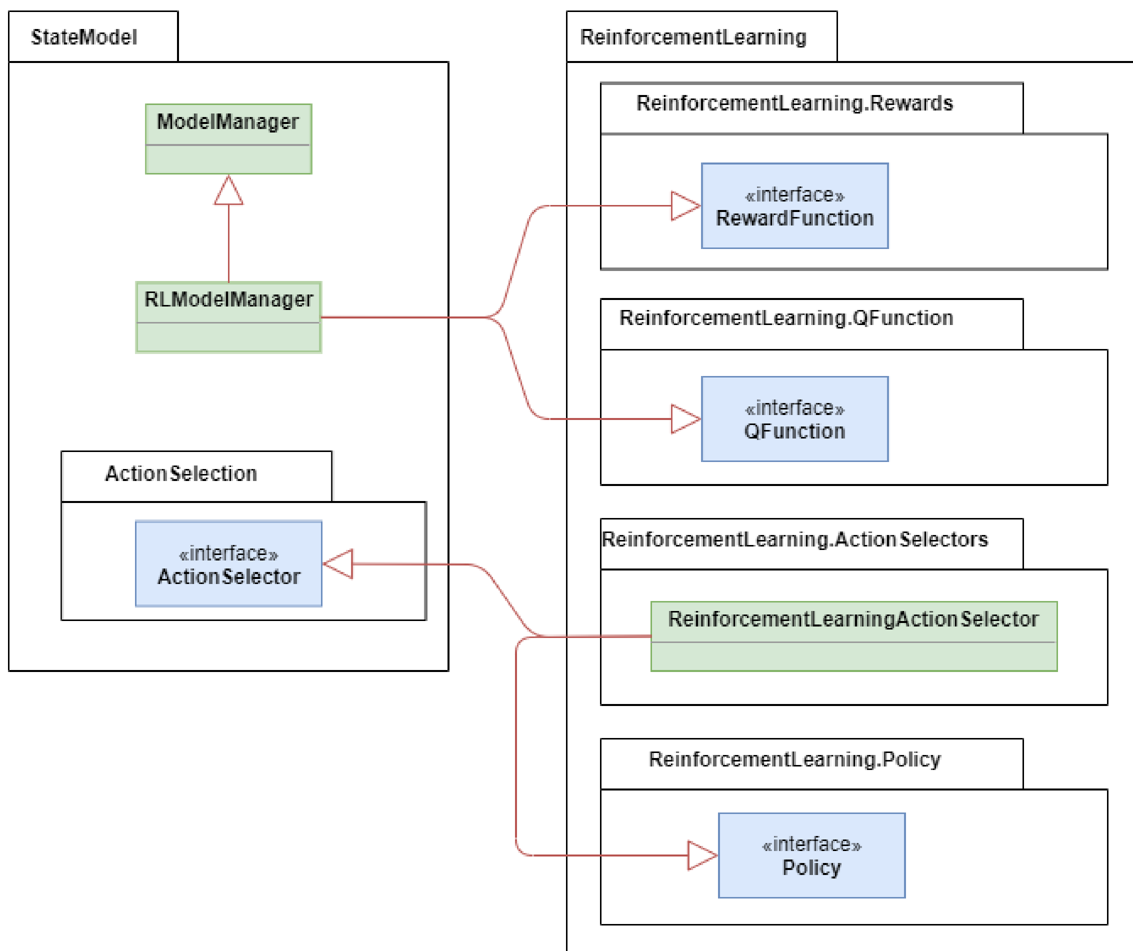

Imagen 3.5: Integración de Reinforcement Learning en TESTAR

Por un lado, el submódulo *ActionSelectors* está compuesto por la clase *ReinforcementLearningActionSelector*, que extiende de la clase *ActionSelector* y se encarga de ejecutar alguna de las políticas de selección de acciones definidas en el submódulo *Policy*, y que implementan la interfaz *Policy*. Es precisamente esta clase la que directamente modifica la estrategia totalmente aleatoria de TESTAR.

Por otro lado, en el módulo *StateModel*, la clase *RLModelManager* contendrá toda la información concerniente al estado actual del SUT para ejecutar los métodos correspondientes y, auxiliándose del módulo de *ReinforcementLearning*, obtener un recompensa basada en el estado, actualizar el valor de Q y seleccionar la futura acción.

El valor de Q se almacena asociado a una acción abstracta, que como se ha explicado en el capítulo 3.1.2, es definida parcialmente por el estado en el que fue ejecutada. De esta forma, se garantiza que la asignación de Q se produzca a un par único de acción-estado.

En lo que respecta a la integración con el algoritmo general de QLearning (1), ubicado en el capítulo 1, esta estructura modular hace posible que se lleven a cabo los procedimientos necesarios de la técnica.

En primer lugar, la interfaz *Policy*, localizada en el submódulo *ActionSlectors*, representa la **política** de selección. Esta escoge una acción de entre todas las disponibles, para ser ejecutada posteriormente. Esto se representa en la línea 7 del algoritmo principal de QLearning, en la que $a \leftarrow selectAction(availableActions)$.

En segundo lugar, la **recompensa** tiene su núcleo en la interfaz *RewardFunction*, alojada en el submódulo *Rewards*. Permite establecer métodos de cálculo de recompensas, las cuales serán asignadas a las acciones ejecutadas, para así premiarlas o sancionarlas. En el algoritmo mencionado, la recompensa está definida por la función *calculateReward(s,a,s')* (línea 10), que toma como parámetros el estado previo (s), la acción ejecutada (a) y el estado alcanzado tras ejecutar la acción (s').

Por último, en la interfaz *QFunction* del submódulo *QFunction* se encuentra la representación de lo que entendemos por la **función Q**. Es la encargada de la asignación de un nuevo valor de Q a la acción que ha sido ejecutada, tomando como parámetros s, s', a, el valor de recompensa y los parámetros γ y α. Este proceso de aprendizaje se sitúa en la línea 11 del algoritmo, y tiene la forma $learn(s, s', a, reward, \gamma, \alpha)$.

En el contexto de TESTAR, la política generalmente seleccionará las acciones con el valor de Q más alto. El cálculo de la recompensa variará dependiendo de la estrategia que se prefiera para premiar o sancionar las acciones. Por su parte, los valores de Q regularmente se actualizarán haciendo uso de la siguiente fórmula:

$Q(s, a) = reward + \gamma \; max_a Q(s', a)$.

En lo que respecta al proceso que sigue TESTAR para seleccionar una acción, calcular su recompensa y actualizar su valor de Q, tomaremos como referencia el siguiente diagrama de secuencia:

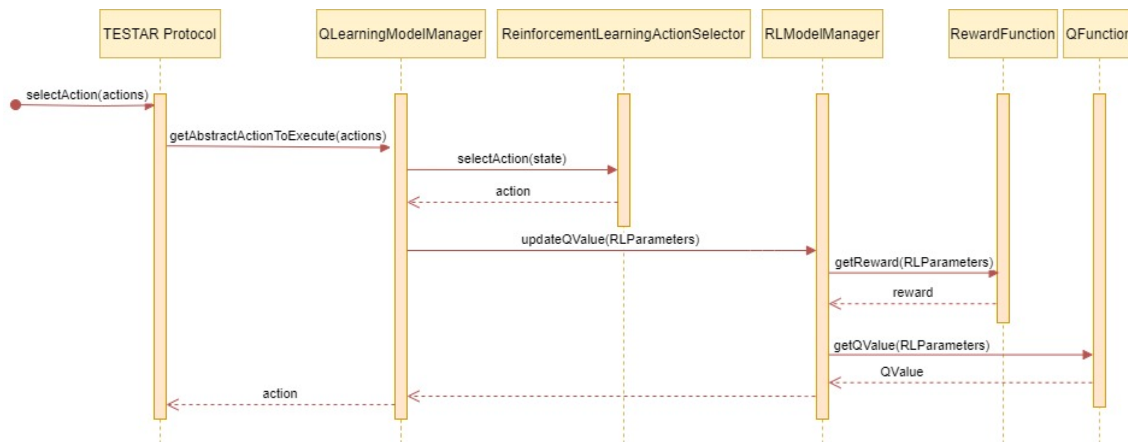

Imagen 3.6: Secuencia de RL en TESTAR

Se busca seleccionar una de las acciones disponibles del estado actual. Para ello, se lanza una petición al protocolo de TESTAR correspondiente, mediante el método *selectAction(actions)*. Después, el protocolo utiliza *getAbstractActionToExecute(actions)* para realizar la consulta a *QLearningModelManager*.

A continuación, se selecciona una acción llamando a *ReinforcementLearningActionSelector* y se alcanza un nuevo estado. Luego, *QLearningModelManager* ejecuta el método *updateQValue(RLParámeters)* para comenzar la actualización del valor de Q de la acción ejecutada.

RLModelManager invoca el método *getReward(RLParameters)* para obtener a la recompensa, accediendo para ello a la clase que representa la función de recompensa correspondiente. Después, se llama a *getQValue(RLParameters)* para obtener el valor de Q final que será asignado.

Finalmente, el nuevo valor de Q es asignado, y la acción seleccionada es devuelta al protocolo correspondiente.

3.3 Análisis del entorno de desarrollo

El desarrollo hace uso de una serie de tecnologías. El código fuente de TESTAR esta alojado en **GitHub**. Esta plataforma de desarrollo colaborativo permite compartir código fuente y mantener un control de versiones. Para comunicar de forma más directa dicho servicio con el código de este trabajo, se ha hecho uso de la aplicación **GitHub Desktop**, la cual ofrece ventajas en la realización y gestión de versiones de código.

El sistema operativo utilizado es **Windows 10**. Además, el sistema de gestión de bases de datos sobre el que se apoya TESTAR es **OrientDB**, con una fuerte orientación a grafos.

Para la codificación de las soluciones utiliza el lenguaje de programación **Java**. Además, el entorno usado ha sido **Eclipse IDE**.

3.4 Identificación y análisis de soluciones posibles

Para lograr los objetivos, se plantean tres enfoques distintos a modo de solución. Estas aproximaciones se explicarán con mayor detalle en el capítulo 4. En lo referente a la integración de Reinforcement Learning en TESTAR es necesario describir una función de recompensa para cada uno de los enfoques propuestos. El resultado puede verse en la siguiente ilustración:

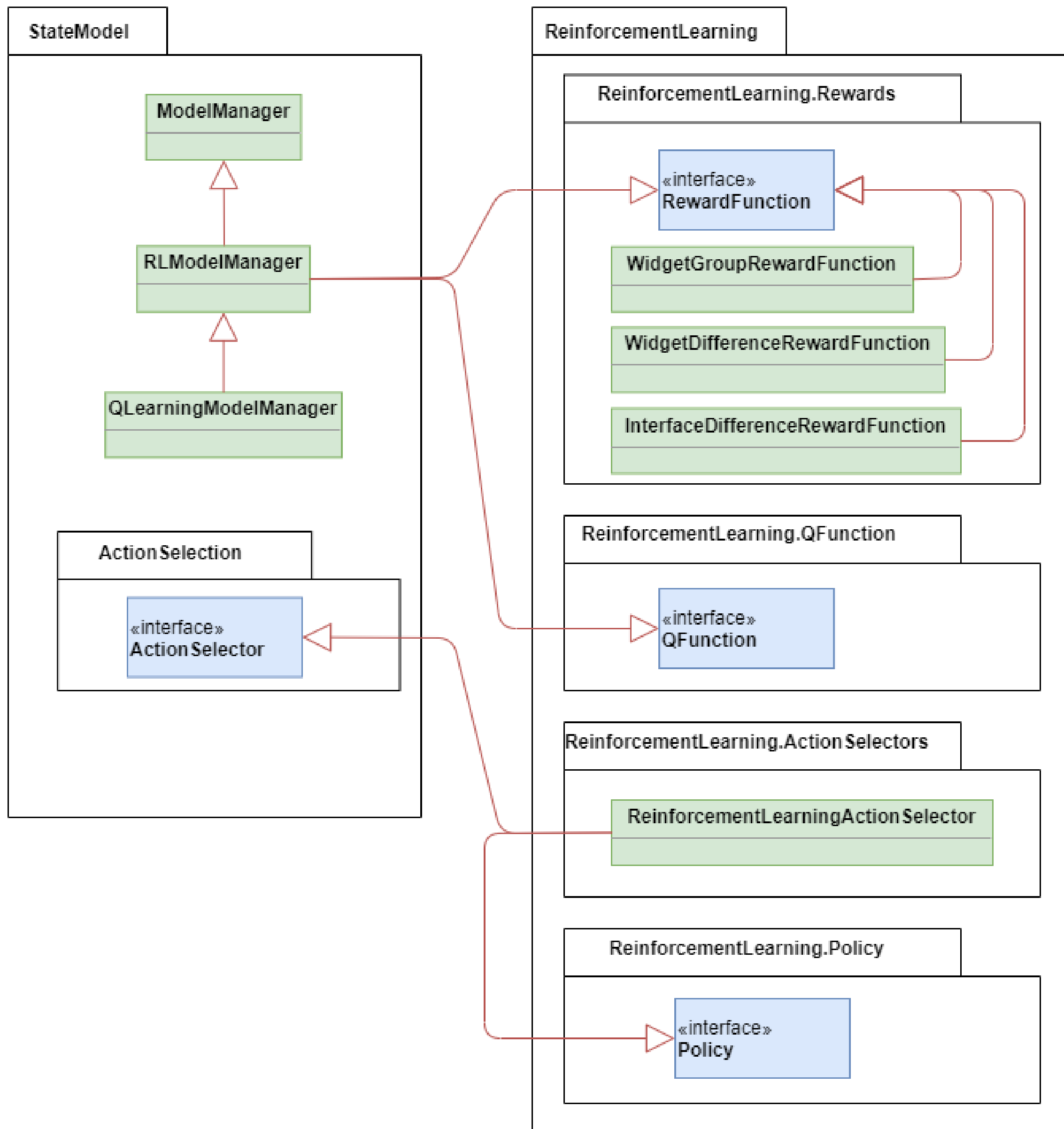

Imagen 3.7: Integración de Reinforcement Learning y funciones de recompensa en TESTAR

Las clases *WidgetGroupRewardFunction*, *WidgetDifferenceRewardFunction* y *InterfaceRewardFunction* extienden la interfaz *RewardFunction*, situada en el submódulo *Rewards*. De este modo, se utilizará una función de recompensa específica para cada enfoque de solución.

Usando en el marco para Reinforcement Learning, se incorporarán nuevos mecanismos de selección de acciones a TESTAR. El objetivo es definir un sistema de recompensas que

mejore el testing. Esto se comprobará haciendo uso de métricas, que evaluarán tras cada acción, secuencia y ejecución, la siguiente información:

- Cobertura de instrucciones
- Cobertura de ramas
- Marca de tiempo

Para recoger estos datos se usará la librería JaCoCo, encargada de medir la cobertura de código.

CAPÍTULO 4

Soluciones propuestas

En este capítulo se explicará la forma en la que se ha llevado a cabo el trabajo. Consideraremos el desarrollo de las tres soluciones propuestas, para finalmente compararlos mediante experimentos controlados.

4.1 Enfoque de selección aleatoria

Asumimos como primer enfoque la forma en la que estaba desarrollada la selección de acciones de TESTAR por defecto, es decir, aleatoriamente.

4.2 Enfoque de agrupación de widgets

En esta heurística se valoran positivamente las acciones que pertenecen a widgets de grupos diferentes. En el capítulo 3.1.1 hemos visto que un estado s de un GUI esta representado por un árbol de widgets, cuyos nodos son los widgets w que están visibles en el GUI en el estado s (por ejemplo, botones, deslizadores, campos de texto, menús, etc). Todos los widgets tienen un **tipo** o un rol (atributo `Role`) y una **profundidad** (atributo `ZIndex`). Estos dos atributos lo vamos a usar para definir un grupo de widgets $G_{t,d}$, el cual está asociado a un tipo de widgets t y a un nivel de profundidad d, siendo t el tipo de un widget w, y d su profundidad:

$$G_{t,d} = \{w \mid getType(w) = t \ \wedge \ getDepth(w) = d\}$$

Mediante el uso de *getType(w)* y *getDepth(w)* obtenemos el tipo y la profundidad de w. Además, diremos que dos acciones son semejantes si los widgets a los que hacen referencia pertenecen al mismo grupo.

Con este enfoque se busca disminuir la probabilidad de que se seleccionen acciones del mismo tipo en igual nivel de profundidad en el árbol de widgets. Por ejemplo, si existen varios botones y un checkbox situados a la misma profundidad, en caso de que se haga click en uno de los botones, se tenderá a seleccionar a continuación una acción relacionada con el checkbox.

Esta propuesta se busca expandir el nivel de exploración del sistema. Su desarrollo como un algoritmo Q-learning puede verse a continuación en el algoritmo 6:

Algoritmo 6 Algoritmo del enfoque de agrupación de widgets

Require: γ ▷ factor de descuento: 0,99
Require: α ▷ tasa de aprendizaje: 1
1: $initializeQValues()$
2: **for each** *sequence* **do**
3: $s \leftarrow getStartingState()$
4: **repeat**
5: $availableActions \leftarrow getAvailableActions(s)$
6: $a \leftarrow selectMaxAction(availableActions)$ ▷ **Selección**
7: $performAction(a)$
8: $s' \leftarrow getReachedState()$
9: $reward \leftarrow -0,05$ ▷ **Recompensa**
10: $learn(s, s', a, reward, \gamma, \alpha)$ ▷ **Aprendizaje**
11: $s \leftarrow s'$
12: **until** s′ is the last state of the sequence
13: **end for**

Antes de iniciar el proceso algorítmico, se introducen los inputs γ (el factor de descuento) y α (la tasa de aprendizaje). El primero de ellos tiene el valor 0,99, y el segundo, 1.

En la línea 6 se selecciona una acción, teniendo en cuenta la **política** existente. En este caso, se escoge la acción con un mayor valor de Q. Después se ejecuta, como puede verse en la línea 7. Es en este punto donde se alcanza un nuevo estado.

Se establece que el valor de **recompensa** de la acción previamente ejecutada será *-0,05* (línea 9). Esto se debe a que este enfoque penaliza la acción que se acaba de ejecutar. Además, en la etapa *learn* se penalizan todas las acciones cuyo widget de origen es el mismo que el del widget correspondiente a la acción ejecutada.

En esta solución no se valoran elementos adicionales para el cálculo de la recompensa, por lo que las actualizaciones del valor de Q de las acciones son lineales y moderadas, permitiendo el aprendizaje del sistema (línea 10). La función *learn* utilizada se desglosa de la siguiente forma:

Algoritmo 7 Función $learn(s, s', a, reward, \gamma, \alpha)$ en el enfoque de agrupación de widgets

1: $Q(s, a) \leftarrow reward + \gamma\ max_a Q(s', a)$
2: $selActionType \leftarrow getType(a)$
3: $selActionDepth \leftarrow getDepth(a)$
4: $availableActions \leftarrow getAvailableActions(s')$
5: **for each** $action \in availableActions$ **do**
6: $actionType \leftarrow getType(action)$
7: $actionDepth \leftarrow getDepth(action)$
8: **if** $(selActionType == actionType)$ && $(selActionDepth == actionDepth)$ **then**
9: $Q(s, action) \leftarrow Q(s, a)$
10: **end if**
11: **end for**

En la línea 1 del algoritmo 7 se calcula y se asigna el valor de Q definitivo a la acción ejecutada, haciendo uso de la **función Q**:

$$Q(s, a) = reward + \gamma\ max_a Q(s', a)$$

Dado que en este caso los valores de α y γ apenas influyen, el cálculo del nuevo valor de Q de la acción se hará sumando la recompensa (-0,05) al valor de Q existente.

El siguiente paso es comprobar si alguna de las acciones del estado actual es semejante a la acción que fue ejecutada, es decir, si los widgets asociados a dichas acciones pertenecen al mismo grupo de widgets. Para ello, primero se obtienen las características de interés de la acción ejecutada, como se puede ver en las líneas 2 y 3. Después se sigue el mismo procedimiento con cada una de las acciones del estado actual (líneas 6 y 7).

En caso de que los valores de profundidad y tipo de la acción ejecutada coincidan con los de alguna acción actual (línea 8), el valor de Q de esta última pasará a ser el mismo que el de la acción ejecutada. Esta actualización de valores se realiza en la línea 9, y tiene como principal objetivo establecer la misma prioridad de selección para acciones con una similitud muy alta. Con esto conseguimos que, de ahora en adelante, además de evitar ejecutar la acción previa, TESTAR evitará ejecutar acciones semejantes a ella, anteponiendo la exploración de nuevos escenarios del SUT.

Finalmente, como podemos ver en la línea 11 del algoritmo 6, s toma el valor del estado actual s', para que así, en las posibles próximas iteraciones, represente al estado previo.

4.3 Enfoque de diferencia de widgets

Este enfoque se basa en premiar las acciones que produzcan un aumento en el número de widgets existentes, o aquellas que lleven a estados en los que haya persistido el menor número de widgets respecto al estado anterior, ocasionando la generación de widgets nuevos. De este modo, se buscará ejecutar en mayor proporción aquellas acciones que provoquen que el widget tree del estado alcanzado tenga un mayor tamaño que el del estado actual.

Para conseguirlo, se analiza el árbol de widgets del estado anterior a la ejecución de una acción y el árbol posterior. En caso de que el número de widgets se vea aumentado, lo hará también el valor de recompensa de la acción ejecutada. Por el contrario, si en el estado al que se ha llegado hay un menor número de widgets, el valor de recompensa de la acción decrecerá. Podemos ver todo el proceso en el algoritmo 8.

Algoritmo 8 Algoritmo del enfoque de diferencia de widgets

Require: γ ▷ factor de descuento: 0,99
Require: α ▷ tasa de aprendizaje: 1
1: $initializeQValues()$
2: **for each** $sequence$ **do**
3: $s \leftarrow getStartingState()$
4: **repeat**
5: $availableActions \leftarrow getAvailableActions(s)$
6: $a \leftarrow selectMaxAction(availableActions)$ ▷ **Selección**
7: $performAction(a)$
8: $s' \leftarrow getReachedState()$
9: $reward \leftarrow calculateReward(s, a, s')$ ▷ **Recompensa**
10: $\gamma \leftarrow 0.01\ getDepth(a)$
11: $learn(s, s', a, reward, \gamma, \alpha)$ ▷ **Aprendizaje**
12: $s \leftarrow s'$
13: **until** s′ is the last state of the sequence
14: **end for**

Los valores de entrada γ y α, al igual que en el enfoque anterior, tienen los valores 0,99 y 1, respectivamente. Sin embargo, se actualizará el valor del primero de ellos más adelante.

La **política** de selección establece que se elegirá aquella acción que tenga el mayor valor de Q de entre las acciones del estado (línea 6). Tras ello es ejecutada (línea), llegando así a un nuevo estado.

La **función de recompensa** se adapta a la diferencia de widgets que existen entre los estados s' y s, y se encuentra en la línea 9. Sus características se exponen en el algoritmo 9:

Algoritmo 9 Función $calculateReward(s, a, s')$ en el enfoque de diferencia de widgets

1: $persistentDecrement \leftarrow getPersistentWidgetNum(s, s')$
2: **if** numWidgets(s) < numWidgets(s') **then**
3: $\quad persistentDecrement \leftarrow 0,01\ (persistentWidgetNum/numWidgets(s))$
4: $\quad widgetDifference \leftarrow numWidgets(s)/numWidgets(s')$
5: $\quad reward \leftarrow -\ halfAssociator(persistentDecrement + widgetDifference)$
6: **else if** numWidgets(s) > numWidgets(s') **then**
7: $\quad persistentDecrement \leftarrow 0,01\ persistentWidgetNum/numWidgets(s')$
8: $\quad widgetDifference \leftarrow numWidgets(s')/numWidgets(s)$
9: $\quad reward \leftarrow -\ (1 - halfAssociator(persistentDecrement + widgetDifference)))$
10: **else**
11: $\quad persistentDecrement \leftarrow persistentWidgetNum/numWidgets(s')$
12: $\quad reward \leftarrow -\ persistentDecrement$
13: **end if**

Para comprender el proceso, llamaremos **numWidgets**(s) al número de widgets en el estado previo s, y **numWidgets**(s') al número de widgets en el estado actual s'.

La recompensa se calcula teniendo en cuenta dos parámetros. En primer lugar, *widgetDifference* representa la proporción de widgets que hay en el estado actual respecto al estado previo. En segundo lugar, *persistentDecrement* es un decremento por la existencia de widgets que persisten en el GUI. Este último tiene un valor bajo, porque la importancia de este algoritmo recae en *widgetDifference*. El decremento por widgets persistentes pretende ser un añadido y no el elemento más notable.

halfAssociator es una función que establece un rango de recompensas para los distintos casos que puedan darse, dando prioridad a las situaciones en las que se produzca un incremento de widgets:

- Si numWidgets(s) < numWidgets(s'): $-\ 0,5 < reward < 0$
- Si numWidgets(s) > numWidgets(s'): $-\ 1 < reward < -\ 0,5$

Para conseguir situar la recompensa en uno de estos rangos, primero debemos determinar en cuál de los dos casos nos encontramos. Si es el primero, se deberá multiplicar el valor de recompensa que hayamos ido calculando por 0,5. Si se trata del segundo caso, en su lugar lo multiplicaremos por 0,5 y sumaremos después 0,5. Este proceso acomodará el valor de recompensa que existía provisionalmente, para que se sitúe proporcionalmente en uno de los dos rangos.

Por otro lado, en el cómputo de *persistentDecrement* se considera el parámetro *persistentWidgetNum*, que representa el número de widgets que han persistido en el GUI al pasar del estado s a s'.

La recompensa nunca será un valor positivo. Si esto fuera así, el valor de Q de las acciones podría superar el valor con el que se inicializan (1), por lo que no se permitiría ejecutar acciones con un valor de Q más bajo. De hecho, al ejecutar acciones que únicamente incrementan su valor de Q se podrían crear bucles de ejecución de los que no es posible salir. Por ello, se premiará a las acciones haciendo que dicha recompensa negativa sea más cercana a cero, y se las penalizará del modo contrario.

En el cálculo de la recompensa podemos distinguir diferentes casos, que aparecen de la línea 2 a la línea 13 en el algoritmo 9:

- **Si numWidgets(s) < numWidgets(s')**:

  $$reward = -\ halfAssociator(persistentDecrement + widgetDifference)$$

  De este modo, cuantos más widgets nuevos se hayan generado al ejecutar una acción, la recompensa estará más próxima a 0, por lo que se le restará una menor cantidad al valor de Q de dicha acción.

  $persistentDecrement = 0{,}01\ \frac{persistentWidgetNum}{numWidgets(s)}$

  $widgetDifference = \frac{numWidgets(s)}{numWidgets(s')}$

- **Si numWidgets(s) > numWidgets(s')**:

  $$reward = -\ (1 - halfAssociator(persistentDecrement + widgetDifference)))$$

  A diferencia del caso anterior, cuanta más diferencia de widgets exista, la recompensa deberá ser más lejana a 0. Así se restará una mayor cantidad al valor de Q de la acción ejecutada. Es por ello que se considera la inversa, restando a 1 el valor de *halfAssociator(persistentDecrement + widgetDifference)*.

  $persistentDecrement = 0{,}01\ \frac{persistentWidgetNum}{numWidgets(s')}$

  $widgetDifference = \frac{numWidgets(s')}{numWidgets(s)}$

  Los cálculos de estos parámetros difieren del caso anterior. En primer lugar, *persistentDecrement* tiene en cuenta el número de widgets de s', y no de s. En segundo lugar, *widgetDifference* se calcula dividiendo *numWidgets(s')* entre *numWidgets(s)*, para así obtener un valor entre 0 y 1, proporcional a la variación de widgets que se ha dado.

- **Si numWidgets(s) == numWidgets(s')**:

  $$reward = -\ persistentDecrement$$

  Si al ejecutar la acción no ha habido un incremento o decermento en el número de widgets, se considerará únicamente el decremento por widgets que han persistido de un estado a otro.

  $persistentDecrement = Q(s,a)\ \frac{persistentWidgetNum}{numWidgets(s')}$

El factor de descuento γ será el resultado de multiplicar 0,01 por la profundidad de la acción ejecutada anteriormente (línea 10). De este modo, una acción *a1*, correspondiente a un widget *w1*, decrementará su valor de Q de una mayor forma que una acción *a2*, correspondiente a un widget *w2*, si la profundidad de *w1* es mayor a la de *w2*.

El cálculo y la asignación del valor de Q se realiza del mismo modo que en el enfoque anterior (línea 11). La **función Q**, basada en la función $learn(s, s', a, reward, \gamma, \alpha)$, tiene la siguiente forma:

$$Q(s, a) = reward + \gamma \ max_a Q(s', a)$$

Para la asignación del valor de Q a la acción a se tienen en cuenta datos calculados anteriormente, como la recompensa o el factor de descuento γ. Una vez terminado el proceso de aprendizaje, a s se le asigna el valor de s' (línea 12).

4.4 Enfoque de diferencia visual

La propuesta de este enfoque parte del mismo razonamiento que el enfoque anterior. En este caso también se busca dar una mayor importancia a las acciones que produzcan un mayor cambio. Sin embargo, para llevarlo a cabo, se analiza aquí el número de píxeles de la GUI que ha cambiado de un estado a otro. Esta heurística premia a aquellas acciones que ocasionen un mayor cambio visual. El algoritmo seguido es el siguiente:

Algoritmo 10 Algoritmo del enfoque de diferencia visual

Require: γ ▷ factor de descuento: 0,99
Require: α ▷ tasa de aprendizaje: 1
1: $initializeQValues()$
2: **for each** $sequence$ **do**
3: $s \leftarrow getStartingState()$
4: **repeat**
5: $availableActions \leftarrow getAvailableActions(s)$
6: $a \leftarrow selectMaxAction(availableActions)$ ▷ **Selección**
7: $performAction(a)$
8: $s' \leftarrow getReachedState()$
9: $reward \leftarrow calculateReward(s, a, s')$ ▷ **Recompensa**
10: $\gamma \leftarrow 0.01 \ getDepth(a)$
11: $learn(s, s', a, reward, \gamma, \alpha)$ ▷ **Aprendizaje**
12: $s \leftarrow s'$
13: **until** s′ is the last state of the sequence
14: **end for**

Al igual que en los enfoques anteriores, se toman dos inputs, gamma (γ) y alpha (α), con los valores 0,99 y 1, respectivamente. Más adelante, el valor del primero de ellos será modificado.

En la línea 6 del algoritmo 10 se selecciona la acción que será ejecutada, teniendo en cuenta la **política** de selección que existe. En este enfoque se escoge aquella acción con el mayor valor de Q. Después, esta es ejecutada y se alcanza un nuevo estado s', como podemos ver en la línea 8.

En lo que respecta al cálculo de la recompensa, en la línea se ejecuta la **función de recompensa** *calculateReward(s, a, s′)*, que está definido en el algoritmo :

Algoritmo 11 Función $calculateReward(s, a, s')$ en el enfoque de diferencia visual

1: $prevStateScreenShot \leftarrow getScreenShot(s)$
2: $currStateScreenShot \leftarrow getScreenShot(s')$
3: $diffPxProportion \leftarrow getDiffPxProportion(prevStateScreenShot, currStateScreenShot)$
4: $reward \leftarrow -(1 - diffPxProportion)$

Para el cálculo de la recompensa, es necesario calcular la proporción de píxeles diferentes entre el estado anterior y posterior a ejecutar la acción (*diffPxProportion*). Con ese fin, obtenemos una captura de pantalla de la GUI de ambos estados, tal y como se ve en las líneas 1 y 2 del algoritmo 11.

Es en la línea 3 donde se calcula la relación de píxeles iguales que existe entre ambos estados. Esto se lleva a cabo iterando sobre cada uno de los píxeles de las imágenes y señalando cuáles de ellos han cambiado tras la ejecución de la acción. De este modo, podemos cuantificar el grado de cambio visual que se ha ocasionado.

La recompensa, al igual que en el enfoque anterior, es negativa. De este modo se evita los valores de Q sobrepasen el valor inicial (1) y se seleccionen siempre las mismas acciones, las cuales tienen el valor de Q más alto.

El valor que devuelve *diffPxProportion* está comprendido entre 0 y 1, y es mayor cuantos más píxeles hayan variado. Sin embargo, a mayor cambio, se deberá premiar más a la acción y restar una menor cantidad a su valor de Q, por lo que la recompensa (siendo negativa) debe estar más próxima a 0. Por esta razón, se opta por la inversa de *diffPxProportion)*, como podemos ver en la línea 4.

Una vez calculada la recompensa, se asigna a γ el valor resultante de multiplicar 0,01 por la profundidad de la acción a (línea 10 del algoritmo 10). Esto es así para que el decremento sea mayor en acciones asociadas a widgets con una mayor profundidad. De este modo, se tiende a seleccionar acciones situadas a una menor profundidad, y así explorar más ampliamente el SUT.

Por último, en la línea se ejecuta la función de Q $learn(s, s', a, reward, \gamma, \alpha)$, adaptada a continuación:

$$Q(s, a) = reward + \gamma \ max_a Q(s', a)$$

CAPÍTULO 5

Validación

En este trabajo se han definido tres mecanismos de selección de acciones diferentes, partiendo del ya existente, el cual se fundamenta en el criterio de la aleatoriedad.

Con el fin de validar si estas estrategias mejoran el testeo con TESTAR, así como cuantificar en qué medida esto se lleva a cabo, ejecutaremos diversos experimentos controlados.

De acuerdo con Wohlin et al. [36], un experimento controlado consiste en un conjunto de pruebas, donde cada una de ellas puede ser considerada como una *variable* de medición al aplicar un *tratamiento* a un *objeto*. El proceso que se propone es el siguiente:

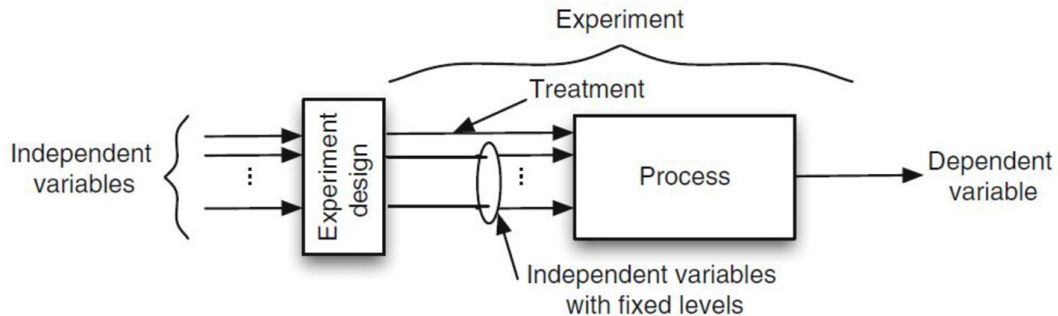

Imagen 5.1: Proceso de un experimento

En el contexto del testeo con TESTAR, un experimento controlado consiste en:

- **Testear** diferentes SUTs (capítulo 5.4).
- **Aplicar** el tratamiento TESTAR a los cuatro mecanismos de selección de acciones diferentes (capítulo 5.7.2).
- **Medir** variables relacionadas con la efectividad y la eficiencia (por ejemplo, el número de fallos, la cobertura, la duración o el número de estados) (capítulo 5.5).

El proceso de recolección de datos y extracción de conocimiento puede esquematizarse de la siguiente forma:

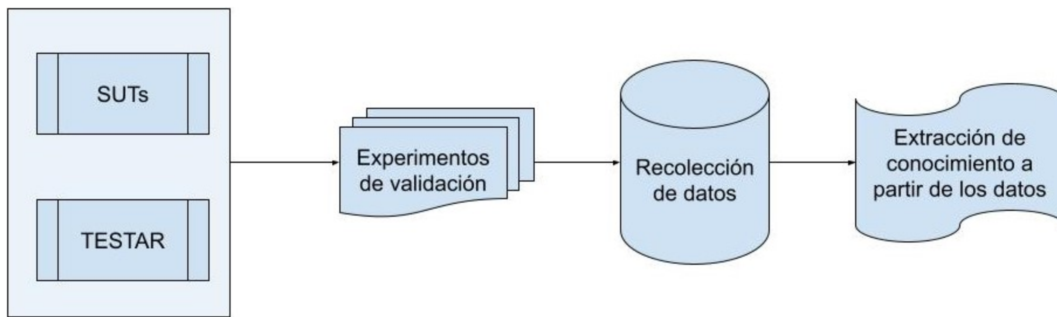

Imagen 5.2: El proceso de recolección de datos y extracción de conocimiento.

5.1 Preguntas de investigación

La respuesta a una pregunta de investigación ayudará a abordar una pregunta o problema de investigación [19]. Es por ello que es necesario acotar la dirección que tomará el presente estudio, teniendo en cuenta los objetivos perseguidos.

En primer lugar, y previamente a identificar la pregunta de investigación global, se plantea la siguiente pregunta, de carácter cualitativo:

¿Cuáles son los problemas que existen en el mecanismo de selección de acciones actual de TESTAR?

Con ella se pretende poner el foco en aquellas características del sistema que, por cuestiones de eficiencia, efectividad o fiabilidad, presentan dificultades o podrían ser mejoradas. En segundo lugar, consideramos la siguiente pregunta:

¿Existe una correlación entre el método de selección de acciones usado en TESTAR y la calidad de los datos de salida?

Esta pregunta valora la posibilidad de que criterios distintos de selección de acciones puedan ocasionar funcionamientos diferentes de TESTAR. Por esta razón, los outputs que emite el sistema en forma de secuencias de test se basarían en exploraciones distintas de los widgets del sistema, y producirían datos diferentes acerca de variables como la cobertura.

Sin embargo, de forma general, planteamos la siguiente pregunta de investigación que hace referencia a la cuestión principal que se aborda en este trabajo:

¿Cuál es la estrategia más ventajosa para abordar la selección de acciones en TESTAR y cuánta diferencia presenta respecto a otras?

Esta es de tipo mixto, dado que combina un planteamiento cualitativo (la identificación de la mejor opción) con otro cuantitativo (cuál es la diferencia numérica medida en cada caso).

5.2 Contexto

En el contexto en el que se desarrolla este trabajo, la técnica de selección de acciones por defecto de TESTAR es la selección aleatoria. Este hecho es de interés y será considerado para construir los pilares de los nuevos planteamientos.

5.3 Hipótesis

Para este trabajo plantearemos una hipótesis nula, la cual servirá como punto de referencia para situar la información de las conclusiones extraídas. La premisa que consideraremos es la siguiente:

H_0: No existe una gran diferencia en los valores de los parámetros medidos, aplicando distintos mecanismos de selección de acciones en TESTAR.

La aleatoriedad (presente en TESTAR y RL) podría perjudicar la fiabilidad de las conclusiones, al realizar el análisis empírico de algoritmos aleatorios. Por tanto, es necesario hacer dos cosas [15]:

1. Decidir cuántos experimentos deben ejecutarse para obtener resultados fiables. Usaremos 30, siguiendo el *rule of thumb* de J. A. Rice[27].

2. Saber cómo evaluar de forma rigurosa si esos resultados son realmente fiables. Para ello se deben usar tests estadísticos.

La técnica estadística t-test supone que los datos tendrían una distribución normal. Sin embargo, con el testing aleatorio no podemos asegurarlo [16]. Por ello, se usará el test estadístico no parametrico de Mann Withley U [29], que no supone normalidad.

El test U funciona bien para muestras de pequeño tamaño, como es el caso actual. Para cada SUT y mecanismo de selección consideramos un total de 30 resultados de cobertura. El uso de esta técnica se detallará en el capítulo 6.

5.4 SUTs

Los tres sistemas que estarán bajo test son Spagetti[7], SwingSet2[8] y Rachota[6], y corresponden a los objetos. El primero de ellos ha sido desarrollado por un estudiante de la Open University of the Netherlands, específicamente para estos experimentos. Como se ha visto anteriormente, a ellos se les aplicarán los tratamientos oportunos. A continuación puede verse una tabla con algunas especificaciones acerca de ellos.

Especificaciones de los SUTs			
Métrica	Spaghetti	SwingSet2	Rachota
Paquetes	1	1	3
Clases Java	1	31	52
Métodos	45	290	934
Líneas de código	350	7029	2722
Aplicación	Experiment	Swing Demo	Business
Java Swing	Sí	Sí	Sí
Clases internas incluidas	14	138	327

5.5 Variables

Las variables del experimento pueden ser de dos tipos: dependientes e independientes. Por un lado, las **variables dependientes** corresponden al output del experimento, y dependen de cada ejecución. Por otro lado, las **variables independientes** se pueden subdividir en aquellas a las que se les ha dado un valor constante, y aquellas que usamos para controlar el experimento y cuyo valor puede cambiar; estas últimas reciben el nombre de factores.

5.5.1. Variables independientes

- Mecanismos de selección de acciones de TESTAR: Corresponden a los cuatro enfoques planteados, siendo el primero de ellos el predeterminado (basado en una selección aleatoria), y los demás, los propuestos como solución en este trabajo. Los archivos de protocolos de TESTAR controlarán los mecanismos de selección de acciones.

De los parámetros en los archivos *test.settings* consideraremos los siguientes:

- Filtros de acciones: Serán los mismos durante el experimento para un SUT, pero variará de un SUT a otro, acorde a la configuración necesaria. En el apéndice A se incluyen tres tablas, en las que se profundiza en cuál es el conjunto de filtros que está asociado a cada uno de los SUTs.
- Tiempo de espera entre clicks: 0.3 segundos
- Duración de acción: 0.3 segundos
- Tiempo máximo de inicio del SUT: 60 segundos
- No habrá oráculos, porque queremos medir cobertura.

En las configuraciones de ejecución de tests definiremos:

- Número de secuencias de test por ejecución: 10
- Número de acciones por secuencia: 300

5.5.2. Variables dependientes

Las variables dependientes que mediremos serán usadas para responder la pregunta de investigación y validar la hipótesis. Tendremos en cuenta, tras cada acción, secuencia y

ejecución, los siguientes datos:

- Cobertura de instrucciones
- Cobertura de ramas
- Marca de tiempo

5.6 Diseño

Los mismos experimentos se ejecutarán en paralelo en 30 máquinas virtuales distintas, generando una gran cantidad de datos.

Haciendo uso de la librería JaCoCo, se recogerá la información correspondiente a la cobertura del código. Para ello, se realizarán mediciones después de cada acción, secuencia de acciones y ejecución completa.

Los datos obtenidos se recopilarán en un archivo de métricas, y se almacenarán en el file server central que usan los workers. Estos elementos se explicarán con más detalle en el capítulo 5.7.1. Los archivos obtenidos contendrán la información de interés en términos de cobertura, y serán objeto de estudio para sacar conclusiones.

Será necesario organizar los datos en una estructura manejable, distinguiendo los distintos algoritmos de selección de acciones existentes. Con tal de evaluar correctamente la información, se crearán gráficas que ilustrarán visualmente las características de cada enfoque.

Finalmente, se compararán los resultados obtenidos y se determinará el impacto de cada mecanismo de selección de acciones analizado.

5.7 Instrumentación

Las herramientas serán usadas para llevar a cabo los experimentos y recoger los datos pertinentes. Destacamos las siguientes:

- Las máquinas virtuales y lworkers.
- Los *test settings* de TESTAR para establecer conexiones con los SUTs.
- Utilidades de JaCoCo

5.7.1. Las máquinas virtuales y los workers

En lo que respecta a lanzar los experimentos, se deberán ejecutar las instrucciones pertinentes desde la máquina controladora. Está alojada en el equipo remoto 10.101.0.100, y coordina las operaciones de cada uno de los treinta workers. Los workers atienden las peticiones del controlador, y se les asignará la la ejecución de 10 secuencias de 300 acciones por cada SUT y por cada mecanismo de selección de acciones, para llevar a cabo las ejecuciones.

La preparación de las instrucciones se realiza en la máquina controladora. Se pueden lanzar en el *cmd* de Windows, pero si en su lugar se usa *bash* se habilita una terminal Linux que facilita los comandos. El orden de ejecución a seguir es el siguiente:

1. Reiniciar el sistema.
2. Preparar la resolución de pantalla.
3. Preparar TESTAR.
4. Preparar OrientDB
5. Lanzar la ejecución para un SUT y un mecanismo de selección determinados.

Esto se hace para que lo sucedido cada ejecución no influya en las siguientes, y así los datos obtenidos tengan una mayor fiabilidad. Para ello, tras reiniciar el sistema se configura la resolución pertinente, se limpia el entorno y se vuelven a copiar TESTAR y OrientDB. Es entonces cuando se llevan a cabo las ejecuciones.

Para llevar a cabo esta secuencia, se ejecutará un archivo *.bat* en cada etapa. Además, se seguirán estos cinco pasos siempre que se quieran lanzar a ejecución experimentos para un SUT y un mecanismo de selección concretos.

Podemos tener distintos archivos para ejecutar, por lo que se deberá establecer un orden entre ellos. Al lanzar las instrucciones se colocarán en una cola, y cada worker se encargará de su ejecución.

5.7.2. Los *test settings* de TESTAR y los SUTs

Las variables independientes a considerar se almacenan en el archivo de configuración *test.settings*, específico para cada SUT. Para tener una imagen clara del contenido de este fichero, las configuraciones principales se esquematizan en el apéndice A.

En este archivo también se encuentra el parámetro *ExtendedSettingsFile*, en el que se indica la ruta de otro archivo que actúa como una extensión del anterior, y cuya función es reunir las configuraciones relacionadas con mecanismos de selección de acciones determinados.

Los archivos referenciados en *ExtendedSettingsFile* son documentos XML, y en el contexto actual representan a cada uno de los tres enfoques de solución que se han propuesto en el capítulo 4. De entre los parámetros que se almacenan en ellos, destacamos los siguientes:

- *<policy>*: Representa el identificador de la política de selección de acciones que se sigue en dicho enfoque.
- *<rewardFunction>*: Simboliza la función a la que se llama para calcular la recompensa.
- *<qFunction>*: Muestra qué función Q es la encargada de asignar a la acción previamente ejecutada el nuevo valor de Q.

El protocolo del SUT que define el comportamiento de la interacción se encargará de:

- **Considerar acciones definidas previamente**, como por ejemplo llenar campos de texto o hacer click en botones.
- **Forzar acciones**. Esto permite tomar el control del rumbo que toma la ejecución.
- **Filtrar widgets indeseados**. Se realiza, por ejemplo, para evitar que se abran ventanas adicionales.

También se debe tener en cuenta el refresco de estados del SUT:

- **Configurar el tiempo mínimo de espera entre acciones**. De este modo, podremos inferir correctamente el Modelo de Estado y evitar que se creen modelos incorrectos. Fijaremos este valor a 0.3 segundos.

Por último, debemos considerar la abstracción del SUT y los widgets dinámicos o emergentes:

- **Identificar propiedades dinámicas de los widgets**, que están en constante cambio.
- **Configurar una abstracción correcta** inspeccionando las propiedades de los widgets. Por ejemplo, en Rachota es necesario quitar "Widget Title" del nivel de abstracción, para así evitar que sea modificado y que continuamente se generen nuevos estados y acciones abstractas.

5.7.3. JaCoCo

JaCoCo es un generador de informes de cobertura de código para proyectos Java. Cada vez que se ejecuta una acción, se guarda información acerca de la cobertura de instrucción y de ramas.

- **Cobertura de instrucciones JaCoCo**: Proporciona información sobre la cantidad de código que se ha ejecutado o se ha omitido. La cantidad de total de instrucciones de una aplicación se determina por JaCoCo durante una ejecución de test (u otro tipo de ejecución) de la aplicación, y se muestra en el informe de JaCoCo.
- **Cobertura de ramas JaCoCo**: Se cuenta el número total de ramas de todas las sentencias if y switch que se han alcanzado. Jacoco determina durante la ejecución de la aplicación la cantidad total de ramas, y la muestra en el informe correspondiente.

Después del proceso de test, estos valores son leídos y listados por un programa escrito en C#. Además, este programa puede escribir la información en un archivo de texto.

La versión 0.8.5 de JaCoCo ya está integrada en la versión para desarrolladores de TESTAR, y no necesita ser instalada previamente fuera del directorio de TESTAR. Solamente necesita ser configurado el archivo *build.xml* de la carpeta *.../testar/target/install/testar/bin/jacoco* después de construir TESTAR desde la versión para desarrolladores. Se debe establecer un vínculo con el archivo *jar* específico del SUT que se testea, además de con los archivos sobre los que debe medirse la cobertura JaCoCo.

CAPÍTULO 6

Resultados

En esta sección se expondrán los resultados acerca de la cobertura temporal, así como el modo en el que se ha aplicado el test estadístico U de Mann-Whitley para aceptar o rechazar la hipótesis nula (mencionada en el apéndice 5.3).

En lo que respecta al progreso de la **cobertura de instrucciones y de ramas** para cada mecanismo de selección de acciones y cada SUT, los resultados pueden observarse en las gráficas del apéndice B.

6.1 Cobertura temporal

En la ejecución de los experimentos se han llevado a cabo un conjunto de mediciones con el objetivo de cuantificar las diferencias al usar diferentes mecanismos de selección de acciones en distintos SUTs. Después de cada acción, secuencia de acciones y ejecución se ha recopilado información acerca de la cobertura de instrucciones y ramas.

Se ha almacenado también información acerca del **tiempo** que ha llevado ejecutar 3000 acciones en cada SUT con cada mecanismo de selección de acciones. Para cada uno de estos casos, se ha calculado la media de los valores ofrecidos por cada una de las 30 máquinas virtuales. Los resultados son los siguientes:

	Random	Widget Group	Widget Difference	Image Difference
Rachota	17728628,1	18778964,3	20696481,97	21007078,3
Spaghetti	16050369,17	16372172,8	16304533,63	16472604,93
SwingSet2	20714630,63	21996697,57	21683716,23	22186915,83

De este modo, la ejecución de los tests en Rachota, usando el mecanismo de selección de acciones *random*, ha tardado un promedio de 17728628,1 segundos. Este valor se obtiene calculando la media del tiempo que le ha llevado a cada máquina virtual ejecutar los tests en Rachota con el mecanismo *random*.

En la tabla podemos observar que Spaghetti es el SUT en el que se invierte el menor tiempo a la hora de ejecutar los tests. En contraposición, SwingSet2 es el que se ha necesitado más tiempo.

En cuestión de mecanismos de selección de acciones, *image difference* es con el que se ha tardado más. Por otra parte, *random* es el que ha requerido el menor tiempo.

6.2 Características del test U

Para comprobar el impacto que ha tenido cada uno de estos mecanismos, se ha realizado el test estadístico U, de Mann-Whitley, tal y como se comentó en el capítulo 5.3.

Esto se ha hecho teniendo en cuenta, para cada caso, **dos grupos de datos**: el primero de ellos (grupo X) siempre corresponde a la cobertura al usar el mecanismo de selección aleatoria en las 30 máquinas virtuales, y el segundo grupo (grupo Y) representa los valores de cobertura asociados a los demás mecanismos. Por lo tanto, se compararán los datos del enfoque de selección aleatoria con los de los demás métodos de selección.

En primer lugar, se plantean las siguientes ideas:

1. Los dos grupos (X e Y) son observados de forma independiente
2. Las observaciones representan variables continuas u ordinales.
3. Si se asume la hipótesis nula, la distribución de los dos grupos es igual: P(X > Y) = P(Y > X).
4. Con la hipótesis alternativa, existe una tendencia a que los valores de una de las muestras excedan a los de la otra: P(X > Y) + 0.5 P(X = Y) > 0.5.

A continuación se lleva a cabo el cálculo del estadístico U. Para ello, se aplican las siguientes fórmulas:

$$U_1 = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1$$
$$U_2 = n_1 n_2 + \frac{n_2(n_2+1)}{2} - R_2$$

- n_1 es el tamaño de la primera muestra.
- n_2 es el tamaño de la segunda muestra.
- R_1 es la suma de la posición relativa de cada individuo que hay en la primera muestra.
- R_2 es la suma de la posición relativa de cada individuo que hay en la segunda muestra.

Teniendo estos datos, el estadístico U es el mínimo de $\{U_1, U_2\}$.

Además, z representa la aproximación a la distribución normal. Para su cálculo, anteriormente debemos calcular m_U y σ_U:

$$m_U = \frac{n_1 n_2}{2}$$
$$\sigma_U = \sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12}}$$

RACHOTA INSTRUCTION COVERAGE		
Virtual Machine	Random	Widget Group
1	41,52	47,01
2	50,75	46,6
3	44,7	46,88
4	54,91	45,24
5	55,89	44,83
6	55,89	55,39
7	41,18	45,77
8	38,17	44,41
9	53,26	54,67
10	53,13	57,33
11	44	56,79
12	44,45	48,78
13	34,08	45,4
14	45,63	55,61
15	49,24	47,47
16	41,18	45,58
17	39,12	56,17
18	44,67	45,39
19	42,58	53,91
20	57,24	52,77
21	41,46	54,62
22	48,73	58,52
23	46,55	55,53
24	49,38	56,44
25	51,02	47,74
26	54,46	47,34
27	44,2	55,39
28	45,55	55,43
29	55,16	56,63
30	44,04	44,82

Imagen 6.1: Tabla de Excel de la cobertura de instrucciones en Rachota con selección aleatoria y agrupación de widgets, al cabo de 200 acciones

- m_U es la media de U si la hipótesis nula es verdadera.
- σ_U es la desviación estándar de U si la hipótesis nula es verdadera.

Así pues, z se obtiene de la siguiente forma:

$$z = \frac{U - m_U}{\sigma_U}$$

Después, se debe consultar qué probabilidad tiene el valor de U respecto al promedio. Para ello, se consulta el valor de z en la tabla de probabilidades asociadas. El dato obtenido recibe el nombre de p. Si es menor que el valor de significancia 0.05, rechazaremos la hipótesis nula, expuesta con anterioridad en el capítulo 5.3:

H_0: No existe una gran diferencia en los valores de los parámetros medidos, aplicando distintos mecanismos de selección de acciones en TESTAR.

6.3 Aplicación del test U

Se ha hecho uso del test estadístico Mann-Whitley para aceptar o rechazar la hipótesis nula. Se han utilizado muestras de 30 datos, cada uno de los cuales corresponde a una máquina virtual. Estos valores representan la cobertura de ramas y de instrucciones que se ha alcanzado en tres casos distintos: tras la ejecución de 200, 300 y 3000 acciones.

Imagen 6.2: Cobertura de instrucciones en Rachota (200 acciones)

Uno de grupos de datos considerados es el que corresponde al mecanismo de selección aleatoria; el otro grupo a comparar representa el alguno de los enfoques propuestos en este trabajo. Los datos se han recopilado en tablas de Excel como en la Imagen 6.1.

La organización en este modelo de tabla de los datos obtenidos, se ha realizado comparando el enfoque de selección aleatoria con el de agrupación de widgets, con el de diferencia de widgets y con el de diferencia de imágenes. Además, se ha hecho realizado este procedimiento para cada uno de los SUTs y para cada tipo de cobertura a considerar (de instrucciones y de ramas).

En lo referente a la **cobertura de instrucciones**, en **Rachota** no es posible desechar la hipótesis nula considerando 3000 acciones, porque el valor de p no es inferior a la cifra de significancia 0,05:

- Comparando *Random* y *Widget Group*: 0,066762614.
- Comparando *Random* y *Widget Difference*: 0,267502703.
- Comparando *Random* e *Image Difference*: 0,976410908.

Tampoco se puede con 300 acciones, debido a que p es:

- Comparando *Random* y *Widget Group*: 0,097751055.
- Comparando *Random* y *Widget Difference*: 0,065671258.
- Comparando *Random* e *Image Difference*: 0,344044941.

En cambio, sí podemos descartar dicha hipótesis tomando 200 acciones:

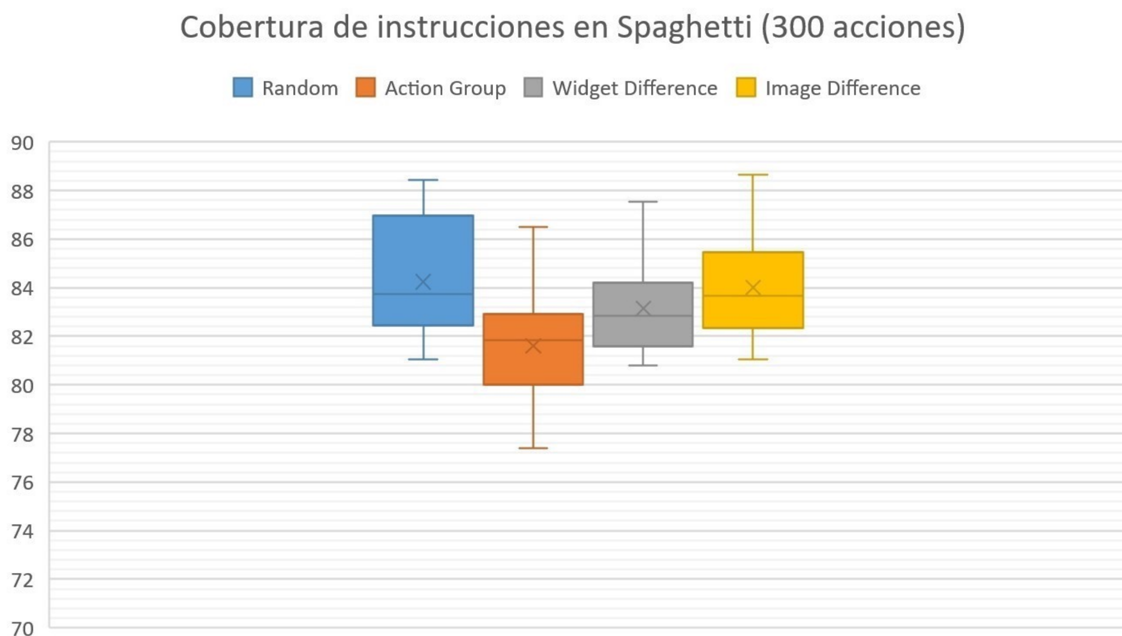

Imagen 6.3: Cobertura de instrucciones en Spaghetti (300 acciones)

- Comparando *Random* y *Widget Group*: *p* tiene el valor de 0,005201922.
- Comparando *Random* y *Widget Difference*: *p* es 0,000163229.
- Comparando *Random* e *Image Difference*: El valor de *p* es 0,007786348.

La representación en boxplots de la cobertura mencionada está en la Imagen 6.2.

Para **Spaghetti**, en cambio, los resultados reflejan que no existen diferencias de interés al usar los nuevos mecanismos de selección de acciones respecto al ya existente, para cualquier número de acciones considerado. La suma de la cobertura total usando selección aleatoria siempre es mayor a la de los demás enfoques. El gráfico para 300 acciones está en la Imagen 6.3.

En **SwingSet2**, se obtienen los mejores resultados al considerar 300 acciones. Un resumen puede verse en la Imagen 6.4.

- Comparando *Random* y *Widget Group*: La cobertura total de *Random* siempre es mayor a la de *Widget Group*.
- Comparando *Random* y *Widget Difference*: *p* es 0,023243447.
- Comparando *Random* e *Image Difference*: *p* tiene el valor de 0,015014133.

En relación a la **cobertura de ramas**, los datos sobre **Rachota** tienen el mejor impacto al tener en cuenta la ejecución de 200 acciones:

- Comparando *Random* y *Widget Group*: *p* tiene el valor de 0,01383162.
- Comparando *Random* y *Widget Difference*: *p* es 0,001003532.
- Comparando *Random* e *Image Difference*: El valor de *p* es 0,049260667.

Imagen 6.4: Cobertura de instrucciones en SwingSet2 (300 acciones)

Estos valores de p permiten rechazar la hipótesis nula H_0. La representación visual la podemos ver en la Imagen 6.5.

En **Spaghetti**, al igual que como se comentó respecto a su cobertura de instrucciones, no presenta avances. En todos los mecanismos de selección y en cada SUT, la cobertura del mecanismo de selección aleatoria es mayor a la cobertura de los demás. El gráfico en el que se consideran 300 acciones se ilustra en la Imagen 6.6.

En *SwingSet2*, los mejores resultados se reflejan al tener en cuenta 300 acciones. Sin embargo, no podemos rechazar la hipótesis nula:

- Comparando *Random* y *Widget Group*: La cobertura total de *Random* siempre es mayor a la de *Widget Group*.
- Comparando *Random* y *Widget Difference*: p tiene el valor de 0,283778048, por lo que no podemos descartar H_0.
- Comparando *Random* e *Image Difference*: p es 0,258051495, por lo que no podemos descartar H_0.

Podemos ver los resultados en forma de gráfica en la Imagen 6.7.

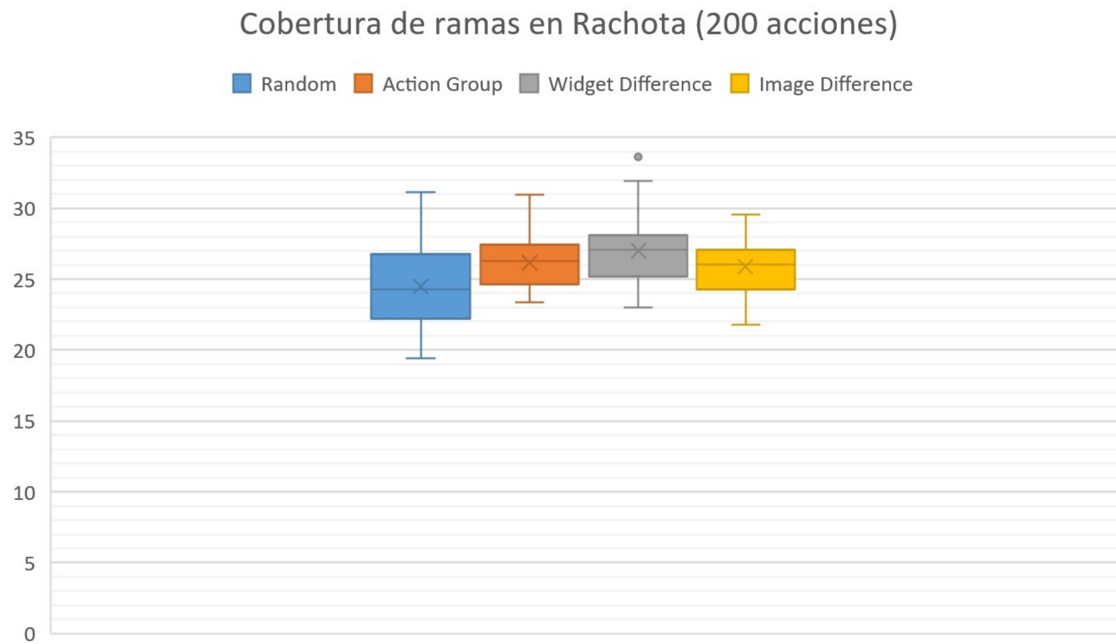

Imagen 6.5: Cobertura de ramas en Rachota (200 acciones)

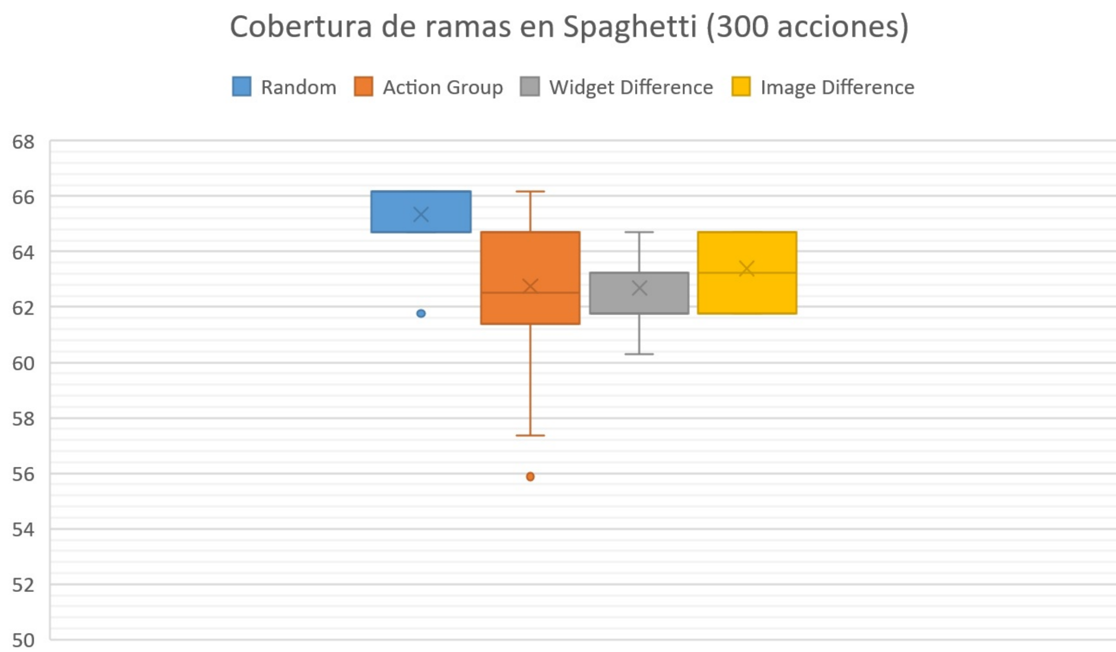

Imagen 6.6: Cobertura de ramas en Spaghetti (300 acciones)

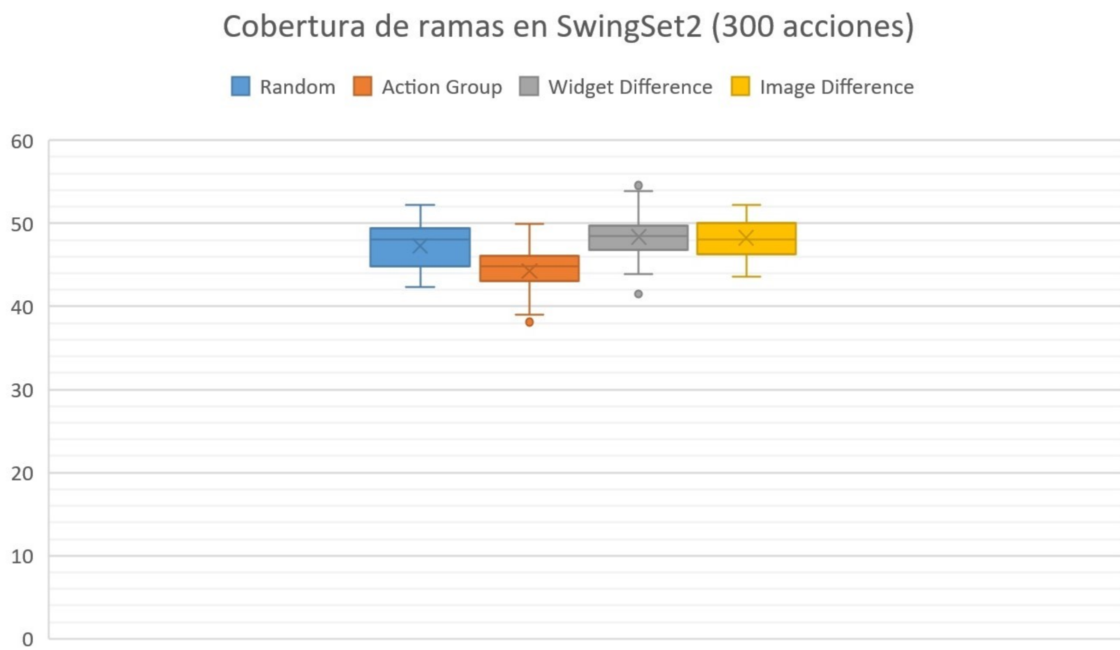

Imagen 6.7: Cobertura de ramas en SwingSet2 (300 acciones)

CAPÍTULO 7
Conclusiones

A partir de los datos recogidos, observamos que la cobertura temporal apenas varía al usar SUTs distintos. Es por ello que en este estudio no es un dato significativo. Sin embargo, generalmente han aparecido ventajas al utilizar los nuevos mecanismos de selección de acciones. A continuación se interpretarán los resultados para cada SUT.

La cobertura de instrucciones y ramas en los nuevos enfoques es más alta para aplicaciones como **Rachota**[6], sin demasiado exceso de widgets y con una estructura similar a la de la mayoría de aplicaciones que vemos día a día. Este hecho permite que el sistema de aprendizaje realice la selección de nuevas acciones de forma más fiable.

Para aplicaciones con menos funcionalidades como **Spaghetti**[7], en cambio, los resultados de los nuevos mecanismos no han sido mejores que en la selección aleatoria. Este tipo de programas tienen pocos tipos de widgets, y provocan poca interacción del sistema cuando se lleva a cabo una acción sobre ellos. Además, existe una carencia de niveles de profundidad, reduciendo la cantidad de acciones totales a ejecutar.

En **SwingSet2**[8] se ha detectado una ligera mejoría en los casos en los que se han aplicado los enfoques *Widget Difference* e *Image Difference*. Este tipo de aplicaciones son de mayor complejidad que las anteriores, y poseen widgets que no provocan apenas cambios en el programa al ejecutar una acción sobre ellos. Esta es la razón de que los resultados del enfoque *Action Group* hayan sido bajos. Al disminuir continuamente la probabilidad de ejecutar un determinado grupo de acciones en una aplicación compleja, la eficacia puede verse afectada.

7.1 Relación del trabajo desarrollado con los estudios cursados

En la realización del actual trabajo han influido distintos conceptos aprendidos en el Grado en Ingeniería Informática y también en el Máster Universitario en Ingeniería y Tecnología de Sistemas Software, ambas enseñanzas ofrecidas por la Universitat Politècnica de València.

Primero, el **ciclo de vida del desarrollo de software** ha sido el pilar fundamental. Este concepto, aprendido y puesto en práctica en multitud de asignaturas, ha presentado una estructura clara del proceso que se realiza para construir un producto software. Ha permitido entender e interiorizar la utilidad de cada una de las fases del desarrollo.

En segundo lugar, también ha contribuido a este trabajo la asignatura **Ingeniería del Software Experimental**, impartida en el máster anteriormente mencionado, en la que los alumnos aprendimos a evaluar la eficacia de nuevas técnicas respecto a a otras que ya existían. Esto se hacía de forma controlada, sistemática, cuantificable y disciplinada, para así determinar bajo qué criterios existían mejores resultados.

7.2 Trabajos futuros

En lo que respecta a al futuro, en primer lugar se investigarán formas de **mejorar los enfoques propuestos**, con el objetivo de que proporcionen resultados de cobertura todavía mayores.

También se hará una **búsqueda de nuevos mecanismos** de selección de acciones o posibles criterios a tener en cuenta, a fin de conseguir un mejor testing.

Por último, se aplicarán estas nuevas técnicas a **un mayor número de casos de uso**, como por ejemplo nuevos SUTs o nuevas tecnologías. De este modo podremos analizar el comportamiento en ámbitos no contemplados hasta el momento.

Bibliografía

[1] Action selection. https://en.wikipedia.org/wiki/Action_selection.

[2] Decoder web. https://www.decoder-project.eu/.

[3] Fittest web. http://www.pros.webs.upv.es/projects/fittest/.

[4] iv4xr web. https://iv4xr-project.eu/.

[5] Ivves web. https://ivves.weebly.com/.

[6] Rachota application. https://sourceforge.net/p/rachota/code/ci/default/tree/.

[7] Spaghetti application. https://github.com/TESTARtool/TESTAR_dev/tree/master_experiments/suts. [Application under the name TestGUI].

[8] Swingset2 application. https://github.com/TESTARtool/TESTAR_dev/tree/master_experiments/suts.

[9] Testar repository. https://github.com/TESTARtool/TESTAR_dev.

[10] Testar web. https://testar.org/.

[11] Testomat web. https://www.testomatproject.eu/.

[12] Uia windows. https://docs.microsoft.com/en-us/windows/win32/winauto/uiauto-entry-propids.

[13] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce. Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 2–8, New York, NY, USA, 2018. Association for Computing Machinery.

[14] P. Aho, T. E. J. Vos, S. Ahonen, T. Piirainen, P. Moilanen, and F. Pastor Ricos. Continuous piloting of an open source test automation tool in an industrial environment. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 1–4. Sistedes, 2019.

[15] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery.

[16] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 219–230, New York, NY, USA, 2010. Association for Computing Machinery.

[17] S. Bauersfeld, A. de Rojas, and T. E. J. Vos. Evaluating rogue user testing in industry: An experience report. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–10, May 2014.

[18] S. Bauersfeld, T. E. J. Vos, N. Condori-Fernández, A. Bagnato, and E. Brosse. Evaluating the TESTAR tool in an industrial case study. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 4, 2014.

[19] W. Booth, G. Colomb, J. Williams, J. Bizup, and W. FitzGerald. *The Craft of Research, Fourth Edition*. Chicago Guides to Writing, Editing, and Publishing. University of Chicago Press, 2016.

[20] C. Cao, H. Ge, T. Gu, J. Deng, P. Yu, and J. Lu. Accelerating automated android gui exploration with widgets grouping. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 268–277, 2018.

[21] H. Chahim, M. Duran, T. E. J. Vos, P. Aho, and N. Condori Fernandez. Scriptless testing at the gui level in an industrial setting. In F. Dalpiaz, J. Zdravkovic, and P. Loucopoulos, editors, *Research Challenges in Information Science*, pages 267–284, Cham, 2020. Springer International Publishing.

[22] A. Esparcia-Alcázar, F. Almenar, M. Martínez, U. Rueda, and T. E. J. Vos. Q-learning strategies for action selection in the testar automated testing tool. 2016.

[23] O. Lysne. *Software Quality and Quality Management*, pages 87–98. 02 2018.

[24] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 81–90, 2012.

[25] M. Martinez, A. I. Esparcia, U. Rueda, T. E. J. Vos, and C. Ortega. Automated localisation testing in industry with testar. In F. Wotawa, M. Nica, and N. Kushik, editors, *Testing Software and Systems*, pages 241–248, Cham, 2016. Springer International Publishing.

[26] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[27] J. A. Rice. *Mathematical statistics and data analysis*. Nelson Education, 2006.

[28] F. P. Ricós, P. Aho, T. Vos, I. T. Boigues, E. C. Blasco, and H. M. Martínez. Deploying testar to enable remote testing in an industrial ci pipeline: A case-based evaluation. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, pages 543–557, Cham, 2020. Springer International Publishing.

[29] M. Rouncefield. Combinations, probabilities and sample size. investigations into the mann–whitney u test. *Teaching Mathematics and Its Applications: International Journal of the IMA*, 17(4):159–161, 1998.

[30] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[31] T. E. Vos, P. Aho, F. P. Ricos, O. R. Valdes, and A. Mulders. Testar – scriptless testing through graphical user interface. *Software Testing, Verification and Reliability, Special issue on new generations of UI testing*, n/a(n/a):e1770.

[32] T. E. J. Vos and A. I. Esparcia. Software testing innovation alliance - the SHIP project -. In *Joint Proceedings of the Doctoral Symposium and Projects Showcase Held as Part of STAF 2016 co-located with Software Technologies: Applications and Foundations (STAF 2016), Vienna, Austria, July 4-7, 2016*, pages 65–71, 2016.

[33] T. E. J. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener. TESTAR: Tool support for test automation at the user interface level. *Int. J. Inf. Syst. Model. Des.*, 6(3):46–83, July 2015.

[34] T. E. J. Vos, P. Tonella, J. Wegener, M. Harman, W. Prasetya, E. Puoskari, and Y. Nir–Buchbinder. Future internet testing with fittest. In *15th European Conference on Software Maintenance and Reengineering*, pages 355–358, March 2011.

[35] D. P. Watkins, C.J.C.H. Q-learning. page 279–292. Mach Learn 8, 1992.

[36] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

Glosario

TFM: Trabajo de Fin de Máster

GUI: Graphical User Interface

SUT: System Under Test

IA: Inteligencia Artificial

RL: Reinforcement Learning

UML: Unified Modeling Language

CRUD: Create, Read, Update and Delete

SGBD: Sistema de Gestión de Base de Datos

APÉNDICE A

Test settings de los SUTs utilizados

Para llevar a cabo tests sobre los distintos SUTs, se han recogido una serie de parámetros que conforman las diferentes configuraciones que se deben usar. Las siguientes tablas contienen la información referente a cada uno de los SUTs: Rachota, Spaghetti y SwingSet2. En cada una de ellas aparecen las expresiones *settings_ag*, *settings_wd* y *settings_id*, para agrupar las configuraciones de selección de acciones para las propuestas *action_group*, *widget_difference* e *image_difference*, respectivamente.

Bajo el nombre de ***settings_ag*** se agrupan las siguientes configuraciones:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardActionGroup
- QFunction = QFunctionActionGroup
- AbstractStateAttributes = WidgetPath, WidgetControlType

Por otro lado, ***settings_wd*** conforma:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardWidgetDifference
- QFunction = QFunctionWidgetDifference
- AbstractStateAttributes = WidgetPath, WidgetControlType

Para finalizar, en ***settings_id*** se engloba:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardImageDifference
- QFunction = QFunctionImageDifference
- AbstractStateAttributes = WidgetPath, WidgetControlType

<table>
<tr><th colspan="6">Test settings de Rachota</th></tr>
<tr><th colspan="2">Enfoque</th><th>purerandom</th><th>action_group</th><th>widget_diff</th><th>image_diff</th></tr>
<tr><td rowspan="2">Accessibility API</td><td>Java Protocol</td><td colspan="4">RachotaProtocol extends JavaSwingProtocol</td></tr>
<tr><td>Settings</td><td colspan="4">JavaAccessBridge</td></tr>
<tr><td>WinProcess.
JavaExecution</td><td>Java Protocol</td><td colspan="4">TESTAR ejecutará el SUT con Java.
Es necesario para añadir parámetros JMX correctamente
(-Dcom.sun.management.jmxremote.port=5000).</td></tr>
<tr><td>AccessBridgeFetcher.
swingJavaTableDescend</td><td>Java Protocol</td><td colspan="4">Se inspeccionarán las celdas internas de las tablas Java Swing.</td></tr>
<tr><td>SuspiciousTitle</td><td>Settings</td><td colspan="4">COVERAGE: Empty.
Deshabilitado, porque nos centramos en la medición de coverage y no en buscar errores.</td></tr>
<tr><td>ProcessListener</td><td>Settings</td><td colspan="4">COVERAGE: False.
Deshabilitado, porque nos centramos en la medición de coverage y no en buscar errores.</td></tr>
<tr><td>ClickFilter</td><td>Settings</td><td colspan="4">.*[cC]errar.* | .*[cC]lose.* | .*[sS]alir.* | .*[eE]xit.* | .*[mM]inimizar.* | .*[mM]inimi[zs]e.* | .*[mM]aximizar.* | .*[mM]aximi[zs]e.* | .*[gG]uardar.* | .*[sS]ave.* | .*[iI]mprimir.* | .*[pP]rint.* | .*[bB]rowse.* | .*[pP]revious.* | Inactivity detection | Interaction</td></tr>
<tr><td>Derive Actions</td><td>Java Protocol</td><td colspan="4">- Top widgets > All widgets
- (Trigger) addFilenameReportAction
- (Trigger) forcePricePerHourAndFinish
- (Add) isEditToClickWidget
- (Filter) isEditableWidget
- (Add) isCalendarTextWidget
- (Add) isSpinBoxWidget
- (Filter) isDurationTableCell
- (Add) forceWidgetTreeClickAction
- (Add) forceListElemetsClickAction
- (Filter) "Cancel" inside Report wizard</td></tr>
<tr><td>Extra features</td><td>Java Protocol</td><td colspan="4">- Create default "settigs.cfg" disabling detectedInactivity feature
- Delete ".rachota" config file at the end of execution
- (Oracle) Freeze issue - Empty Actions on State</td></tr>
<tr><td rowspan="2">Action Selection</td><td>Java Protocol</td><td rowspan="2">El protocolo devolverá una acción aleatoria.</td><td colspan="3">Se usa el State Model, llamando internamente a ReinforcementLearningModelManager. ReinforcementLearningActionSelector se inicializa con la política que existe en los settings (GreedyPolicy).</td></tr>
<tr><td>Settings</td><td>settings_ag</td><td>settings_wd</td><td>settings_id</td></tr>
</table>

Tabla A.1: *Test settings* de Rachota

En lo que respecta a Spaghetti, ***settings_ag*** agrupa:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardActionGroup
- QFunction = QFunctionActionGroup
- AbstractStateAttributes = WidgetPath, WidgetControlType, WidgetTitle

En ***settings_wd*** se representan las siguientes configuraciones:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardWidgetDifference
- QFunction = QFunctionWidgetDifference
- AbstractStateAttributes = WidgetPath, WidgetControlType, WidgetTitle

Además, ***settings_id*** agrupa:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardImageDifference
- QFunction = QFunctionImageDifference
- AbstractStateAttributes = WidgetPath, WidgetControlType, WidgetTitle

<table>
<tr><th colspan="6">Test settings de Spaghetti</th></tr>
<tr><th colspan="2">Enfoque</th><th>purerandom</th><th>action_group</th><th>widget_diff</th><th>image_diff</th></tr>
<tr><td rowspan="2">Accessibility API</td><td>Java Protocol</td><td colspan="4">SpaghettiProtocol extends JavaSwingProtocol</td></tr>
<tr><td>Settings</td><td colspan="4">JavaAccessBridge</td></tr>
<tr><td>WinProcess.
JavaExecution</td><td>Java Protocol</td><td colspan="4">TESTAR ejecutará el SUT con Java.
Es necesario para añadir parámetros JMX correctamente
(-Dcom.sun.management.jmxremote.port=5000).</td></tr>
<tr><td>AccessBridgeFetcher.
swingJavaTableDescend</td><td>Java Protocol</td><td colspan="4">No se inspeccionarán las celdas internas de las tablas Java Swing.</td></tr>
<tr><td>SuspiciousTitle</td><td>Settings</td><td colspan="4">COVERAGE: Empty.
Deshabilitado, porque nos centramos en la medición de coverage y no en buscar errores.</td></tr>
<tr><td>ProcessListener</td><td>Settings</td><td colspan="4">COVERAGE: False.
Deshabilitado, porque nos centramos en la medición de coverage y no en buscar errores.</td></tr>
<tr><td>ClickFilter</td><td>Settings</td><td colspan="4">.*[sS]istema.* | .*[sS]ystem.* | .*[cC]errar.* | .*[cC]lose.* |
.*[sS]alir.* | .*[eE]xit.* | .*[mM]inimizar.* | .*[mM]inimi[zs]e.* |
.*[mM]aximi[zs]e.* | .*[gG]uardar.* | .*[sS]ave.* |
.*[iI]mprimir.* | .*[pP]rint.* | FILE</td></tr>
<tr><td>Derive Actions</td><td>Java Protocol</td><td colspan="4">- Top widgets J16> All widgets
- (Add) Force Tree, ComboBox, List</td></tr>
<tr><td rowspan="2">Action Selection</td><td>Java Protocol</td><td rowspan="2">El protocolo devolverá una acción aleatoria.</td><td colspan="3">Se usa el State Model, llamando internamente a ReinforcementLearningModelManager. ReinforcementLearningActionSelector se inicializa con la política que existe en los settings (GreedyPolicy).</td></tr>
<tr><td>Settings</td><td>settings_ag</td><td>settings_wd</td><td>settings_id</td></tr>
</table>

Tabla A.2: *Test settings* de Spaghetti

Por último, en SwingSet2, ***settings_ag*** reúne:

- MaxQValue = 1

- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardActionGroup
- QFunction = QFunctionActionGroup
- AbstractStateAttributes = WidgetPath, WidgetControlType, WidgetTitle

Bajo el nombre ***settings_wd*** se engloba:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardWidgetDifference
- QFunction = QFunctionWidgetDifference
- AbstractStateAttributes = WidgetPath, WidgetControlType, WidgetTitle

Y ***settings_id*** hace referencia a:

- MaxQValue = 1
- Alpha = 1
- Gamma = 0.99
- Policy = GreedyPolicy
- RewardFunction = RewardImageDifference
- QFunction = QFunctionImageDifference
- AbstractStateAttributes = WidgetPath, WidgetControlType, WidgetTitle

<table>
<tr><td colspan="6">Test settings de SwingSet2</td></tr>
<tr><td colspan="2">Enfoque</td><td>purerandom</td><td>action_group</td><td>widget_diff</td><td>image_diff</td></tr>
<tr><td rowspan="2">Accessibility API</td><td>Java Protocol</td><td colspan="4">SwingSet2Protocol extends JavaSwingProtocol</td></tr>
<tr><td>Settings</td><td colspan="4">JavaAccessBridge</td></tr>
<tr><td>WinProcess.
JavaExecution</td><td>Java Protocol</td><td colspan="4">TESTAR ejecutará el SUT con Java.
Es necesario para añadir parámetros JMX correctamente
(-Dcom.sun.management.jmxremote.port=5000).</td></tr>
<tr><td>AccessBridgeFetcher.
swingJavaTableDescend</td><td>Java Protocol</td><td colspan="4">No se inspeccionarán las celdas internas de las tablas Java Swing.</td></tr>
<tr><td>SuspiciousTitle</td><td>Settings</td><td colspan="4">COVERAGE: Empty.
Deshabilitado, porque nos centramos en la medición de coverage y no en buscar errores.</td></tr>
<tr><td>ProcessListener</td><td>Settings</td><td colspan="4">COVERAGE: False.
Deshabilitado, porque nos centramos en la medición de coverage y no en buscar errores.</td></tr>
<tr><td>ClickFilter</td><td>Settings</td><td colspan="4">.*[sS]istema.* | .*[sS]ystem.* | .*[cC]errar.* | .*[cC]lose.* | .*[sS]alir.* | .*[eE]xit.* | .*[mM]inimizar.* | .*[mM]inimi[zs]e.* | .*[mM]aximi[zs]e.* | .*[gG]uardar.* | .*[sS]ave.* | .*[iI]mprimir.* | .*[pP]rint.* | .*JFileChooser.*</td></tr>
<tr><td>Derive Actions</td><td>Java Protocol</td><td colspan="4">- Top widgets > All widgets
- (Filter) isSourceCodeEditWidget
- (Add) forceWidgetTreeClickAction
- (Add) forceListElemetsClickAction</td></tr>
<tr><td rowspan="2">Action Selection</td><td>Java Protocol</td><td rowspan="2">El protocolo devolverá una acción aleatoria.</td><td colspan="3">Se usa el State Model, llamando internamente a ReinforcementLearningModelManager. ReinforcementLearningActionSelector se inicializa con la política que existe en los settings (GreedyPolicy).</td></tr>
<tr><td>Settings</td><td>settings_ag</td><td>settings_wd</td><td>settings_id</td></tr>
</table>

Tabla A.3: *Test settings* de SwingSet2

APÉNDICE B

Progreso de la cobertura de ramas e instrucciones

B.1 Cobertura de instrucciones con selección aleatoria

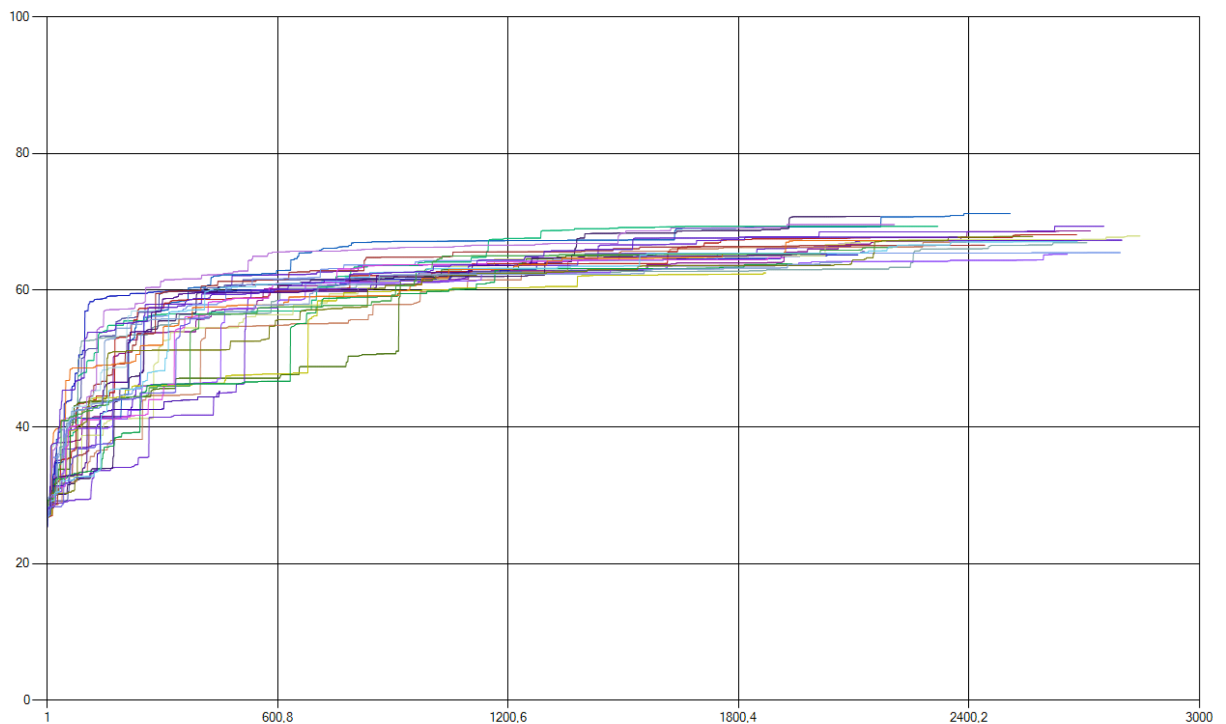

Imagen B.1: Progreso de la cobertura de instrucciones en Rachota usando selección aleatoria

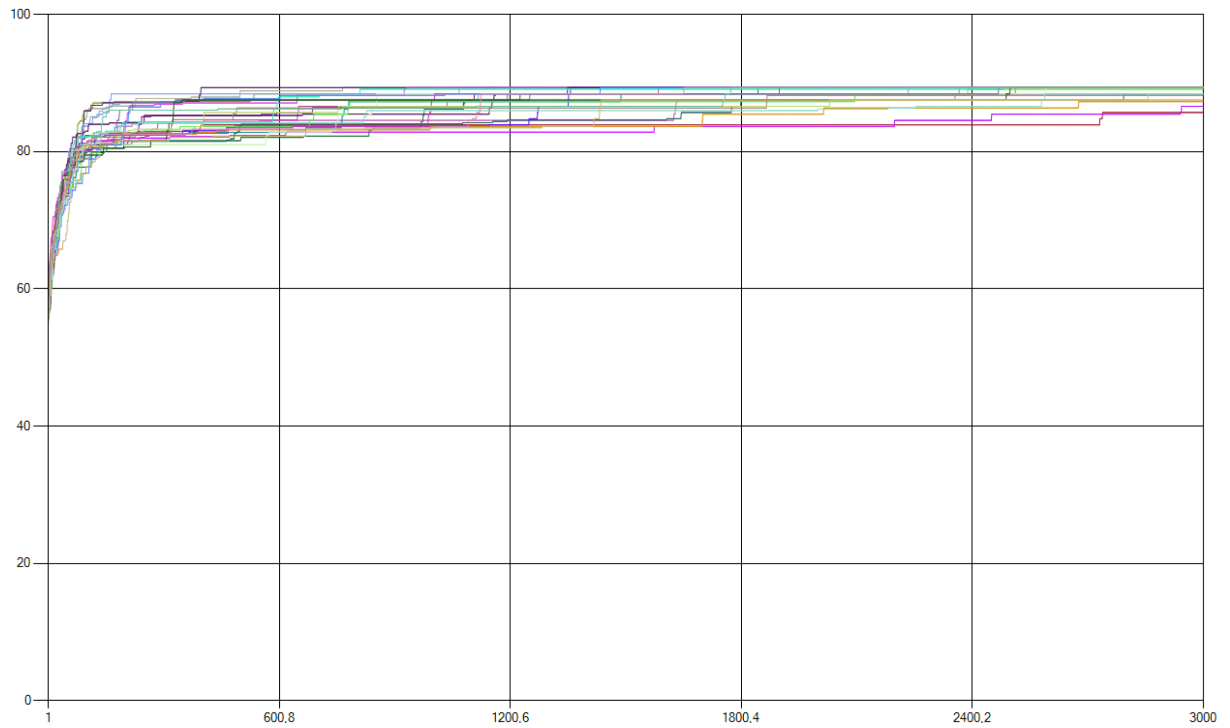

Imagen B.2: Progreso de la cobertura de instrucciones en Spaghetti usando selección aleatoria

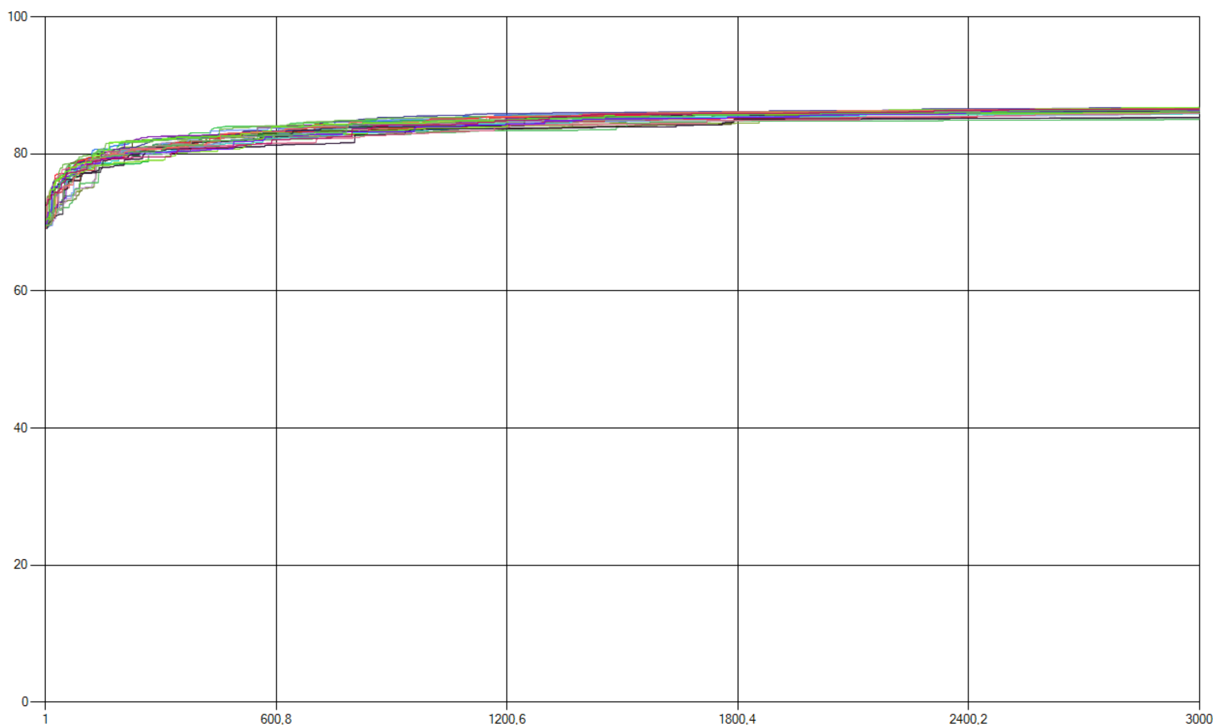

Imagen B.3: Progreso de la cobertura de instrucciones en SwingSet2 usando selección aleatoria

B.2 Cobertura de instrucciones con agrupación de widgets

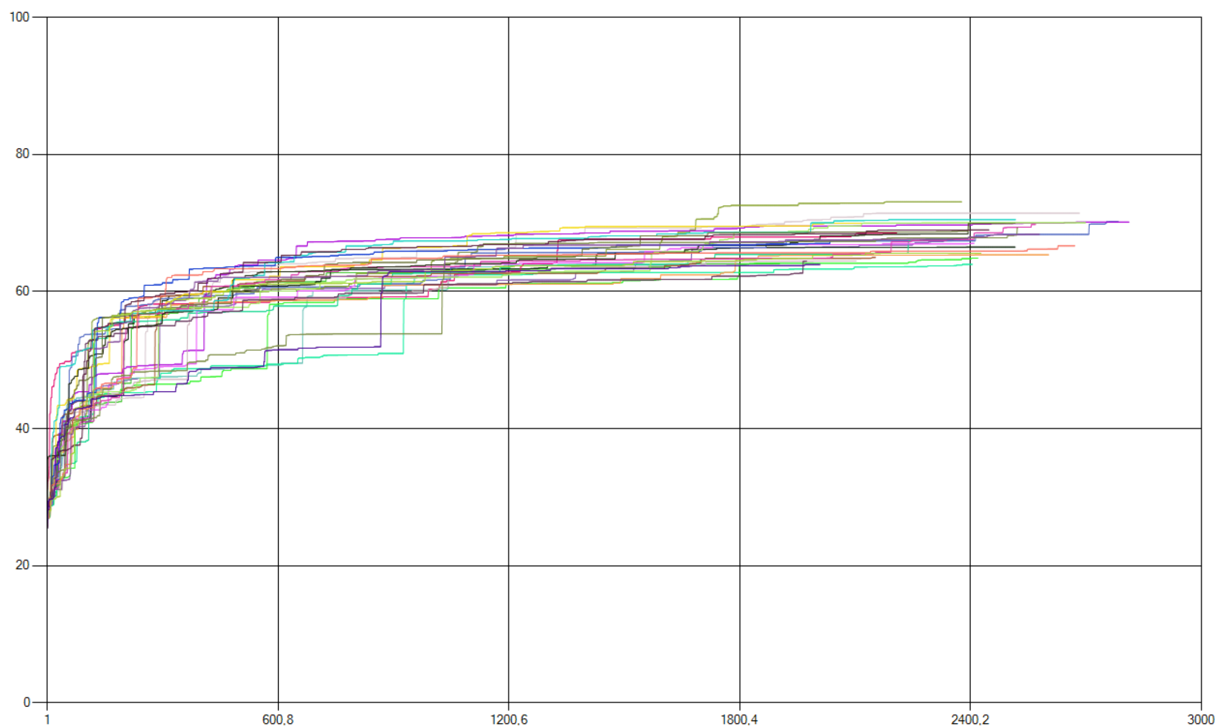

Imagen B.4: Progreso de la cobertura de instrucciones en Rachota usando agrupación de widgets

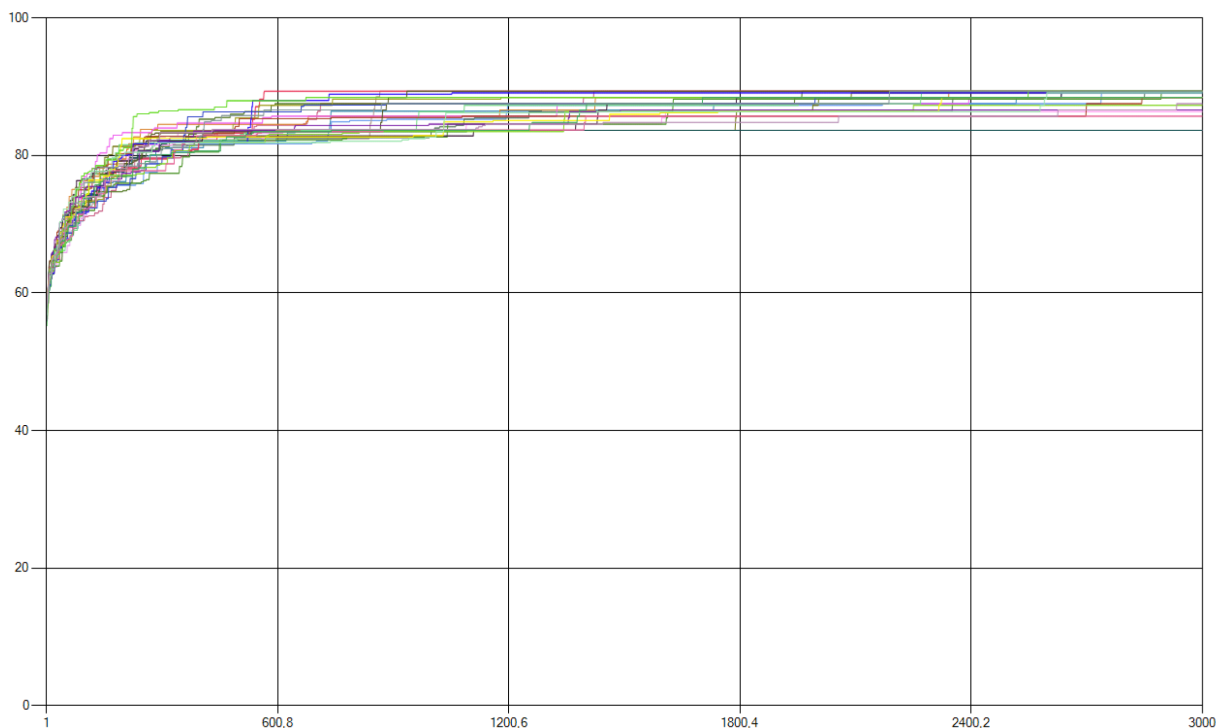

Imagen B.5: Progreso de la cobertura de instrucciones en Spaghetti usando agrupación de widgets

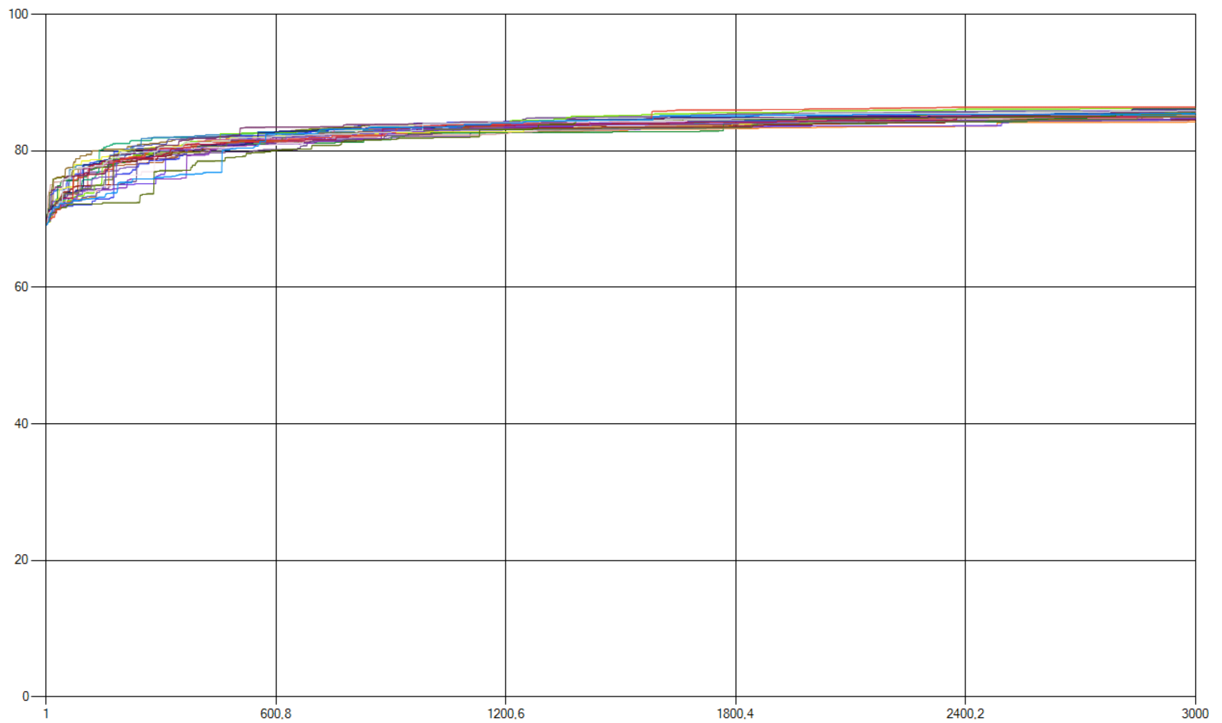

Imagen B.6: Progreso de la cobertura de instrucciones en SwingSet2 usando agrupación de widgets

B.3 Cobertura de instrucciones con diferencia de widgets

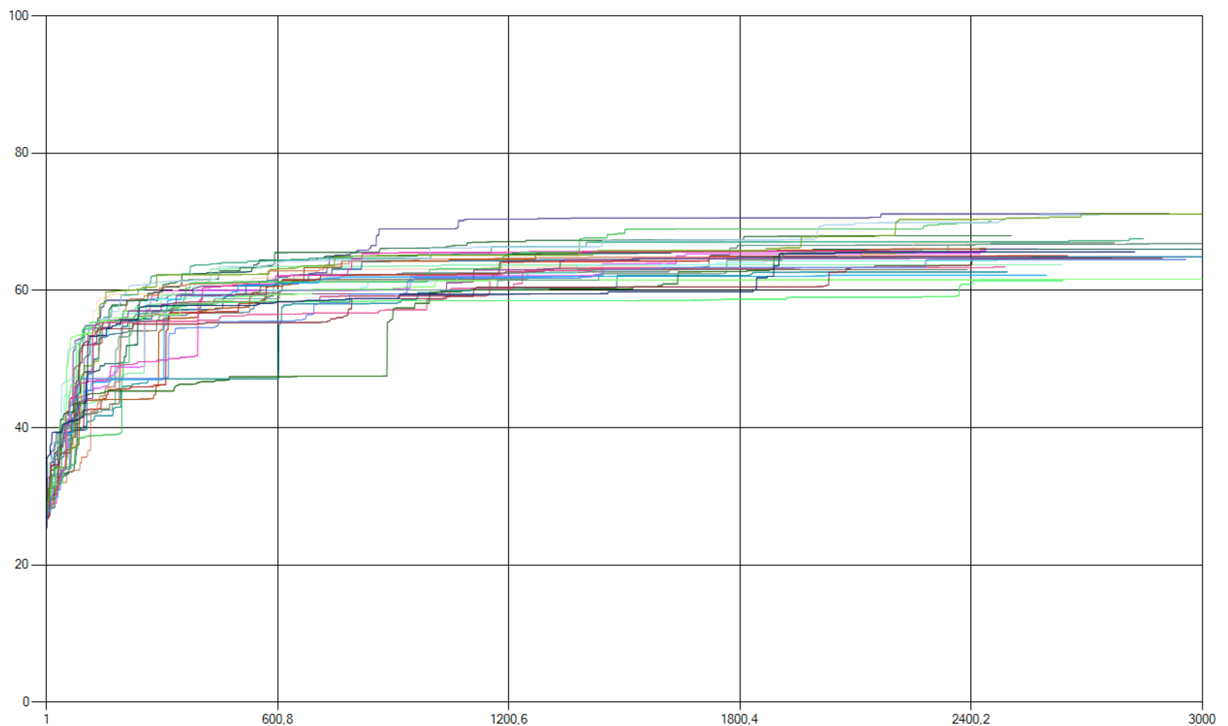

Imagen B.7: Progreso de la cobertura de instrucciones en Rachota usando diferencia de widgets

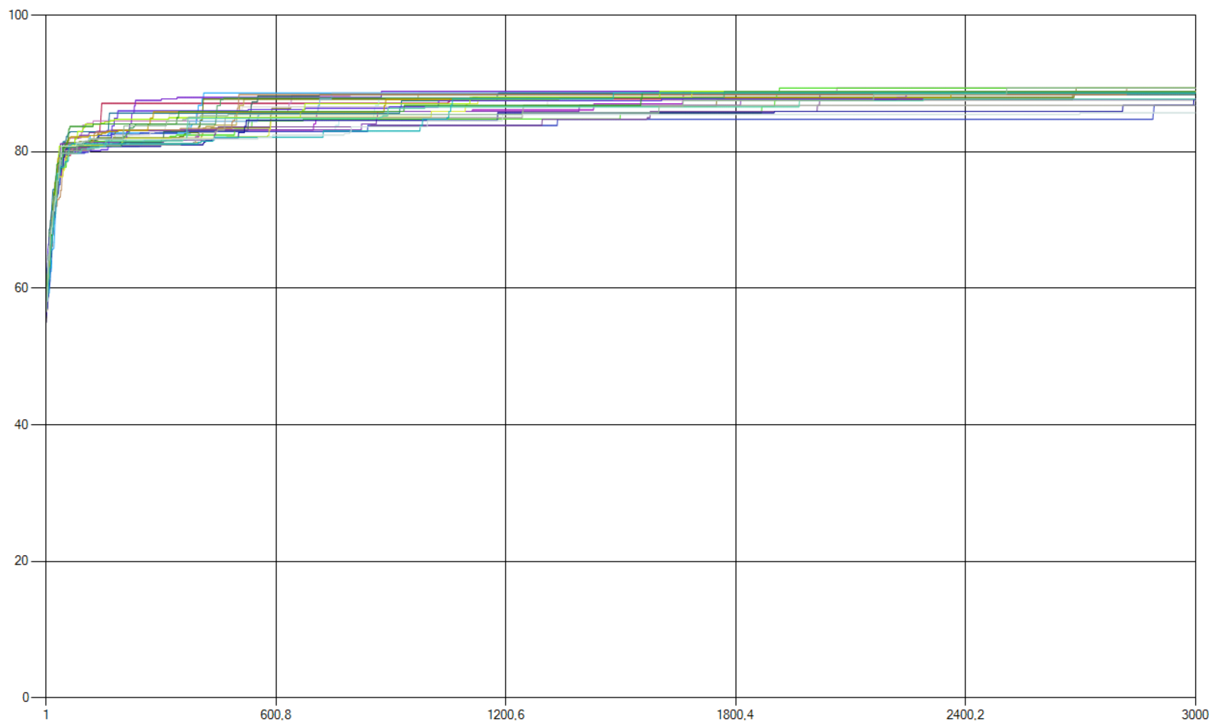

Imagen B.8: Progreso de la cobertura de instrucciones en Spaghetti usando diferencia de widgets

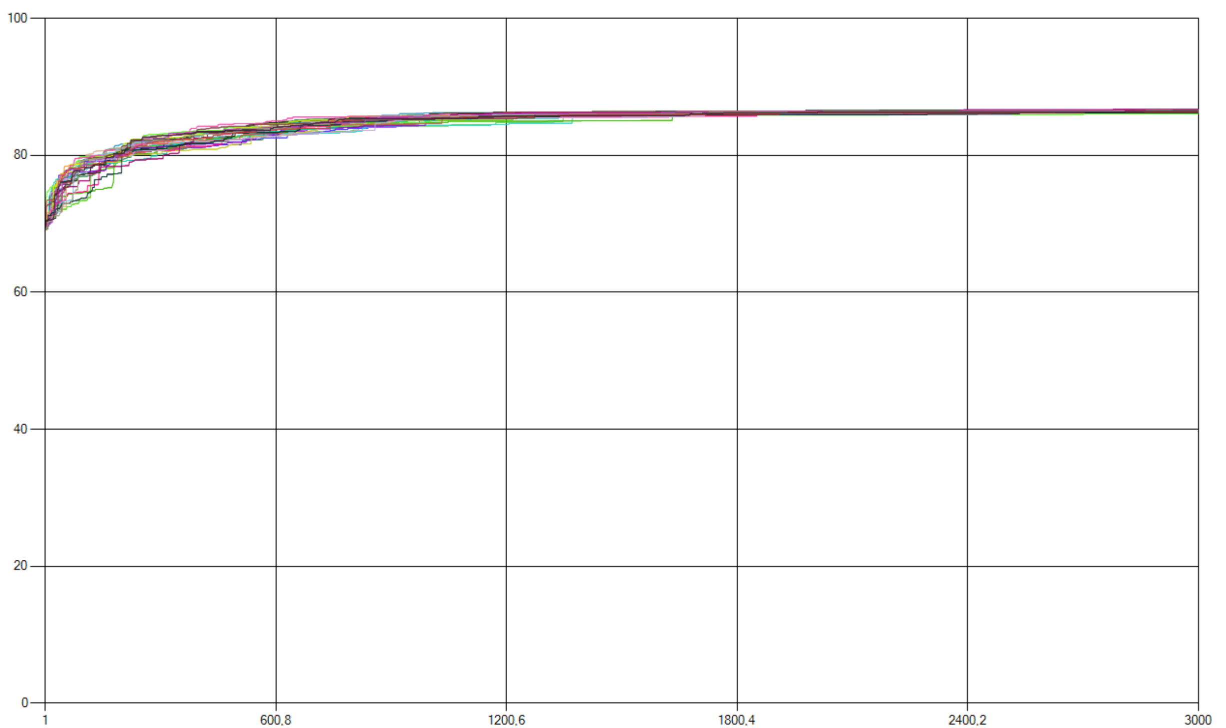

Imagen B.9: Progreso de la cobertura de instrucciones en SwingSet2 usando diferencia de widgets

B.4 Cobertura de instrucciones con diferencia de imágenes

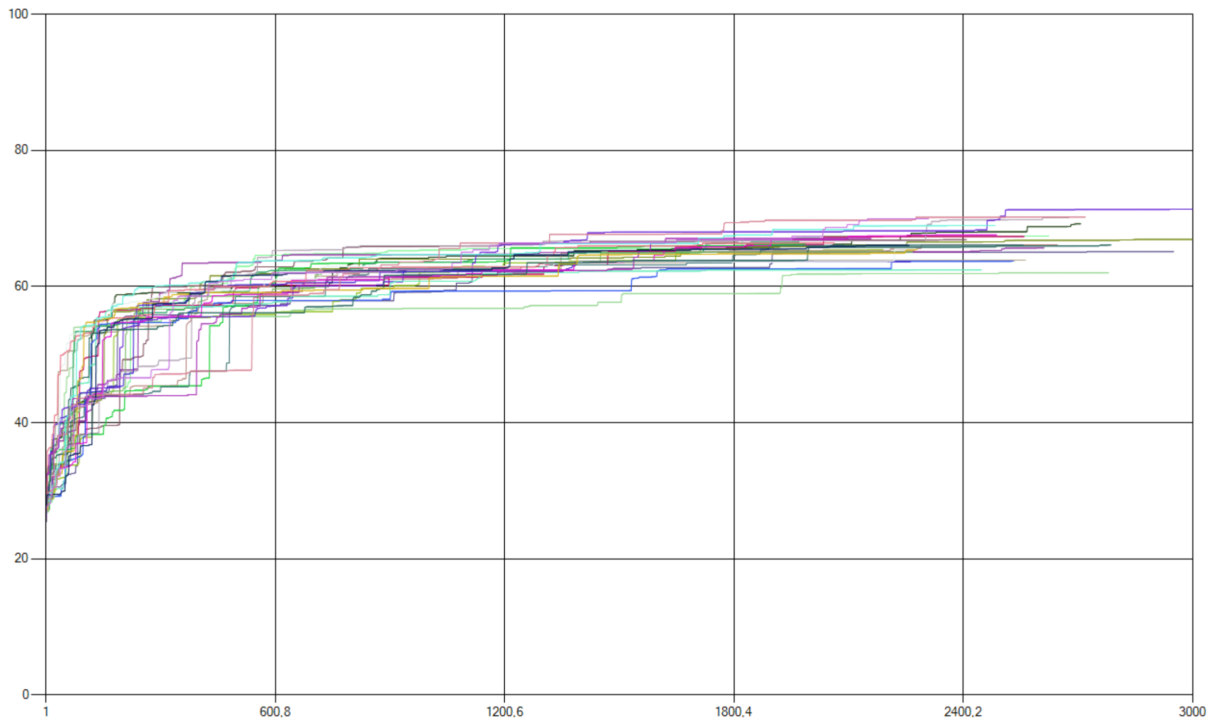

Imagen B.10: Progreso de la cobertura de instrucciones en Rachota usando diferencia de imágenes

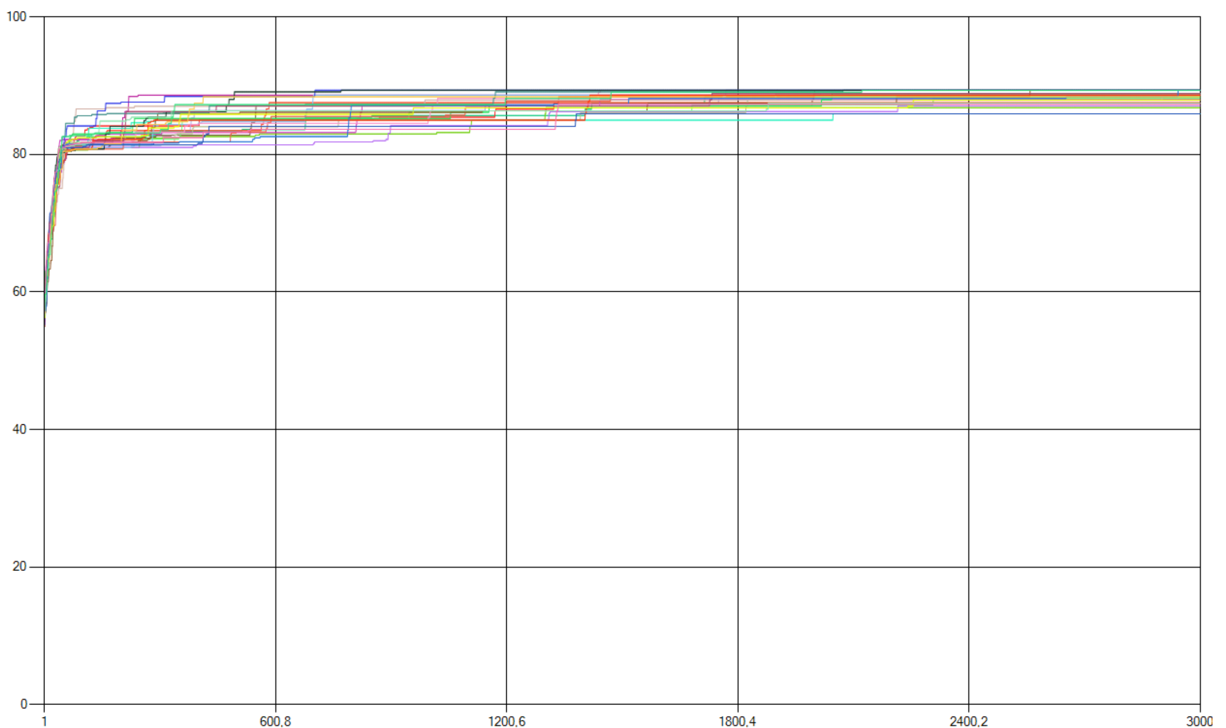

Imagen B.11: Progreso de la cobertura de instrucciones en Spaghetti usando diferencia de imágenes

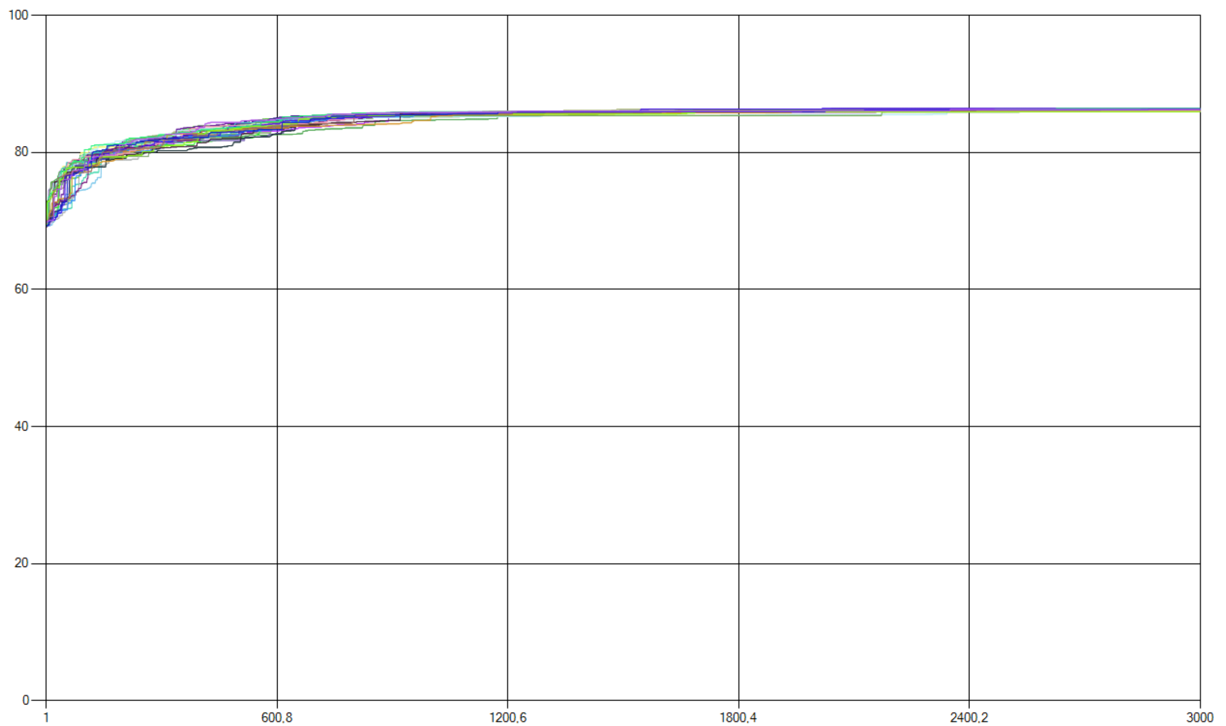

Imagen B.12: Progreso de la cobertura de instrucciones en SwingSet2 usando diferencia de imágenes

B.5 Cobertura de ramas con selección aleatoria

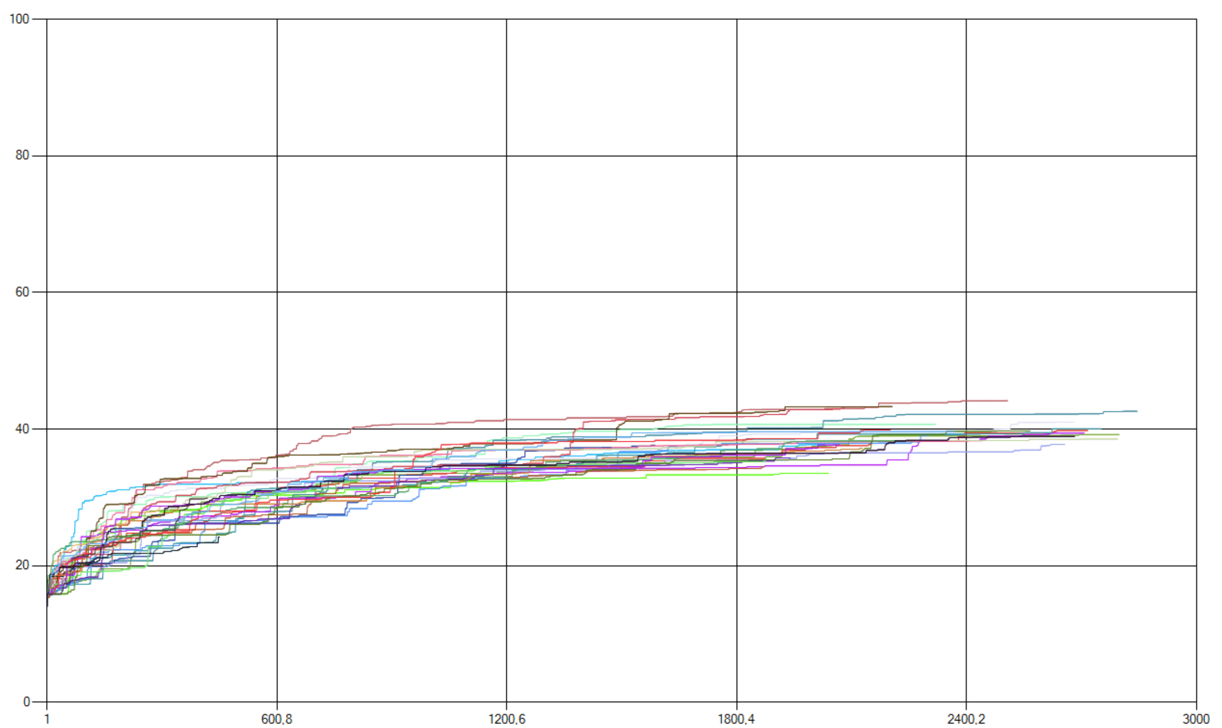

Imagen B.13: Progreso de la cobertura de ramas en Rachota usando selección aleatoria

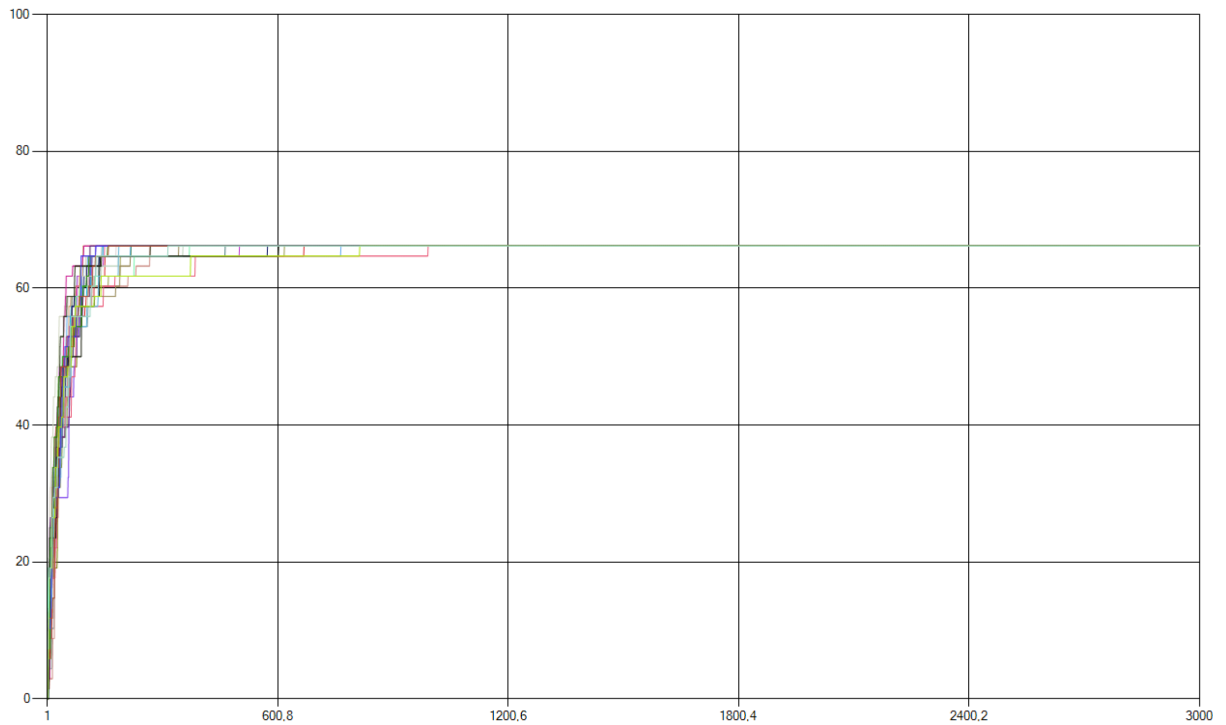

Imagen B.14: Progreso de la cobertura de ramas en Spaghetti usando selección aleatoria

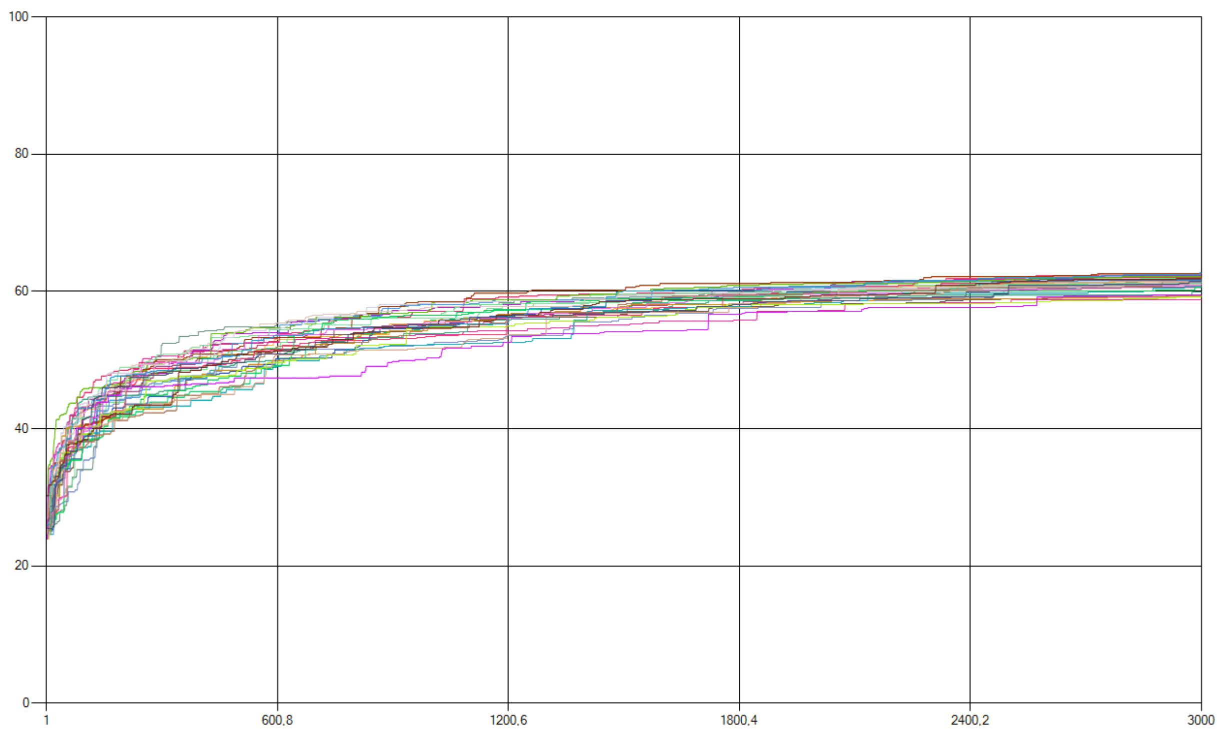

Imagen B.15: Progreso de la cobertura de ramas en SwingSet2 usando selección aleatoria

B.6 Cobertura de ramas con agrupación de widgets

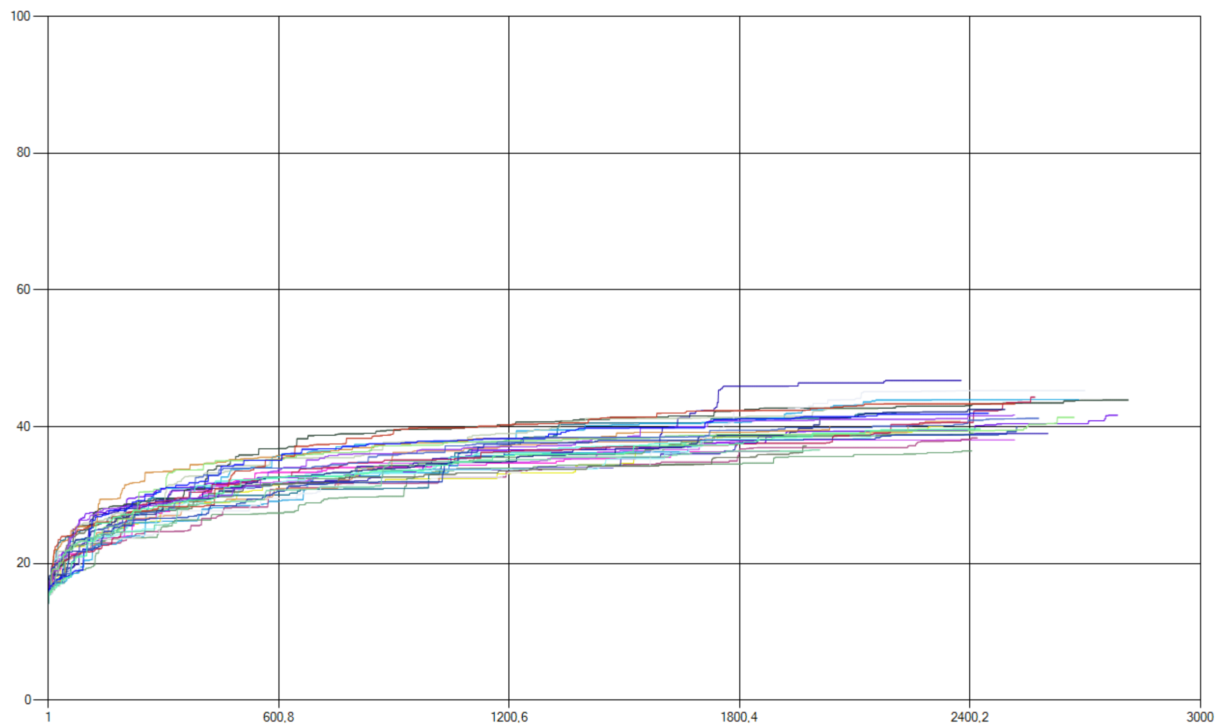

Imagen B.16: Progreso de la cobertura de ramas en Rachota usando agrupación de widgets

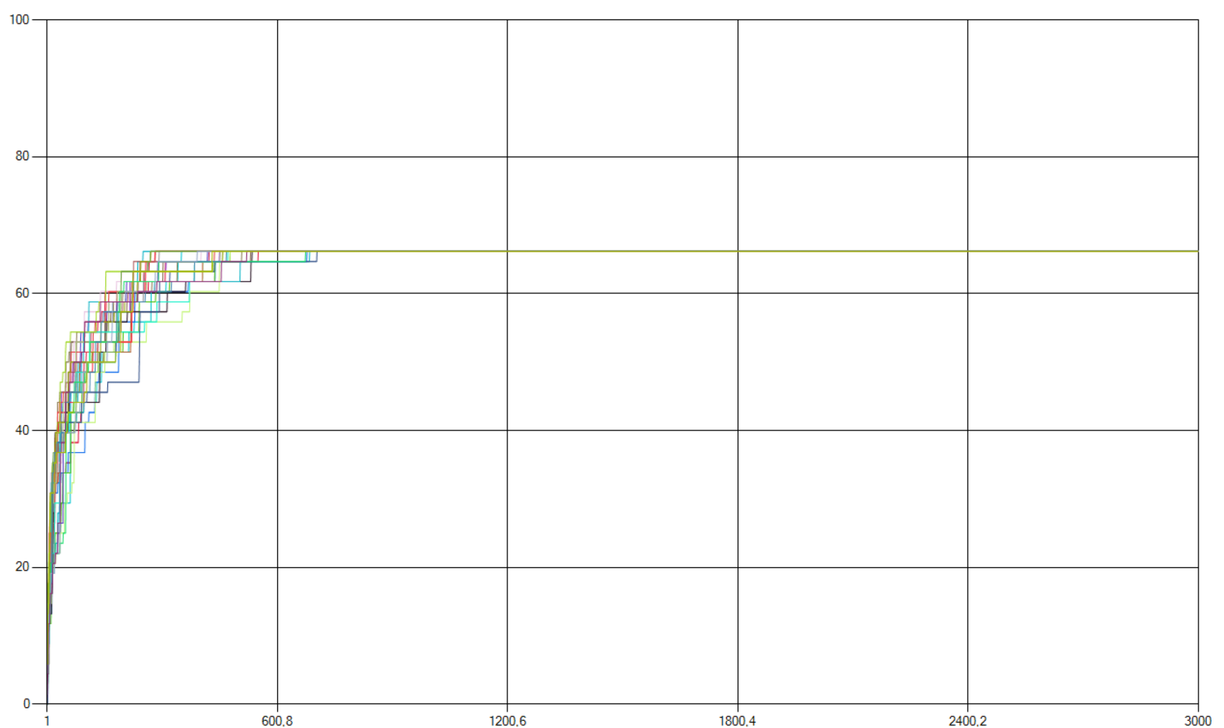

Imagen B.17: Progreso de la cobertura de ramas en Spaghetti usando agrupación de widgets

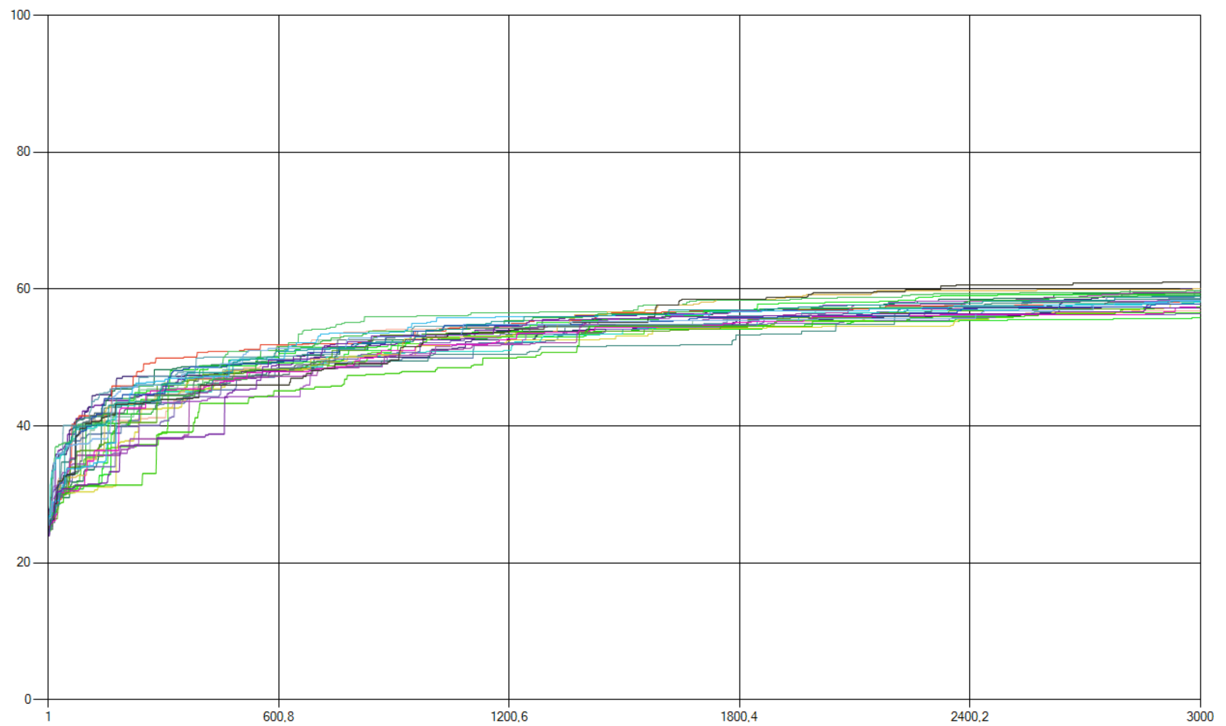

Imagen B.18: Progreso de la cobertura de ramas en SwingSet2 usando agrupación de widgets

B.7 Cobertura de ramas con diferencia de widgets

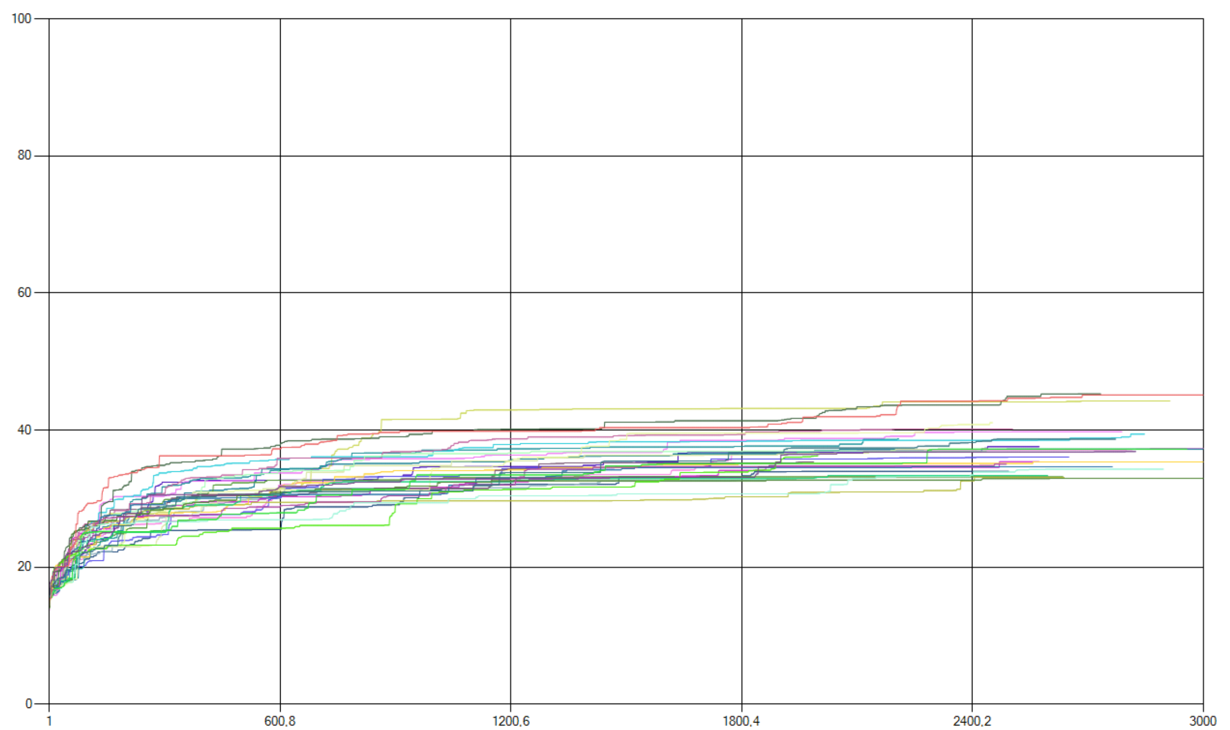

Imagen B.19: Progreso de la cobertura de ramas en Rachota usando diferencia de widgets

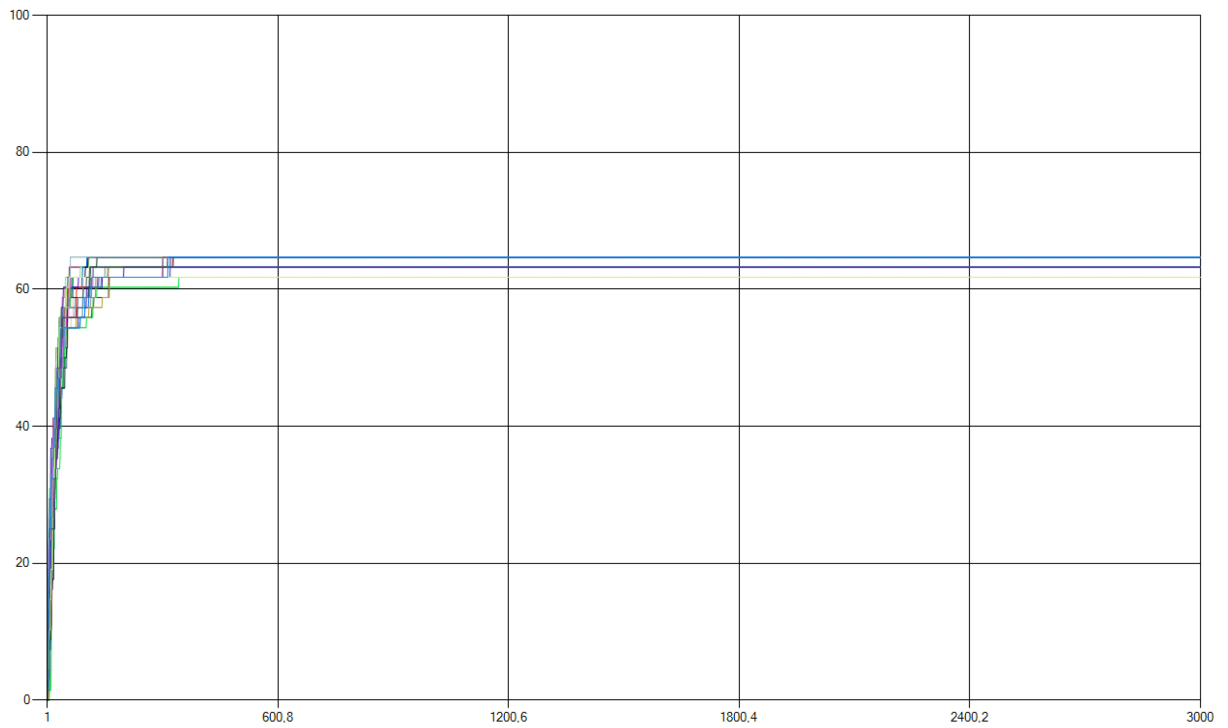

Imagen B.20: Progreso de la cobertura de ramas en Spaghetti usando diferencia de widgets

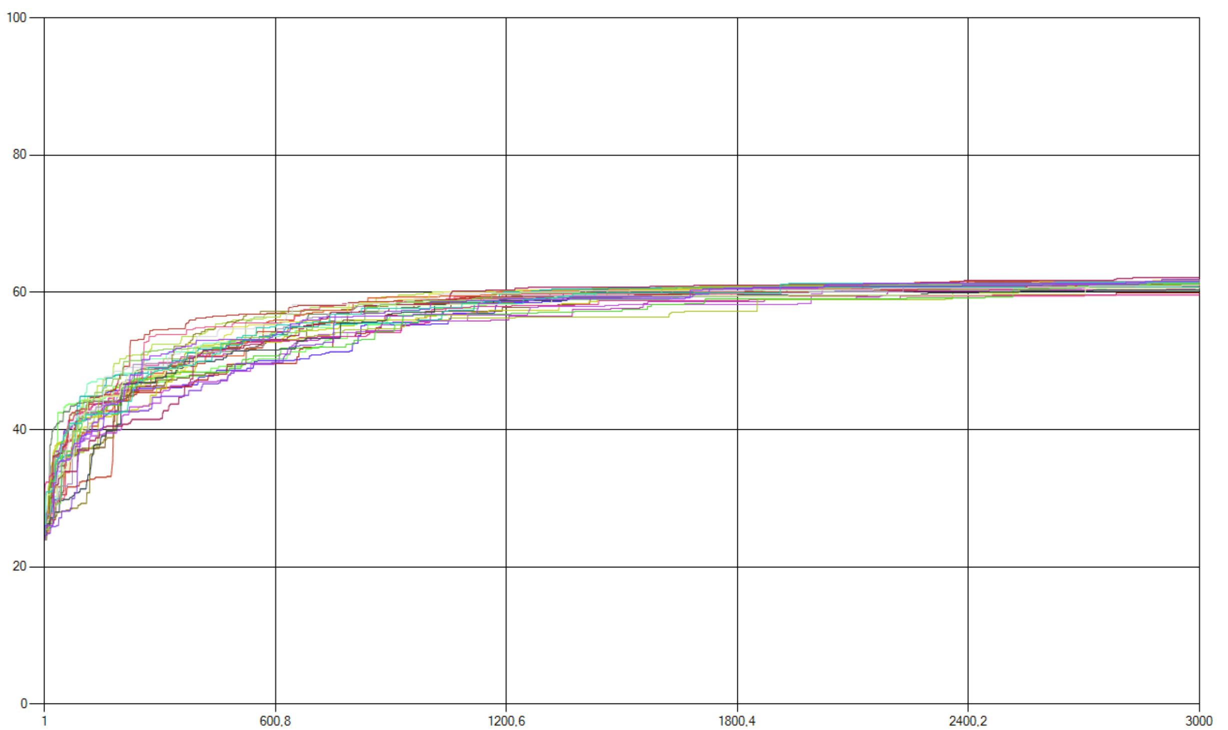

Imagen B.21: Progreso de la cobertura de ramas en SwingSet2 usando diferencia de widgets

B.8 Cobertura de ramas con diferencia de imágenes

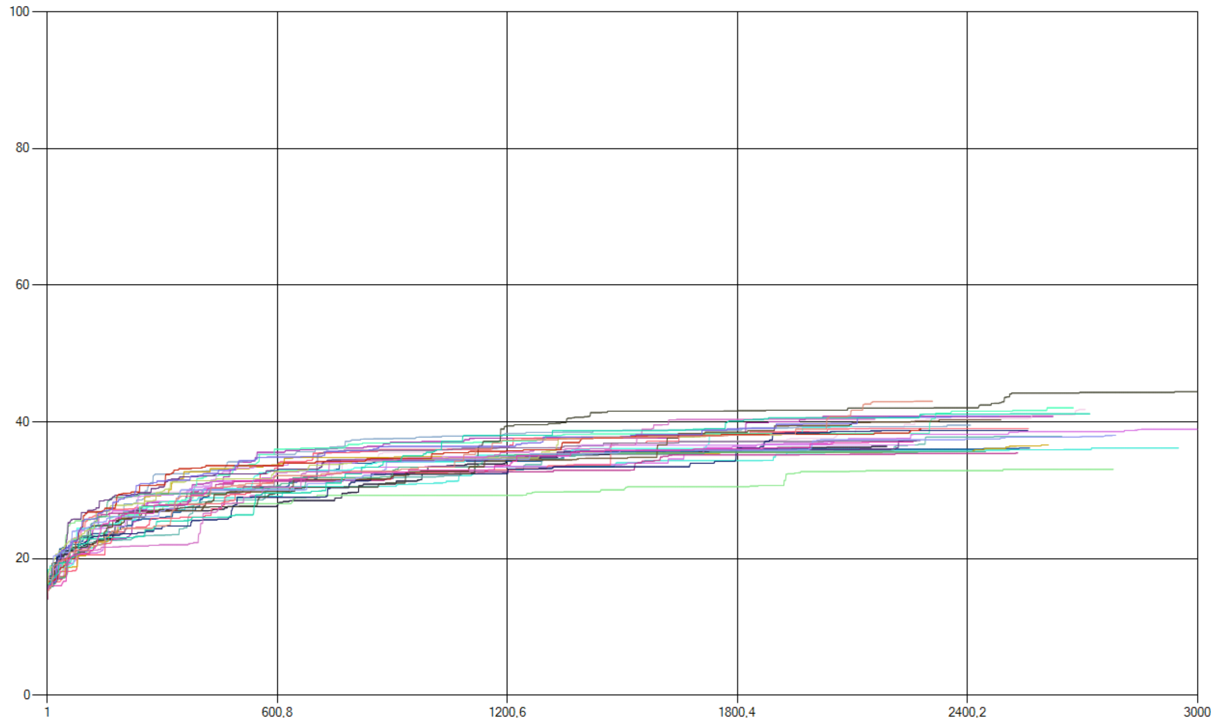

Imagen B.22: Progreso de la cobertura de ramas en Rachota usando diferencia de imágenes

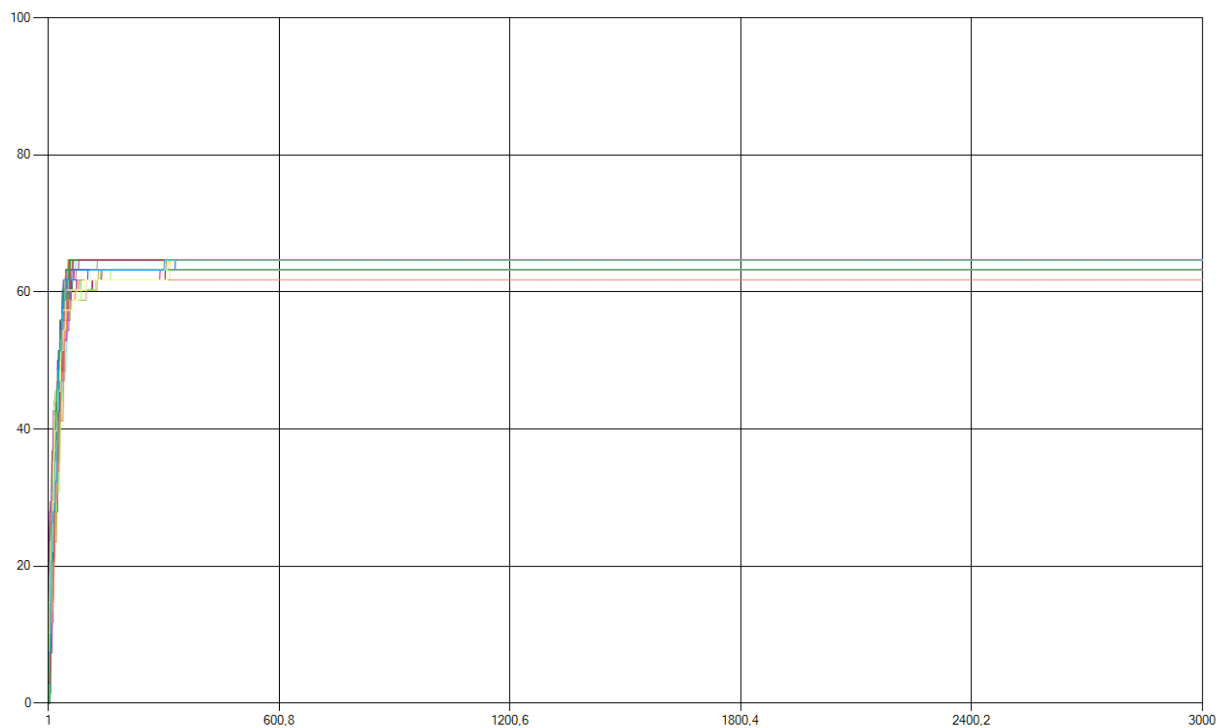

Imagen B.23: Progreso de la cobertura de ramas en Spaghetti usando diferencia de imágenes

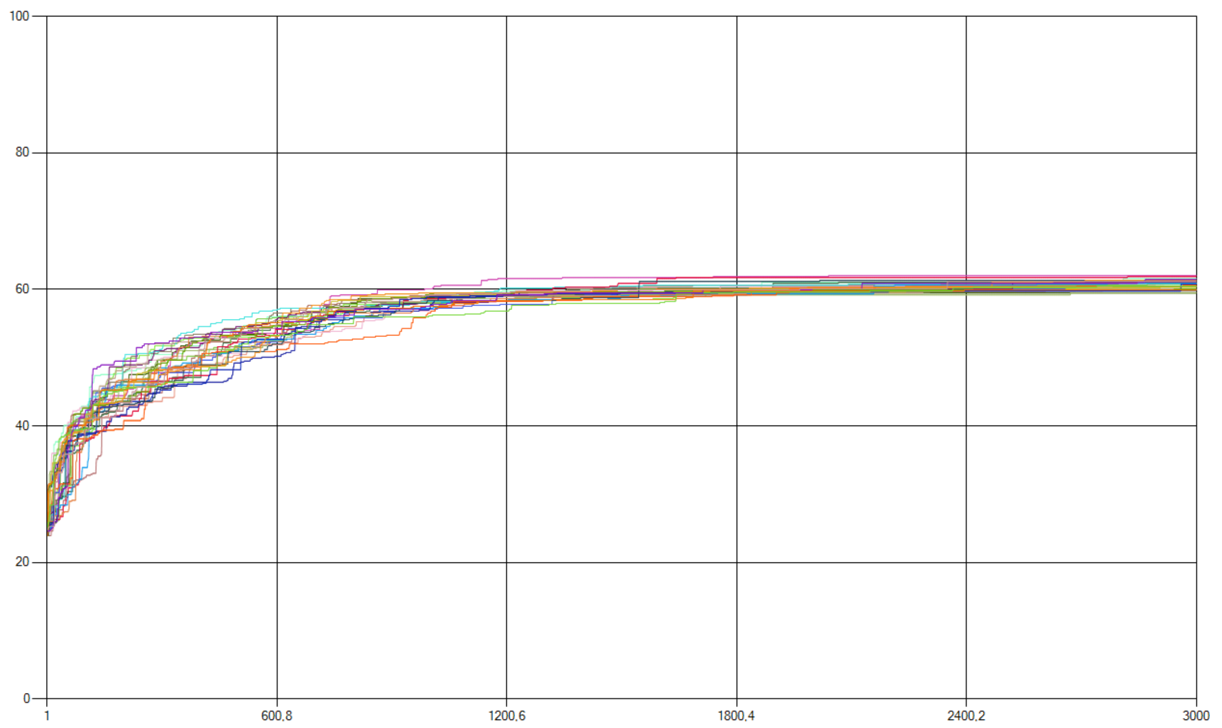

Imagen B.24: Progreso de la cobertura de ramas en SwingSet2 usando diferencia de imágenes