Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Validación Automática de Contratos Software

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autora: Orengo Faus, Sandra Maria

Tutoras: Alpuente Frasnedo, María
Villanueva García, Alicia

Curso 2019-2020

Resumen

En Ingeniería de Software, el concepto de contrato está relacionado con una descripción del comportamiento de los programas utilizando precondiciones, postcondiciones e invariantes. El estado del arte actual permite generar automáticamente contratos a partir del código fuente que pueden ser usados como entrada para analizadores cada vez más potentes. Sin embargo, los contratos generados automáticamente pueden no ser completamente precisos o correctos, conteniendo algunos elementos que no están verificados.

El objetivo de este proyecto es desarrollar una aplicación que permita refinar dichos contratos software. Mediante la utilización del generador automático de casos de prueba KLEE se identificará y eliminará aquellos componentes que el proceso de validación determina que son demostradamente falsos. El trabajo consiste en una herramienta software que, a partir de un programa C y de las restricciones representadas por su contrato software asociado (generado automáticamente) y mediante el uso de la herramienta KLEE, proporciona soporte automático a la generación de casos de prueba con los que poder detectar y eliminar partes del contrato que son demostradamente incorrectas.

Palabras clave: Contratos software, Generación automática de casos de prueba, Validación de contratos, KLEE, Ejecución Simbólica

Resum

En Enginyeria de Software, el concepte de contracte està relacionat amb una descripció del comportament dels programes utilitzant precondicions, postcondicions i invariants. L'estat de l'art actual permet generar automàticament contractes a partir del codi font que poden ser usats com a entrada per a analitzadors cada vegada més potents. No obstant això, els contractes generats automàticament poden no ser completament precisos o correctes, contenint alguns elements que no estan verificats.

L'objectiu d'aquest projecte és desenvolupar una aplicació que permeta refinar aquests contractes software. Mitjançant l'utilització del generador automàtic de casos de prova KLEE s'identificarà i eliminarà aquells components que el procés de validació determina que són demostradament falsos. El treball consisteix en una eina software que, a partir d'un programa C i de les restriccions representades pel seu contracte software associat (generat automàticament) i mitjançant l'ús de l'eina KLEE, proporciona suport automàtic a la generació de casos de prova amb els quals poder detectar i eliminar parts del contracte que són demostradament incorrectes.

Paraules clau: Contractes software, Generació automàtica de casos de prova, Validació de contractes, KLEE, Execució Simbòlica

Abstract

In Software Engineering, the concept of contract is related to a description of the behavior of programs using preconditions, postconditions and invariants. The current state of the art allows to automatically generate contracts from source code that can be used as input for increasingly more powerful analysers. However, automatically generated contracts may not be completely accurate or correct, containing some elements that are not verified.

The objective of this project is to develop an application that allows to refine these software contracts. Using the automatic test case generator KLEE we will identify and eliminate those components that the validation process determines to be demonstrably false. The work consists of a software tool that, based on a C program and the restrictions represented by its associated software contract (automatically generated) and through the use of the KLEE tool, provides automatic support for the generation of test cases with those that can detect and eliminate parts of the contract that are demonstrably incorrect.

Key words: Software contracts, Automatic generation of test cases, Contract validation, KLEE, Symbolic Execution

Índice general

Índice general v
Índice de figuras vii
Índice de tablas ix
Índice de fragmentos de código x

1 Introducción 1
1.1 Motivación . . . 3
1.2 Objetivos . . . 3
1.3 Impacto Esperado . . . 4
1.4 Metodología y planificación . . . 5
1.5 Estructura . . . 9
2 Contexto tecnológico 11
2.1 KindSpec . . . 11
2.1.1 Método de inferencia . . . 12
2.2 $\mathbb{K}$: A Semantic Framework . . . 16
2.3 Symbolic Java Pathfinder . . . 17
2.4 Pex . . . 18
3 KLEE: Generación automática de pruebas de alta cobertura para software complejo 19
3.1 Ejecución simbólica . . . 19
3.2 KLEE . . . 20
3.2.1 Introducción . . . 20
3.2.2 Instalación . . . 20
3.2.3 Preparación . . . 21
3.2.4 Generación de casos de prueba . . . 23
3.2.5 Ficheros generados . . . 24
3.2.6 Herramientas auxiliares . . . 26
3.2.7 Ejecución de los casos de prueba . . . 28
3.3 El uso de KLEE para la refutación de contratos software . . . 29
4 Análisis del problema 39
4.1 Análisis de requisitos . . . 39
4.1.1 Requisitos funcionales . . . 39
4.1.2 Requisitos no funcionales . . . 44
4.2 Identificación y análisis de soluciones posibles . . . 44
4.3 Solución propuesta . . . 45
5 Diseño de la solución 47
5.1 Arquitectura del Sistema . . . 47
5.2 Diseño Detallado . . . 48
5.3 Tecnologías Utilizadas . . . 51

6 Desarrollo de la solución **53**
6.1 Desarrollo de la interfaz de línea de comandos 54
6.2 Desarrollo de la funcionalidad 55
6.2.1 Método compilar(String) 55
6.2.2 Método ejecutar(String) 56
6.2.3 Método leerTest() 56
6.2.4 Método validarArchivo(File) 57
6.3 Desarrollo del almacenamiento 57
7 Validación de la solución propuesta **59**
7.1 Validación con el archivo **get_sign()** 59
7.2 Validación con el archivo **insert()** 61
8 Conclusiones **63**
Bibliografía **65**

Apéndices
A Código de insert() **69**
B Instalación y uso de la aplicación **71**
B.1 Instalación 71
B.2 Uso 72
B.2.1 Comando *Add* 72
B.2.2 Comando *Run* 72
B.2.3 Comando *Check* 73
B.2.4 Comando *Help* 73

Índice de figuras

1.1 Puesta en marcha tablero Kanban. 5
1.2 Detalle de una UT. 5
1.3 Priorización de tareas Sprint 1. 6
1.4 Representación de la mitad del Sprint 1. 6
1.5 Inicio del Sprint 2. 7
1.6 Aspecto de la UT Añadir archivo en el Sprint 1. 7
1.7 Aspecto de la UT Añadir archivo modificada en el Sprint 2. 8
1.8 Representación del tablero con la tarea reabierta en el Sprint 2. . . . 8

2.1 Axiomas de poscondición esperados en el método **insert(s,x)** [2]. . . 15
2.2 Arquitectura del *framework* $\mathbb{K}$ [20]. 16
2.3 Esquema de Symbolic Java Pathfinder [15]. 17

3.1 Ejecución simbólica. 20
3.2 Salida por terminal de la generación de casos de prueba. 24
3.3 Salida por terminal del fichero de texto info. 25
3.4 Ejemplo de la herramienta ktest-tool. 27
3.5 Uso de la librería **libkleeRuntest**. 28
3.6 Primer axioma candidato del método **insert()**. 30
3.7 Salida por consola de la ejecución del archivo compilado mediante KLEE. 31
3.8 Lectura del único caso generado para **insert.c** y la precondición $isnull(s) = 1$. 31
3.9 Ejecución del caso generado. 32
3.10 Primer axioma candidato del método **insert()**. 32
3.11 Casos de prueba generados si el elemento no está contenido en la estructura. 34
3.12 Ejecución de los casos de prueba si el elemento no está contenido en la estructura. 34
3.13 Ejecución de los casos de prueba si el elemento no está contenido en la estructura. 35
3.14 Ficheros ejecutados por KLEE. 35
3.15 Resultados si se contiene el elemento parte 1. 36
3.16 Resultados si se contiene el elemento parte 2. 36
3.17 Resultados si se contiene el elemento parte 3. 36
3.18 Resultados si se contiene el elemento parte 4. 36

4.1 Diagrama de casos de uso de la herramienta desarrollada. 40

5.1 Diagrama de Arquitectura por capas. 47
5.2 Implementación del patrón *Singleton*. 48

5.3 Esquema global de nuestra aplicación. 49
5.4 Diagrama de clases de la capa de presentación. 49
5.5 Diagrama de clases de la capa de lógica. 50
5.6 Diagrama de clases de la capa de datos. 50

6.1 Detalle de la estructura de *ContractFalsifier*. 53
6.2 Detalle de la estructura y clases de *ContractFalsifier*. 53
6.3 Archivo con los casos de prueba generados. 56

7.1 Ejecución del comando *Add* en **get_sign()**. 59
7.2 Ficheros en **get_sign()** para la utilización de *ContractFalsifier*. 59
7.3 Ejecución de *Run* en **get_sign()**. 60
7.4 Fichero con los casos de **get_sign()** *Run*. 60
7.5 Ejecución de *Check* en **get_sign()**. 60
7.6 Resultados de *Check* en **get_sign()**. 61
7.7 Ejecución del comando *Add* en **insert()**. 61
7.8 Ficheros en **insert()** para la utilización de *ContractFalsifier*. 61
7.9 Ejecución de *Run* en **insert()**. 62
7.10 Fragmento del fichero con los casos de **insert()** generado con el comando *Run*. 62
7.11 Ejecución de *Run* en **insert()**. 62
7.12 Fragmento del fichero con los casos de **insert()** generado con el comando *Run*. 62

B.1 Ejemplo de las variables de entorno necesarios 71
B.2 Archivos necesarios para la instalación de la aplicación 71

Índice de tablas

4.1 Caso de uso Añadir archivo. 41
4.2 Caso de uso Compilar archivo. 41
4.3 Caso de uso Ejecutar archivo. 42
4.4 Caso de uso Leer casos de prueba. 42
4.5 Caso de uso Validar. 43
4.6 Caso de uso Leer resultado. 43

Índice de fragmentos de código

2.1 Representación de los métodos en **insert()**. 12
2.2 Estructura utilizada en **insert()**. 13
2.3 Constructor en **insert()**. 13
2.4 Observadores en **insert()**. 13
2.5 Método modificador en **insert()**. 14
3.1 Fragmento introductorio de la técnica de ejecución simbólica. . . . 19
3.2 Fragmento del método **get_sign()**. 21
3.3 Representación de **insert()** en apéndice A. 21
3.4 Fragmento añadido a **get_sign()** para hacerlo simbólico. 22
3.5 Fragmento añadido a **insert()** para hacerlo simbólico. 22
3.6 Marcación de la variables como simbólicas. 29
3.7 Creación de una precondición en KLEE. 30
3.8 Fragmento de código preparado para la utilización de **insert().** (A) en KLEE 30
3.9 Creación de una precondición en KLEE. 33
3.10 Creación de una precondición en KLEE. 33
6.1 Comando para la creacción de *Add*. 54
6.2 Código pata la creación de opciones. 54
6.3 Implementación de **Callable**. 55
6.4 Ejecución del comando **clang** a través de Java. 55
6.5 Código para ejecutar varios comandos en el mismo proceso. 57

CAPÍTULO 1
Introducción

En los últimos años, la industria del Software se ha abierto paso de forma abrupta en cada aspecto de nuestra sociedad, desde ámbitos políticos hasta culturales. El desarrollo de software se ha convertido, a pasos acelerados, en uno de las mayores mercados mundiales, con un crecimiento exponencial y ganancias multimillonarias, trasformándose en uno de los principales sustentos económicos de nuestra sociedad actual.

Aunque suene inimaginable, todo lo que vemos y utilizamos contiene algo de software detrás. Podemos encontrar software en todas las facetas de nuestro día a día, desde que nos levantamos hasta que nos acostamos, durante el trabajo, en la enseñanza actual, en los nuevos avances médicos... facilitándonos y ayudándonos en nuestra vida cotidiana. Esto ha supuesto una gran influencia sobre nuestra sociedad, ocasionando que cada vez sea mayor el número de personas interesadas en la funcionalidad y características de calidad proporcionadas por una aplicación específica.

Durante los primeros años de vida del software, fue prioritario el desarrollo y la creación de nuevas aplicaciones, sin tener en cuenta el proceso o la propia calidad de éste sino solo la obtención del producto y, por consecuente, el afán por conseguir beneficios económicos. Esto ocasionó que su desarrollo se descontrolara, produciendo grandes pérdidas económicas debido a la multitud de proyectos inacabados o proyectos que terminaban en un cajón porque no cumplían con las especificaciones demandadas por el cliente, lo que llevó a lo que conocemos como la Crisis del Software.

Para poder controlar y manejar esta crisis, apareció la Ingeniería del Software. La Ingeniería del Software es la encargada de aplicar conocimiento práctico y sistemático propio del conocimiento científico a la producción de programas que se desarrollan a tiempo y dentro de las estimaciones de presupuesto, y la correspondiente documentación para desarrollarlos, instalarlos, usarlos y mantenerlos [19].

Con la llegada de la Ingeniería del Software, uno de los mayores retos en la producción del software es ofrecer productos que cumplan con la especificación del cliente dentro de su plazo correspondiente y, además, sin ningún sobrecoste que no esté presupuestado. Para ello, todo producto software debe seguir un proceso de producción controlado por las siguientes fases: Análisis y Especificación, Diseño, Desarrollo, Validación y Mantenimiento.

En todo momento, se debe tener en cuenta dichas fases durante el proceso de producción, ya que son dependientes entre ellas. El no realizarlas debidamente, siguiendo con los estándares, podría tener consecuencias desagradables para el desarrollo del producto, sobre todo si se incumple en las primeras fases del proceso. Es decir, cuando más tardío es el descubrimiento de una mala práctica o de un error, más difícil será su rectificación. Así pues, el mal desarrollo en las primeras fases provocará costes inimaginables en otros puntos de la producción e incluso problemas irreparables que serían más fácil de resolver si se iniciase de nuevo el desarrollo de dicho software. Pero cuando está en juego la seguridad de miles de personas y la pérdida de millones de euros, la solución no es tan simple como reiniciar o abandonar un proyecto.

Algunos grandes accidentes Software como el lanzamiento del Ariane 5 [3], cuya autodestrucción a los 40 segundos de despegar se debió a un error en la especificación y la mala adaptación del software desarrollado previamente para Ariane 4, o como el caso del acelerador lineal de radioterapia Therac-25 [21], en los que varios pacientes recibieron sobredosis de radiación, provocado incluso muertes por un mal diseño del software y unas prácticas de desarrollo inadecuadas. Estos hechos, entre otros tantos, demuestran la importancia de regular el desarrollo software y la necesidad de la aparición de la Ingeniería del Software como método regulador.

Como ya hemos podido ver en los ejemplos anteriores, sin la existencia de un proceso software capaz de controlar el desarrollo y la calidad de un producto software, todo intento de desplegar una aplicación llevaría al fracaso en algún punto del proceso, estando incluso en juego vidas humanas. Por eso, entre los mayores retos actuales en la Ingeniería del Software está asegurar dicha calidad y conseguir un producto que esté libre de errores.

Por ello, para poder hacer frente a la creciente complejidad de las aplicaciones software, es importante hacer uso de metodologías capaces de verificar el desarrollo de los sistemas mediante la lógica y las matemáticas, es decir, hacer uso de métodos formales. Los métodos formales permiten representar la especificación del software, verificación y diseño de componentes mediante notaciones matemáticas. El uso de métodos formales permite plantear de manera clara la especificación de un sistema, generando modelos que definen el comportamiento en términos del "qué debe hacer" y no del "cómo lo hace" [24] [9].

En consecuencia, en los últimos años se han incrementado las técnicas dentro de los métodos formales y muchas de ellas han empezado a automatizarse para reducir su gran complejidad. Este trabajo se centrará en una de estas técnicas: el diseño por contrato, que trata de aplicar ciertas condiciones y obligaciones a la implementación de un diseño software a través del uso de aserciones. Para conseguir una completa automatización, se utilizan técnicas potentes de abstracción, lo cuál puede conducir a que dichos contratos no sean del todo completamente correctos, creando la necesidad de herramientas de verificación y validación de contratos.

Con la motivación de este desafío, el principal objetivo de este trabajo es desarrollar una herramienta de análisis que, a partir de un programa en C y las restricciones representadas por su contrato software asociado, y mediante el uso de

la herramienta KLEE, permita la generación automática de casos de prueba con los que poder detectar y eliminar partes del contrato que no son ciertas.

1.1 Motivación

Para muchos desarrolladores la fase de especificación es una de las fases más tediosas. En muchos casos, la dificultad para escribir especificaciones software es consecuencia de una alta relación entre el costo y el beneficio de escribir y mantener especificaciones precisas sobre el código. Los desarrolladores escribirían especificaciones siempre que fueran simples, tuvieran una conexión directa con la implementación y les ayudasen a escribir y depurar código mejor y más rápido [18]. Una de las formas de hacerlo sería el diseño por contrato, simples especificaciones ejecutables escritas con la misma sintaxis que las expresiones de lenguajes de programación y que incrementan el desarrollo y ayudan durante las pruebas y la depuración.

En la actualidad, aunque existen lenguajes como JML o ACSL, que permiten especificar contratos para código Java o C respectivamente, muchos programadores son reacios a la idea de crear contratos por sí mismos pero dando las herramientas correctas para la generación de los contratos, dichos programadores los incluirían [17]. El estado del arte actual permite generar automáticamente contratos a partir del código fuente que pueden ser usados como entrada para analizadores cada vez más potentes. Sin embargo, debido a la dicotomía entre terminación del proceso y precisión del resultado, los contratos generados automáticamente pueden no ser completamente precisos o correctos, conteniendo algunos elementos que no están verificados [2].

En esta memoria analizaremos los contratos generados automáticamente y desarrollaremos una aplicación que permita refinar dichos contratos mediante el uso de la herramienta KLEE, que se encargará de generar automáticamente casos de prueba que permiten falsificar fragmentos de contratos erróneos. Usando estos casos, identificaremos y eliminaremos aquellos componentes que el proceso de validación determine que son demostradamente falsos.

1.2 Objetivos

El propósito de este trabajo es desarrollar una herramienta de validación automática de contratos software partiendo de un programa en C junto a su contrato, que asumiremos inferido automáticamente aunque la técnica se aplicaría igualmente a contratos desarrollados por un programador manualmente. Posteriormente, mediante el uso de la herramienta KLEE, generaremos casos de prueba que serán usados durante el proceso de validación para poder detectar qué componentes del contrato son decididamente falsos.

Para ello, seguiremos las siguientes fases:

- Ejecutar simbólicamente con KLEE el programa en C asumiendo ciertas condiciones que son dadas en los asertos del contrato inferido. Ejecutare-

mos KLEE por cada uno de los asertos que contenga el contrato, centrándonos particularmente en aquellos que la herramienta que los infirió marcó como «axiomas candidatos», lo que significa que su corrección no fue garantizada por construcción debido a que se necesitó aplicar abstracción para obtenerlo.

- Acceder e interpretar los casos de prueba creados por KLEE por cada ejecución del programa fuente.

- Ejecutar esta vez el programa en C pero usando los datos obtenidos en los casos de prueba como datos de entrada. En dicha ejecución haremos uso de observadores que se dedicarán a observar el desarrollo del programa, desde el inicio al fin.

- Analizar los resultados obtenidos de la ejecución mediante el uso de las precondiciones (condiciones dadas antes de la ejecución) y postcondiciones (condiciones dadas después de la ejecución) pertenecientes a los asertos del contrato.

- Falsificar los asertos que claramente son incorrectos.

En síntesis, el resultado principal de este proyecto es una herramienta para refinar los contratos inferidos automáticamente, detectando y eliminando los asertos que no sean totalmente correctos.

1.3 Impacto Esperado

En la actualidad, no existen herramientas capaces de generar automáticamente contratos software totalmente correctos o precisos, pudiendo contener ciertos elementos o componentes que no están totalmente verificados.

Por ello, nuestra intención es desarrollar una herramienta capaz de poder realizar una verificación y validación automática de dichos contratos software. Para ello, nuestra propuesta es descubrir principalmente qué componentes son totalmente falsos para poder descartarlos del contrato generado. Dicha tarea, como ya hemos comentado será realizada mediante la ayuda de la generación automática de casos de prueba KLEE.

Se espera esta herramienta contribuye a fomentar la utilización, por parte de los desarrolladores, de herramientas capaces de generar contratos software de forma automática. Este aumento se deberá a que, gracias a nuestra herramienta, los programadores serían capaces de validar o corregir los contratos inferidos por estas herramientas, o en otros casos, validar sus propios contratos de forma más eficaz y rápida. Esto llevaría a desarrollos de mayor calidad y con menos sobrecostes a lo planeados.

1.4 Metodología y planificación

La metodología elegida para desarrollar este proyecto ha sido una metodología ágil. Para ello hemos hecho uso de dos técnicas ágiles: SCRUM y KANBAN.

En concordancia, hemos creado un tablero Kanban, en la plataforma Trello[1], donde las tareas se han dividido en dos sprints (de dos semanas cada uno) mediante etiquetas de colores: rosa para el primer sprint y rojo para el segundo. También hemos dividido el tablero con cinco actividades globales: Registrar Unidades de Trabajo o UT, Esperar prioridad, Realizar tarea, Pruebas y Finalización de la tarea, por las cuales deberán pasar cada tarea hasta finalizar.

Figura 1.1: Puesta en marcha tablero Kanban.

Las tareas a realizar se han creado dentro de la columna «registrar UT». Dentro de cada una podemos encontrar en profundidad cuál es el objetivo a realizar de dicha tarea y una etiqueta indicando a qué parte del programa afecta.

Figura 1.2: Detalle de una UT.

[1]https://trello.com/es

Antes de empezar el primer sprint hemos priorizado las tareas, ordenándolas dentro de la columna de «esperar prioridad». Dichas tareas se han ordenado según su complejidad, tiempo invertido y aportación para el programa.

Figura 1.3: Priorización de tareas Sprint 1.

Una vez priorizadas y empezado el sprint 1, dichas tareas han circulado entre las diferentes actividades -realizar, probar y finalizar- durante las dos semanas de duración del sprint. Por ejemplo, la tarea de preparar código estuvo varias veces en pruebas antes de, por fin, darse por finalizada. Por otra parte, antes de finalizar el primer sprint, priorizamos las tareas del segundo sprint, otra vez, en la columna de «esperar prioridad».

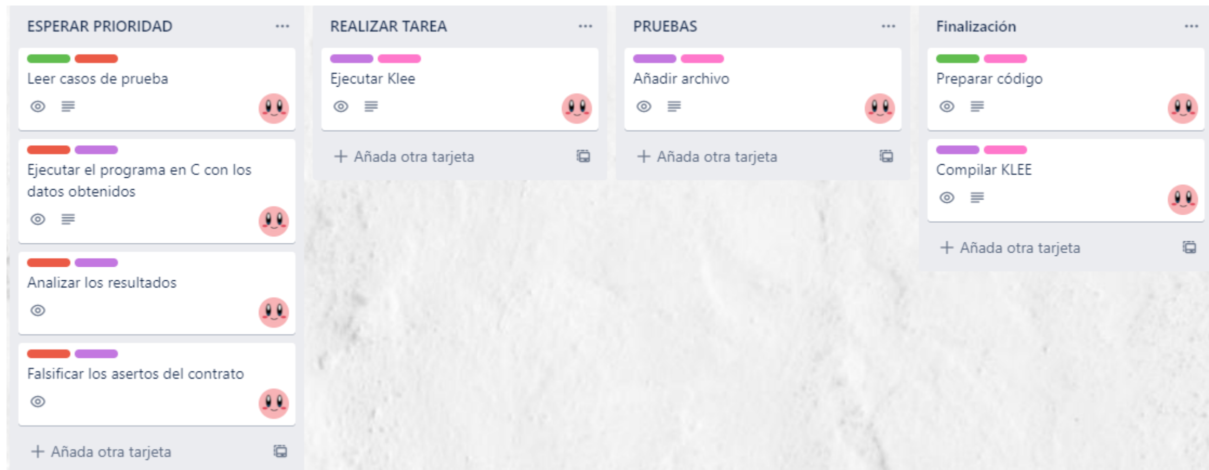


Figura 1.4: Representación de la mitad del Sprint 1.

Una vez finalizado el transcurso del primer sprint, donde todas sus tareas llegaron a finalizarse, y preparado correctamente todas la tareas para el inicio del

segundo sprint, se pudo empezar con el mismo sin ningún retraso o problema. En la figura de a continuación podemos observar estos hechos mirando que las tareas de cada sprint, etiquetadas de rosa y rojo, están respectivamente en «finalización» y en «realizar».

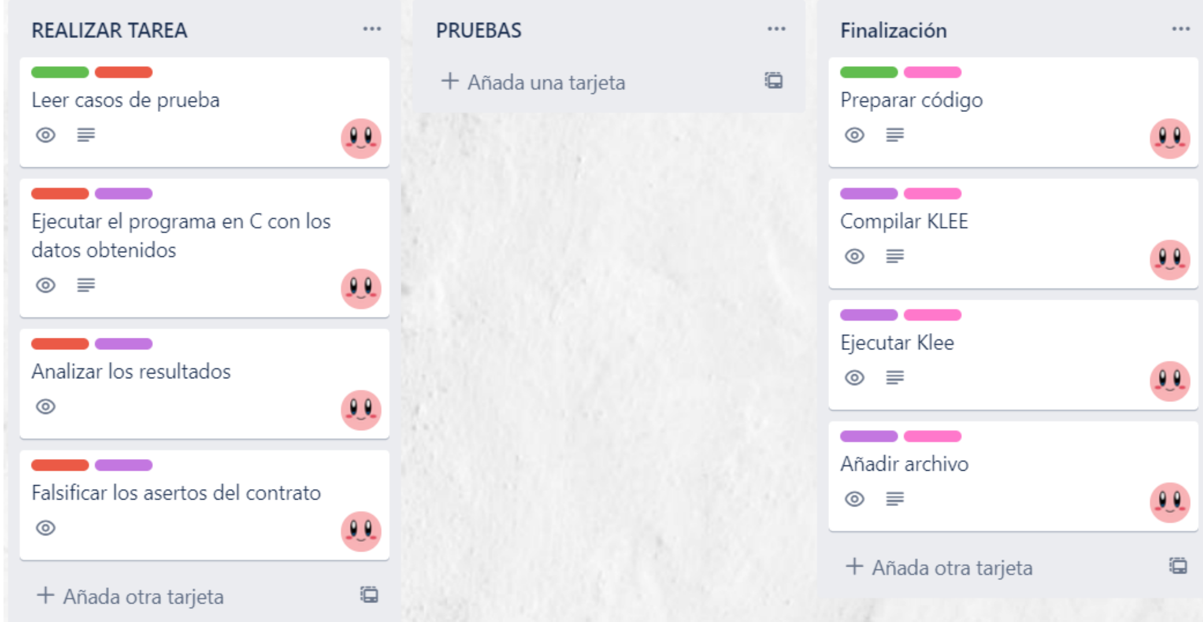


Figura 1.5: Inicio del Sprint 2.

Al empezar a realizar las tareas para el segundo sprint nos dimos cuenta que había que modificar una tarea ya realizada y terminada en el primer sprint, Añadir archivo, para poder desarrollar correctamente las tareas de analizar los resultados y falsificar los asertos del contrato.

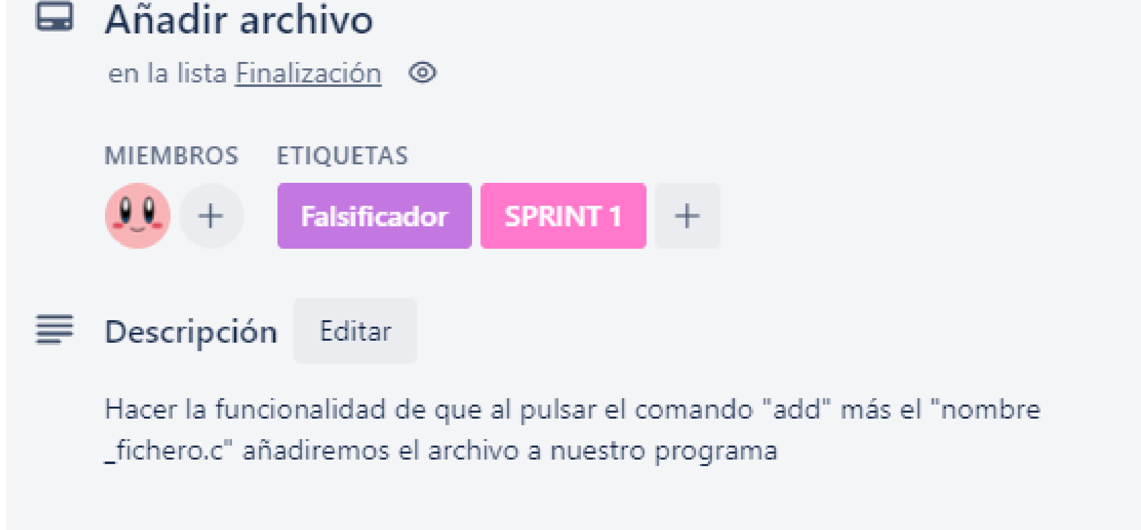


Figura 1.6: Aspecto de la UT Añadir archivo en el Sprint 1.

Como podemos observar, comparando las imágenes 1.6 y 1.7, hemos tenido que reabrir la tarea y pasarla a «realizar tarea» para cambiar y ampliar su funcionalidad con respecto al comando *Add*. Este cambio se ha debido a la necesidad de

tener todos estos archivos a la hora de la validación de los axiomas candidatos, funcionalidad que comprende tres tareas de nuestro tablero.

Figura 1.7: Aspecto de la UT Añadir archivo modificada en el Sprint 2.

En la imagen 1.8 vemos representado el aspecto que tenía nuestro KANBAN cuando reabrimos la tarea. Una vez realizado este cambio no tuvimos más problemas y el segundo sprint prosiguió de manera acorde a lo esperado hasta su finalización.

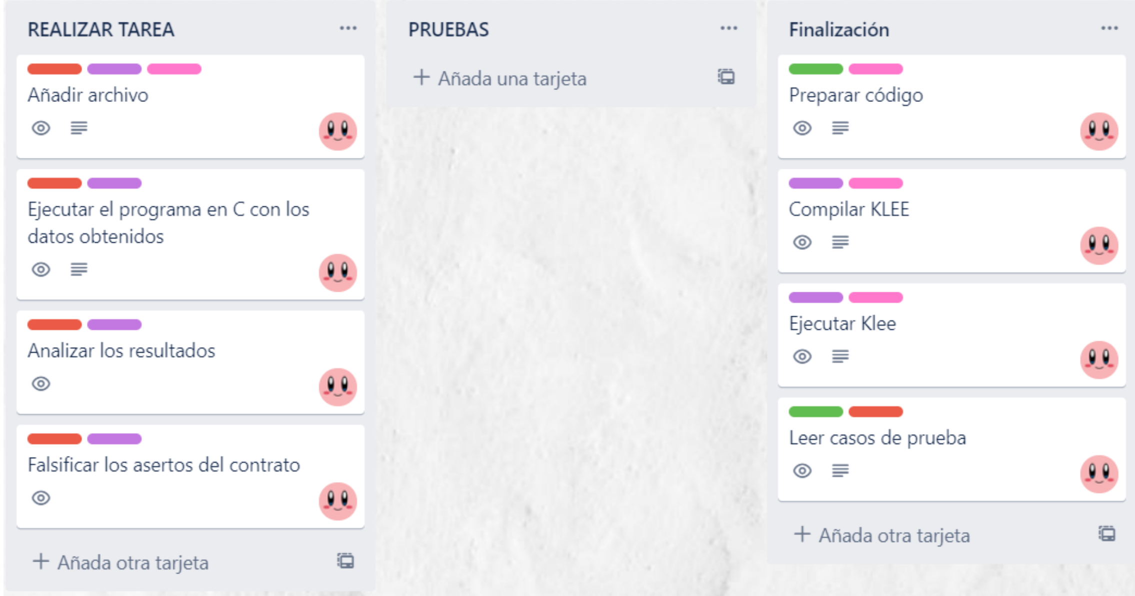


Figura 1.8: Representación del tablero con la tarea reabierta en el Sprint 2.

1.5 Estructura

Este trabajo se estructura en ocho Capítulos diferentes y dos apéndices:

En el Capítulo 1 se hace una introducción analizando el contexto de este trabajo y explicando la motivación para la realización de este.

En el Capítulo 2 se expone el contexto tecnológico alrededor de nuestro trabajo explicando algunas herramientas relacionadas con los contratos software y la generación de casos de pruebas. También en este capítulo, introduciremos el programa conductor del proyecto mediante una de las herramientas.

En el Capítulo 3 se analiza la herramienta KLEE, explicaremos con detalle su funcionalidad y el papel clave de esta herramienta en nuestro trabajo.

En el Capítulo 4 se descubre el problema a tratar junto a sus necesidades y oportunidades, tales como el análisis de requisitos funcionales y no funcionales, para poder llevar a cabo una correcta realización del validador automático de contratos software.

En el Capítulo 5 se documenta con detalle el diseño elegido y las tecnologías utilizadas para llevarlo a cabo.

En el Capítulo 6 se explica en profundidad el desarrollo y las pautas que se han seguido para la creación de nuestro programa.

En el Capítulo 7 se presenta todas las pruebas realizadas para la comprobación y validación de la correcta solución del problema.

En el Capítulo 8 se ofrece un resumen del trabajo y las conclusiones que se han obtenido con la realización de este.

En el Apéndice A, se proporciona el código en C del programa **insert()**, conductor de nuestra investigación.

Y finalmente en el Apéndice B, se detalla como se instala y ejecuta nuestro programa desarrollado para sistemas Linux.

CAPÍTULO 2
Contexto tecnológico

Debido a la gran complejidad que supone para los desarrolladores escribir especificaciones, ya que se requieren ciertas habilidades por parte de éstos y su mal desempeño puede provocar costes severos en otras fases de la producción, se han empezado a desarrollar herramientas capaces de computar de forma semiautomática distintas formas de especificaciones. En los últimos años, el amplio interés que han adquirido en estas herramientas, por las grandes ventajas que otorgan, ha incrementado notablemente el interés por este campo, lo que ha conducido a una variedad de ellas.

En este Capítulo, en primer lugar, vamos a describir una herramienta capaz de inferir contratos software de forma automática, **KindSpec**. La herramienta KindSpec [1, 2] está desarrollada por el grupo ELP del Departamento de Sistemas Informáticos y Computación de la UPV. Para la especificación del ejemplo utilizado en nuestro proyecto, hemos usado la técnica de inferencia utilizada en la herramienta, la cual será explicada con detalle en la siguiente sección.

En segundo lugar, resumimos la semántica utilizada por KindSpec, $\mathbb{K}$, tecnología que se puede usar tanto para la creación de nuevos lenguajes de programación o de herramientas de análisis, entre otras cosas, que se explicará también a continuación.

Y por último, vamos a analizar otras herramientas capaces de generar casos de prueba automáticamente como KLEE , herramienta muy importante para el desarrollo de este proyecto. Por una parte, hablaremos de **Symbolic Java Pathfinder** desarrollada por la NASA y se consideraría como la herramienta análoga de KLEE pero para programas JAVA en de vez de programas en C. Por otra parte, explicaremos **Pex** herramienta automática de generación de pruebas de caja blanca para .NET integrada en Microsoft Visual Studio.

2.1 KindSpec

KindSpec[1] es una herramienta automática que sintetiza contratos software basándose en una abstracción de la ejecución simbólica para un fragmento significativo del lenguaje C, llamado KernelC que soporta estructuras basadas en

[1] http://safe-tools.dsic.upv.es/kindspec2_2/

punteros, manipulación de montículos y recursividad. Partiendo de la definición semántica de KernelC en el marco semántico de 𝕂 (véase la siguiente sección 2.2), KindSpec enriquece las facilidades de ejecución simbólica proporcionadas recientemente por K con capacidades novedosas para la síntesis de contratos que se basan en la subsunción abstracta.

Los contratos que sintetiza consisten, esencialmente, en afirmaciones lógicas que caracterizan el comportamiento de la función y que se expresan como precondiciones del método (impuestas a los argumentos) y poscondiciones (relacionando los argumentos y el resultado de un método), todas ellas representadas en varios conjuntos de axiomas. Dentro de esos axiomas existen dos tipos: axiomas que son correctos por construcción, cuando la abstracción no es necesaria, y axiomas candidatos cuya corrección no puede garantizarse debido al uso de la teoría de interpretación abstracta [6].

Con una correcta validación podríamos probar eventualmente la corrección de los axiomas candidatos y descartar los erróneos debidos a una sobreaproximación. Es aquí donde se origina el desafío de nuestro proyecto, en el que vamos a intentar falsificar los axiomas candidatos falsos usando la herramienta KLEE .

2.1.1. Método de inferencia

A continuación, vamos a introducir el ejemplo conductor de nuestro proyecto, al mismo tiempo que describimos la metodología de inferencia usada en la herramienta KindSpec.

Para empezar, la técnica de inferencia usada en KindSpec se basa en el esquema de clasificación desarrollado para la abstracción de datos en [13], donde una función KernelC o un método puede ser considerado como un constructor, que devuelve una nueva estructura de datos u objeto; un modificador, que altera un objeto existente, es decir, cambia el estado de uno o más de sus atributos; y un observador, que inspecciona el objeto y devuelve un valor (sin modificar ningún objeto del programa) que caracteriza uno o más de sus atributos de estado (en nuestro ejemplo, dicho estado se representa con un 1 para el valor verdadero y con un 0 para el valor falso).

Nuestro ejemplo[2] conductor es una implementación de KernelC donde, dada una estructura, insertaremos elementos si éste no está ya incluido y la capacidad de la estructura es mayor que el tamaño actual. Para realizar esa tarea, el programa está compuesto por siete métodos: un constructor, un modificador, y cinco métodos observadores. En el siguiente fragmento podemos ver sintetizados los perfiles de estos métodos.

```
1 struct set {...}
2
3 /* constructor */
4 struct set* new(int capacity){...}
5
6 /* observadores */
7 int isnull(struct set *s) {...}
```

[2]El programa completo se encuentra en el Apéndice A

```
int isempty(struct set *s) {...}
int isfull(struct set *s) {...}
int contains(struct set *s,int x){...}
int length(struct set *s){...}

/* modificador */
int insert(struct set *s, int x) {...}
```

Código 2.1: Representación de los métodos en **insert()**.

En primer lugar, el programa está constituido por una estructura en la que se almacenan la capacidad máxima de la estructura, el tamaño actual de elementos y los elementos.

```
struct set {
  int capacity;
  int size;
};
```

Código 2.2: Estructura utilizada en **insert()**.

En segundo lugar, nos encontramos con el constructor de la estructura de datos, **set* new(c)** donde, dada una capacidad **c**, construimos una nueva estructura con el formato descrito anteriormente.

```
struct set* new(int capacity) {
  struct set *new_set;
  new_set = (struct set*) malloc(sizeof(struct set));
  if(new_set == NULL) return NULL; /* no memory left */
  new_set->capacity = capacity;
  new_set->size = 0;
  *(new_set->elem) = malloc(capacity * sizeof(int));
  return new_set;
}
```

Código 2.3: Constructor en **insert()**.

En tercer lugar, podemos ver los cinco observadores que conforman este fragmento. Dichos métodos devuelven un cero excepto si se cumplen ciertas condiciones: En **isnull(s)** devolverá un uno si la estructura **s** es igual al valor nulo; **isempty(s)** devolverá uno si no hay ningún elemento; **isfull(s)** también devolverá un uno si el tamaño es mayor o igual a la capacidad; **contains(s,x)** retornará uno si la estructura contiene el elemento **x**; por último **length(s)** indica el tamaño de la estructura si no es nula. Como hemos comentado anteriormente el valor cero y uno representan respectivamente en C los valores falso y verdadero.

```
int isnull(struct set *s) {
  if(s==NULL) return 1;
  return 0;
}

int isempty(struct set *s) {
  if(s==NULL) return 0;
  if(s->elem==NULL) return 1; /* s is empty */
```

```
9     return 0;
10  }
11
12  int isfull(struct set *s) {
13    if(s==NULL) return 0;
14    if(s->size >= s->capacity) return 1; /* s is full */
15    return 0;
16  }
17
18  int contains(struct set *s, int x) {
19      int i;
20    if(s==NULL) return 0; /* s is NULL */
21    for(i = 0; i < s->capacity; i++){
22      if(s->elem[i] == x) return 1; /* element found */
23    }
24    return 0; /* element NOT found */
25  }
26
27  int length(struct set *s) {
28    if(s==NULL) return 0; /* s is NULL */
29    return s->size;
30  }
```

Código 2.4: Observadores en **insert()**.

Por último, tenemos el método modificador **insert(s,x)** que se encarga de insertar el elemento **x** a la estructura pero para ello, tiene que comprobar antes ciertas condiciones. Primero, revisa si el puntero a la estructura es distinto del valor nulo y que el elemento a insertar tampoco lo sea. Después verifica sí en la estructura caben más elementos y si contiene algún elemento, que estos sean diferentes al que deseamos insertar, **x**. Si nada de esto se cumpliera devolvería un cero, en caso contrario devolvería un uno y modificaría la estructura **s** para añadir el nuevo elemento **x** a la estructura.

```
1   int insert(struct set *s, int x) {
2     int found;
3     int i;
4
5     if(x==NULL)
6       return 0;
7     if(s==NULL)
8       return 0; /* NULL set */
9
10    if(s->size >= s->capacity) return 0; /* no space left */
11
12    if(s->elem == NULL) { /* empty set */
13      s->elem[s->size] = x;
14      s->size = 1;
15      return 1;
16    }
17
18    found = 0;
19    for(i = 0; i<3; i++) {
20      if(s->elem[i] != NULL){
21        if(s->elem[i] == x) {
22          found = 1;
23        }
```

```
24        }
25    }
26
27    if(found) return 0; /* element already in the set */
28    s->elem[s->size] = x;
29    s->size = s->size + 1;
30
31    return 1; /* element added */
32 }
```

Código 2.5: Método modificador en **insert()**.

A partir del fragmento KernelC, KindSpec sintetizará, por cada método modificador **m**, un contrato de la forma $< P, Q, L >$ donde P es la condición previa del método o precondición, Q es la condición posterior del método o poscondición y por último, L es el conjunto de ubicaciones del programa que se ven (potencialmente) afectados por la ejecución del método modificador.

Para ello, primero calculará el conjunto de formulas de implicación de la forma $p => q$, donde p y q son conjunciones de ecuaciones de la forma $l = r$. En cada ecuación, por una parte, en el lado izquierdo l puede haber una llamada a una función observadora o la palabra clave *ret* y en la parte derecha r está representada el valor de retorno de esa llamada (cuando l es *ret*, r representa el valor de retorno del método modificador m siendo observado).

Por lo tanto, dado el conjunto de formulas $\{p_1 \Rightarrow q_1, ..., p_n \Rightarrow q_n\}$, P se define como $p_1 \vee ... \vee p_n$, la poscondición Q como la fórmula $(p_1 \Rightarrow q_1) \wedge ... \wedge (p_n => q_n)$, y L como el conjunto de variables/objetos del programa cuyo valor puede verse afectado mediante la ejecución de m.

Siguiendo con nuestro ejemplo, el contrato inferido para nuestro método modificador **insert(s,x)** contiene cinco axiomas (todos ellos determinados por implicación) representados en la siguiente figura.

$$(\ \mathtt{isnull(s)} = 1\ ) \Rightarrow (\ \mathtt{isnull(s')} = 1\ \wedge\ \mathtt{ret} = 0\ )$$

$$(\ \mathtt{isfull(s)} = 1\ ) \Rightarrow \begin{pmatrix} \mathtt{contains(s',x)} = \mathtt{contains(s,x)} \wedge \\ \mathtt{length(s')} = \mathtt{length(s)} \wedge \\ \mathtt{isfull(s')} = 1\ \wedge\ \mathtt{ret} = 0 \end{pmatrix}$$

$$(\ \mathtt{contains(s,x)} = 1\ ) \Rightarrow \begin{pmatrix} \mathtt{contains(s',x)} = 1 \wedge \\ \mathtt{length(s')} = \mathtt{length(s)}\ \wedge\ \mathtt{ret} = 0 \end{pmatrix}$$

$$(\ \mathtt{isempty(s)} = 1\ \wedge \mathtt{isfull(s)} = 0\ ) \Rightarrow \begin{pmatrix} \mathtt{isempty(s')} = 0\ \wedge\ \mathtt{contains(s',x)} = 1 \wedge \\ \mathtt{length(s')} = \mathtt{length(s)} + 1\ \wedge\ \mathtt{ret} = 1 \end{pmatrix}$$

$$\begin{pmatrix} \mathtt{isnull(s)} = 0\ \wedge \mathtt{isempty(s)} = 0 \wedge \\ \mathtt{isfull(s)} = 0\ \wedge \mathtt{contains(s,x)} = 0 \end{pmatrix} \Rightarrow \begin{pmatrix} \mathtt{isnull(s')} = 0\ \wedge\ \mathtt{isempty(s')} = 0 \wedge \\ \mathtt{contains(s',x)} = 1 \wedge \\ \mathtt{length(s')} = \mathtt{length(s)} + 1\ \wedge \mathtt{ret} = 1 \end{pmatrix}$$

Figura 2.1: Axiomas de poscondición esperados en el método **insert(s,x)** [2].

Dichos axiomas se pueden interpretar como sigue: en el primer axioma, siendo la estructura nula, si intentamos hacer uso de método **insert(s,x)**, después de la ejecución del método el valor de la estructura seguirá siendo nulo y el método (representado como *ret*) devolverá un cero; es decir, que no se habrá podido insertar ningún elemento ya que la estructura es igual al valor nulo. En el segundo

axioma, si la estructura está llena, al ejecutar **insert(s,x)** el contenido será el mismo que antes de la ejecución, el tamaño también lo será, continuará estando llena y el método devolverá cero, ya que al estar lleno no ha sido posible la inserción. En el tercer axioma, si la estructura contiene el elemento que queremos insertar no se producirá ningún cambio. En el cuarto axioma si la estructura esta vacía insertaremos el elemento **x** y la estructura dejará de estar vacía. Finalmente, en el último, si la estructura no es nula, ni está vacía, ni está llena y ni contiene al elemento que queremos insertar, al finalizar la ejecución la estructura contendrá el elemento **x**, el tamaño se habrá incrementado en uno, seguirá siendo no nula y no vacía y el valor $ret = 1$ confirmará la inserción.

2.2 $\mathbb{K}$: A Semantic Framework

$\mathbb{K}$ [20] es un *framework* semántico ejecutable en el que se pueden definir lenguajes de programación, cálculos, así como sistemas de tipos o herramientas de análisis formal, haciendo uso de configuraciones, cálculos y reglas.

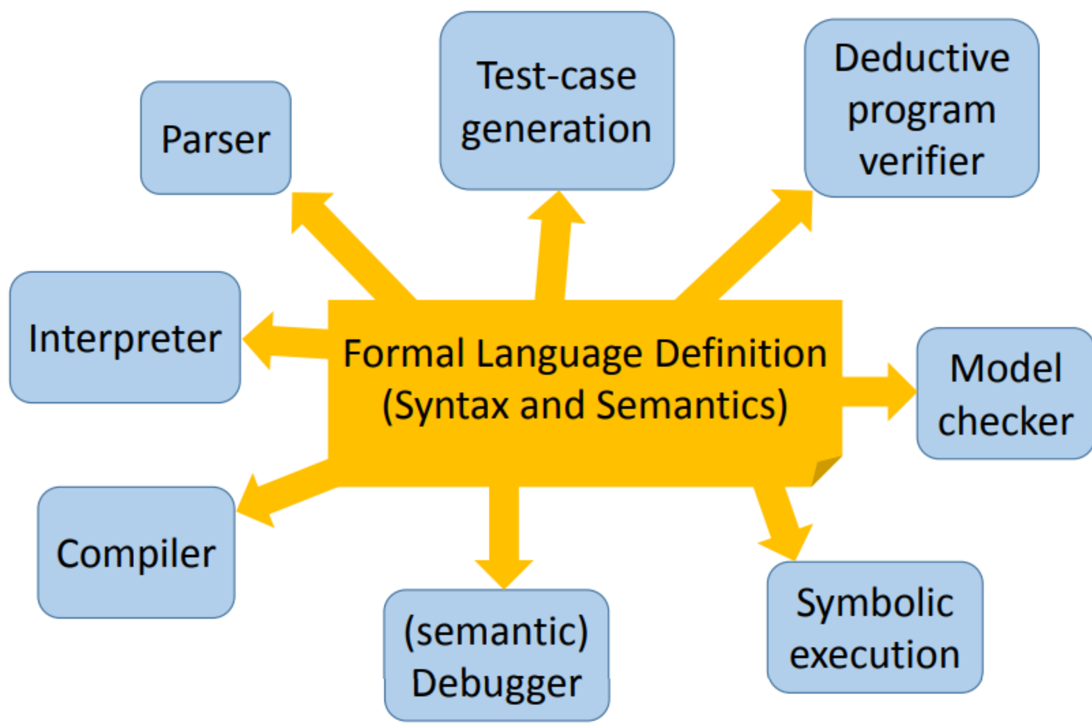


Figura 2.2: Arquitectura del *framework* $\mathbb{K}$ [20].

Las configuraciones organizan el estado del sistema o programa en unidades llamadas celdas, que se etiquetan y se pueden anidar. Los cálculos llevan un «significado computacional» como estructuras especiales de listas anidadas que secuencian tareas computacionales, como fragmentos de programa. En particular, los cálculos amplían el lenguaje original o la sintaxis del cálculo.

Las reglas $\mathbb{K}$ generalizan las reglas de reescritura convencionales al hacer explícitas qué partes del término leen, escriben o no interesan. Esta distinción hace

de 𝕂 un marco adecuado para definir lenguajes o cálculos verdaderamente concurrentes, incluso en presencia de *sharing*. Los cálculos se pueden manejar como cualquier otro término en un entorno de reescritura, es decir, se pueden combinar, mover de un lugar a otro en el término original, modificar o incluso eliminar.

2.3 Symbolic Java Pathfinder

La herramienta de análisis de software Symbolic Java Pathfinder (SPF) [14, 15] combina la ejecución simbólica con la verificación de modelos para la generación automatizada de casos de prueba y la detección de errores en programas de código en Java. SPF incorpora técnicas para manejar estructuras de datos de entrada, cadenas y llamadas nativas a bibliotecas externas, así como para resolver restricciones matemáticas complejas.

En esta herramienta, los programas se ejecutan en entradas simbólicas que representan múltiples entradas concretas y los valores de las variables del programa se representan mediante expresiones sobre esas entradas simbólicas. Las restricciones sobre estas expresiones se generan a partir del análisis de diferentes rutas a través del programa. Las restricciones se resuelven utilizando resolutores estándar para generar entradas de prueba garantizadas para lograr criterios de cobertura complejos.

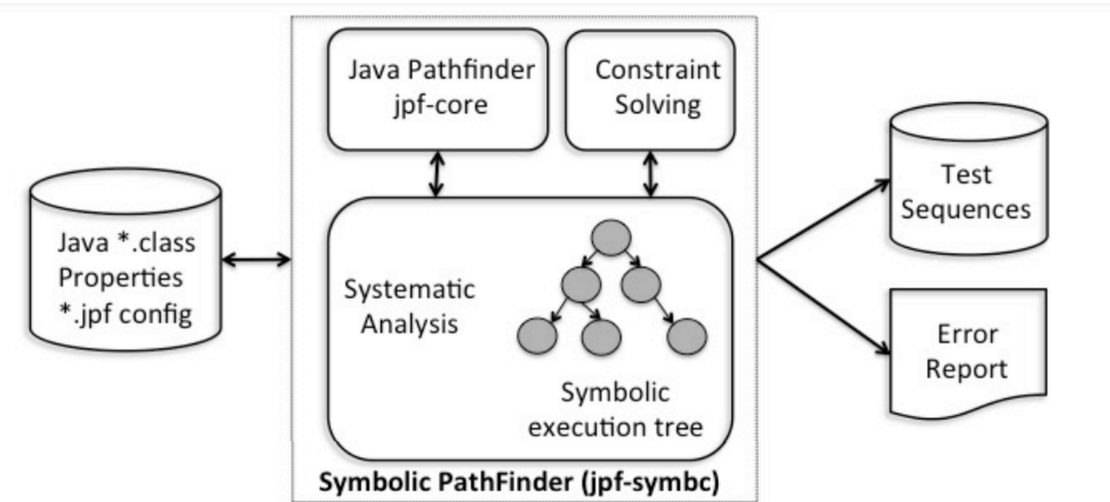


Figura 2.3: Esquema de Symbolic Java Pathfinder [15].

La verificación de modelos se utiliza para explorar diferentes ejecuciones de programas simbólicos, para manejar sistemáticamente los alias en las estructuras de datos de entrada y para analizar el multiproceso presente en el código. SPF se ha utilizado con éxito en la NASA, en el ámbito académico y en la industria.

2.4 Pex

Pex [22] genera automáticamente conjuntos de pruebas con una alta cobertura de código mediante el análisis automatizado de caja blanca para programas .NET. Con este fin, Pex realiza un análisis sistemático del programa (utilizando ejecución simbólica dinámica, similar a la verificación del modelo delimitada por la ruta) para determinar las entradas de prueba para las pruebas unitarias parametrizadas.

Pex aprende el comportamiento del programa al monitorizar los seguimientos de ejecución. Pex utiliza el demostrador de teoremas y el resolutor de restricciones Z3 [7] para razonar sobre la viabilidad de los caminos de ejecución y para obtener modelos básicos para sistemas de restricciones.

CAPÍTULO 3

KLEE: Generación automática de pruebas de alta cobertura para software complejo

En este Capítulo explicamos, en primer lugar, en qué consiste la ejecución simbólica ya que tanto la inferencia de los contratos como su validación (o, en su caso, refutación) se basa fundamentalmente en esta técnica de análisis. Después de esta explicación, detallamos en profundidad en qué consiste KLEE y con qué fin la hemos utilizado en nuestro trabajo.

3.1 Ejecución simbólica

La ejecución simbólica es una técnica de análisis en la cuál se ejecutan programas con expresiones simbólicas en lugar de usar datos concretos, es decir, usa valores de entrada simbólicos en lugar de valores concretos. Con estos valores se recorre el programa creando condiciones de camino que se van actualizando cada vez que encontramos una instrucción condicional en el código, descubriendo así distintas restricciones de entrada para los diferentes caminos de ejecución y con qué valores concretos poder acceder a ellos.

Para hacer más visible cómo funciona esta técnica vamos a hacer uso del siguiente ejemplo:

```
if(a > b)
    then a:= a - b
    else b:= b - a
```

Código 3.1: Fragmento introductorio de la técnica de ejecución simbólica.

Este fragmento contiene una condición que será evaluada a verdadero o falso. A simple vista podemos ver que, al ejecutar este programa, vamos a desarrollar dos posibles caminos de ejecución. Para poder acceder a estos caminos, los datos de entrada deberán cumplir con la condición **a >b**, conocida como condición de camino.

El propósito de la ejecución simbólica es construir un árbol de ejecución que cubre todas las posibles trayectorias haciendo uso de variables simbólicas. Estas variables son representaciones simbólicas de los datos de entrada del programa que nos permitirán explorar los distintos caminos de ejecución.

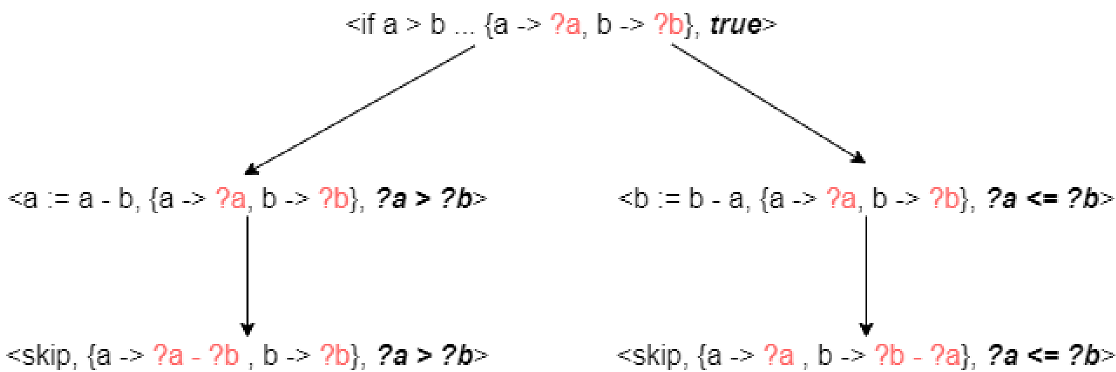


Figura 3.1: Ejecución simbólica.

En la Figura 3.1 podemos observar el árbol de ejecución de nuestro ejemplo donde, para acceder hasta el final de los posibles caminos, habrá que cumplir con las expresiones **?a >?b** ó **?a <= ?b**. Estas expresiones están constituidas por las variables simbólicas en que ***a*** representa nuestro dato y ***?a*** una representación simbólica de este. Por lo tanto, para acceder a un cierto camino, nuestros datos de entrada deberán cumplir con la condición asociada a dicho camino.

3.2 KLEE

Una vez ya explicado en que consiste la ejecución simbólica, ya podemos entender mejor como funciona la la herramienta KLEE explicada detalladamente a continuación.

3.2.1. Introducción

KLEE es una herramienta de ejecución simbólica, desarrollada por la Universidad de Stanford, capaz de generar automáticamente pruebas con un alto rango de cobertura en una gran variedad de programas. Las pruebas generadas automáticamente por KLEE cubren entre el 80 % y el 100 % de las declaraciones ejecutables y, en conjunto, superan significativamente la cobertura de las propias pruebas escritas a mano por los desarrolladores [5].

3.2.2. Instalación

Antes de nada, KLEE solo se puede instalar en sistemas Linux x86-64 y macOs. En este segundo sistema, solo se pueden usar programas muy sencillos ya que los dos entornos usados por KLEE, uClibc y POSIX, que dan la posibilidad de ejecutar programas más reales y complejos, no dan soporte a macOS.

Hay diversas maneras de poder instalar KLEE, desde versiones web donde se puede ejecutar programas muy pequeños y fáciles hasta la creación de un contenedor *Docker*.

En nuestro caso, hemos instalado una versión de KLEE a través de LLVM6 (actualmente se puede instalar ya desde LLVM9 pero en este proyecto se ha preferido mantener la versión anterior, porque este proyecto se empezó a desarrollar con dicha versión).

Para más información, consultar las guías de instalación [1].

3.2.3. Preparación

Como hemos explicado anteriormente, KLEE es una herramienta que ejecuta de forma simbólica un programa en C. Para poder hacer esto, hay que preparar el código de antemano para que KLEE pueda cumplir debidamente con su función, crear casos de prueba.

Para ello vamos a hacer uso de dos ejemplos: por una parte vamos a usar un programa sencillo en C para explicar cómo funciona KLEE (dicho programa se puede encontrar en la documentación oficial [2]) y, por otra parte, un programa más complejo con estructuras y librerías, también en C, con el cuál hemos probado nuestro trabajo.

Este primer ejemplo 3.2 es una función muy sencilla, **get_sign**, en que dado un valor de entrada nos dirá si es 0 (neutro), positivo o negativo. Observando este código se puede deducir que solo hay tres caminos de ejecución; es decir, al introducir este programa en KLEE veremos que creará solo tres casos de prueba que corresponde a estos tres caminos posibles de ejecución.

```
1 int get_sign(int x) {
2     if (x == 0)
3         return 0;
4     if (x < 0)
5         return -1;
6     else
7         return 1;
8 }
```

Código 3.2: Fragmento del método **get_sign()**.

Este segundo ejemplo 3.3, explicado anteriormente de forma detallada en la Sección 2.1.1, se trata de una estructura lineal en la cuál iremos insertando valores. Nuestro propósito es que dicha estructura sea simbólica y los valores que añadamos también lo sean.

```
1 struct set {...}
2 struct set* new(int capacity){...}
3 int isnull(struct set *s) {...}
4 int isempty(struct set *s) {...}
5 int isfull(struct set *s) {...}
```

[1]https://klee.github.io/getting-started/

[2]http://klee.github.io/releases/docs/v1.4.0/tutorials/testing-function

```
6 int contains(struct set *s,int x){...}
7 int length(struct set *s){...}
8 int insert(struct set *s, int x) {...}
```

Código 3.3: Representación de **insert()** en apéndice A.

Junto a estos ejemplos vamos a explicar dos funciones esenciales que necesitaremos para poder preparar nuestro código antes de ejecutar KLEE.

klee_make_symbolic()

Esta función nos permite indicar qué variables queremos que se ejecuten como datos de entrada simbólicos. Dicha función requiere de tres argumentos: la dirección de memoria de la variable que queremos que se comporte de forma simbólica, su tamaño y su nombre.

En la siguiente función **main** creada para el primer ejemplo, vemos cómo hemos marcado la variable de entrada de la función **get_sign()** como simbólica usando la función **klee_make_symbolic()**.

```
1 int main() {
2     int a;
3     klee_make_symbolic(&a, sizeof(a), "a");
4     return get_sign(a);
5 }
```

Código 3.4: Fragmento añadido a **get_sign()** para hacerlo simbólico.

Para el segundo ejemplo, ha sido más compleja la implementación de ciertas variables como simbólicas; concretamente, marcar la estructura como una estructura simbólica mediante la función **klee_make_symbolic()**. Dicha función no se comporta como indica la documentación [23], pero pudimos descubrir que basta con crear una variable del mismo tipo y tamaño que la estructura que queremos señalar como simbólica. Una vez creada, marcamos esta variable como simbólica en vez de la estructura. Ya señalada como simbólica, copiaremos dicha variable en la estructura usando la función **memcpy(void *destino, const void *fuente, size_t n)** [4].

```
1  int main(){
2      int capacidad;
3      int elementoInsertar;
4      int lista[3];
5
6      klee_make_symbolic(&capacidad, sizeof(capacidad), "capacidad");
7      klee_make_symbolic(&elementoInsertar, sizeof(elementoInsertar), "
           elementoInsertar");
8      klee_make_symbolic(&lista,sizeof(lista),"lista");
9
10     struct set *listaElementos = new(capacidad);
11     memcpy(listaElementos->elem,lista,sizeof(lista));
12
13     insert(listaElementos,elementoInsertar);
14 }
```

Código 3.5: Fragmento añadido a **insert()** para hacerlo simbólico.

En esta función **main()** vemos representado cómo hemos conseguido que la estructura sea simbólica. En primer lugar, para poder marcar como simbólicos los elementos de la estructura y no la estructura en sí, hemos tenido que forzar el tamaño de los elementos de la estructura creando una lista de elementos con un tamaño en concreto, **lista[3]**. Después de esto, hemos marcado como simbólicas la capacidad, el elemento a insertar y la lista de elementos. Esta lista de elementos, como hemos comentado en el apartado anterior, la hemos copiado en la estructura con **memcpy**.

klee_assume

Esta función es usada para restringir los valores que pueden tomar las variables simbólicas; es decir, equivaldría a envolver el programa dentro de un *if(condición)* haciendo que KLEE cumpla esta condición a la hora de crear los casos de prueba. Gracias a esto, podremos asumir ciertas precondiciones para nuestro programa usando las variables simbólicas y los observadores, si los tiene. Con ellos, formaremos diferentes condiciones y las envolveremos dentro de la función **klee_assume()**.

Para el primer ejemplo, un posible uso (entre otros) sería asumir que la variable de entrada simbólica sea distinta de 0 y esto se representaría como **klee_assume(a != 0)**. Al hacer uso de esta función haremos que KLEE tenga en cuenta dicha condición para crear sus casos de prueba, mostrando esta vez solo la creación de dos casos, positivo y negativo.

En el segundo ejemplo, al tener varios observadores, podemos jugar un poco más con las posibilidades que ofrece esta función. Hay que tener en cuenta que el uso de operadores de circuito corto dentro de la condición puede resultar errónea por como esta configurado KLEE [12]. Por eso es mejor poner cada condición a cumplir por separado, es decir, un **klee_assume()** por cada condición que, para nuestro ejemplo, quedaría así:

klee_assume(contains(listaElementos,elementoInsertar) == 1);

klee_assume(isfull(listaElementos) == 1);

donde, dentro de la condición, hemos usado los observadores pertenecientes al programa para asumir que se darán ciertas precondiciones al empezar.

3.2.4. Generación de casos de prueba

Por lo que se refiere a la generación de casos de prueba, necesitaremos acceder al terminal de nuestro sistema en el cuál se ha instalado la herramienta previamente. También necesitaremos el programa para el que deseamos que KLEE cree casos de prueba y preparar dicho código como ya hemos mencionado en la sección anterior.

Una vez preparado todo, nos encargaremos de compilar el archivo usando la terminal con el siguiente comando:

clang -I ../../include -emit-llvm -c -g ***nombre_del_fichero.c***

Si se produce algún error de compilación, KLEE nos avisaría en qué consiste dicho error y en que línea del programa se encontraría para poder así solucionarlo de forma más rápida y cómoda.

Una vez conseguida la compilación del programa, ejecutaremos el programa con las variables marcadas anteriormente como simbólicas usando el siguiente comando de KLEE:

klee ***nombre_del_fichero.bc***

Si todo va según lo esperado, se generaran los diversos casos de prueba junto a un directorio donde se guardaran los ficheros creados - en la siguiente sección hablaremos de ellos - y nos lo anunciará con un texto parecido al siguiente:

```
KLEE: output directory is "/home/sandra/Escritorio/TFG/KLEE/klee-out-0"
KLEE: Using STP solver backend

KLEE: done: total instructions = 32
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
```

Figura 3.2: Salida por terminal de la generación de casos de prueba.

Si ese no es el caso, nos indicará con un mensaje de error detalladamente dónde esta el problema del fallo de la ejecución. También creará unos ficheros con información más clara del fallo -explicados también en la siguiente sección.

3.2.5. Ficheros generados

Antes de empezar, siempre que ejecutemos la herramienta KLEE se creará un directorio donde se guardarán varios ficheros relacionados con la ejecución que ha realizado. Este directorio es nombrado por defecto como **klee-out-0**; si dicho directorio existe se nombrará como **klee-out-1** y así sucesivamente.

En primer lugar, siempre que ejecutemos KLEE se generarán estos ficheros globales estándar:

1. **info:** Este fichero de texto contiene información relacionada con la ejecución de KLEE. En concreto, almacena con qué comando se ejecuto KLEE y el tiempo total de dicha ejecución. Por ejemplo:

```
sandra@sandra:~/Escritorio/TFG/KLEE/klee-out-0$ cat info
klee get_sign.bc
PID: 12967
Using monotonic steady clock with 1/1000000000s resolution
Started: 2020-08-05 19:39:50
BEGIN searcher description
<InterleavedSearcher> containing 2 searchers:
RandomPathSearcher
WeightedRandomSearcher::CoveringNew
</InterleavedSearcher>
END searcher description
Finished: 2020-08-05 19:39:50
Elapsed: 00:00:00
KLEE: done: explored paths = 3
KLEE: done: avg. constructs per query = 14
KLEE: done: total queries = 3
KLEE: done: valid queries = 0
KLEE: done: invalid queries = 3
KLEE: done: query cex = 3

KLEE: done: total instructions = 32
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
```

Figura 3.3: Salida por terminal del fichero de texto info.

2. **warnings.txt:** Este fichero de texto contiene todas las advertencias comunicadas por KLEE durante la ejecución.

3. **messages.txt:** Este último fichero de texto contiene el resto de mensajes omitidos por KLEE.

4. **assembly.ll:** Este fichero contiene una versión legible para personas del lenguaje intermedio de LLVM ejecutado por KLEE.

5. **run.stats:** Este fichero contiene varias estadísticas emitidas por KLEE. Para poder leer este fichero hay que hacer uso de la herramienta **klee-stats**.

6. **run.istats:** Este es un fichero binario que contiene estadísticas globales emitidas por KLEE por cada linea de código del programa.

Además de los ficheros globales estándar que hemos visto, KLEE también generará los siguientes ficheros:

1. **all-queries.kquery:** Este fichero contiene todas las consultas que KLEE ha hecho durante la ejecución con el formato KQuery. La generación de este fichero se puede activar especificando la opción en KLEE con **- -use-query-log=all:kquery**.

2. **all-queries.smt2:** Este fichero contiene todas las consultas que KLEE ha hecho durante la ejecución con el formato SMT-LIBv2. Contiene la misma información que el fichero **all-queries.kquery**. La generación de este fichero se puede activar especificando la opción en KLEE con **- -use-query-log=all:smt2**.

3. **solver-queries.kquery:** Este fichero contiene todas las consultas pasadas al solucionador durante la ejecución de KLEE con el formato KQuery. La generación de este fichero se puede activar especificando la opción en KLEE con **- -use-query-log=solver:kquery**.

4. **solver-queries.smt2:** Este fichero contiene todas las consultas pasadas al solucionador durante la ejecución de KLEE con el formato SMT-LIBv2. Contiene la misma información que el fichero **solver-queries.kquery**. La generación de este fichero se puede activar especificando la opción en KLEE con **- -use-query-log=solver:smt2**.

Por último, KLEE genera unos ficheros específicos por cada *N* camino explorado en la ejecución simbólica.

1. **test<N>.ktest:** Este fichero contiene el caso de prueba generado por KLEE en el N-ésimo camino. La generación de los ficheros **.ktest** puede desactivarse usando la opción **- -no-output**.

2. **test<N>.<error-type>.err:** Este fichero es generado en los caminos donde KLEE encuentra algún error. Contiene información acerca del error de manera textual.

3. **test<N>.kquery:** Este fichero contiene las restricciones asociadas con la ruta N en el formato KQuery. La generación de estos ficheros puede ser activada usando **- -write-kqueries**.

4. **test<N>.cvc:** Este fichero contiene las restricciones asociadas a la ruta N en el formato CVC. Contiene la misma información que el fichero **.kquery**. La generación de estos ficheros puede ser activada usando **- -write-cvcs**.

5. **test<N>.smt2:** Este fichero contiene las restricciones asociadas a la ruta N en el formato SMT-LIBv2. Contiene la misma información que los dos ficheros anteriores (**.kquery** y **.cvc**). La generación de estos ficheros puede ser activada usando **- -write-smt2s**.

3.2.6. Herramientas auxiliares

KLEE también dispone de herramientas auxiliares que nos permite obtener más información sobre la ejecución.

ktest-tool

KLEE puede ser configurado para generar ficheros de salida del tipo **.ktest** cuando encuentre un error, cubre código nuevo o termina un camino de

ejecución. El contenido de este tipo de fichero describe la entrada necesaria en el programa para guiar la ejecución a la correspondiente ruta de ejecución. Sabiendo esto, **ktest-tool** es un *script* que nos permite transformar el contenido de estos ficheros en formatos legibles por humanos; por ejemplo:

```
sandra@sandra:~/Escritorio/TFG/KLEE$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
```

Figura 3.4: Ejemplo de la herramienta ktest-tool.

klee-stats

klee-stats es un *script* de Python que se utiliza para extraer y presentar en forma de tabla las estadísticas de una ejecución de KLEE. Las estadísticas de ejecución pueden incluir:

- El número de instrucciones ejecutadas.
- El porcentaje de cobertura de instrucciones en el código binario LLVM.
- El porcentaje de cobertura de cada rama de ejecución en el código binario LLVM.
- El total de instrucciones estáticas en el código binario LLVM.
- El número actual de estados activos.
- La cantidad de *megabytes* de memoria actualmente en uso.
- El número de consultas lanzadas al solucionador SMT.
- La cantidad promedio de construcciones por consulta.
- Varias estadísticas de tiempo (tiempo total, tiempo utilizados...)

klee-stats extrae la información estadística del fichero **run.stats** visto en la sección anterior (3.2.5).

El uso exacto de ***klee-stats*** es el siguiente:

klee-stats [options] directories

El parámetro *directories* es una lista de **klee-out-N** directorios creados por KLEE. Normalmente se simplifica la ejecución de ***klee-stats*** haciendo uso de ***klee-last*** (el último directorio creado) en la opción *directories*.

Con respecto a las opciones, para poder limitar sólo la información con respecto a los tiempos mostrados, podemos usar:

- -print-rel-times: Esta opción muestra los valores de tiempo relativos al tiempo de ejecución.

- -print-abs-times: Esta otra opción muestra valores de tiempo absolutos.

Otra opción disponible es **- -precision** que puede usarse para configurar el número de decimales mostrado en los valores. Por defecto, se muestran solo dos decimales pero puede caber la necesidad de necesitar más valores, por ejemplo, en parámetros muy pequeños en los que necesitemos más precisión.

Existen otras opciones que pueden utilizarse para especificar qué valores son mostrados y cómo son mostrados. Para obtener más información sobre las opciones disponibles, se puede consultar con el comando:

klee-stats –help

3.2.7. Ejecución de los casos de prueba

Una vez leídos los ficheros **.ktest** con la herramienta auxiliar **ktest-tool**, podríamos ejecutar nuestro programa a mano con los casos de prueba generados pero existe una librería que nos facilita esta tarea. La librería proporcionada por KLEE **libkleeRuntest** reemplaza la llamada **klee_make_symbolic()** con una llamada a una función que asigna a su entrada el valor almacenado en el archivo **.ktest**. Para usarla, simplemente hay que enlazar nuestro programa con la librería y configurar la variable de entorno **KTEST_FILE** para que apunte al nombre del caso de prueba deseado.

Por ejemplo, si deseamos ejecutar los casos de prueba para el ejemplo anterior **get_sign()**, debemos ejecutar las siguientes sentencias:

```
sandra@sandra:~$ export LD_LIBRARY_PATH=/home/sandra/klee/build/lib/:$LD_LIBRARY_PATH
sandra@sandra:~$ gcc -L /home/sandra/klee/build/lib/ get_sign.c -lkleeRuntest
get_sign.c: In function ‘main’:
get_sign.c:12:2: warning: implicit declaration of function ‘klee_make_symbolic’
  klee_make_symbolic(&a, sizeof(a), "a");
  ^~~~~~~~~~~~~~~~~~
sandra@sandra:~$ KTEST_FILE=klee-last/test000001.ktest ./a.out
sandra@sandra:~$ echo $?
0
sandra@sandra:~$ KTEST_FILE=klee-last/test000002.ktest ./a.out
sandra@sandra:~$ echo $?
1
sandra@sandra:~$ KTEST_FILE=klee-last/test000003.ktest ./a.out
sandra@sandra:~$ echo $?
255
```

Figura 3.5: Uso de la librería **libkleeRuntest**.

3.3 El uso de KLEE para la refutación de contratos software

Como hemos podido observar, KLEE nos ofrece generar casos de prueba automáticamente indicando qué variables queremos marcar como simbólicas y asumir ciertas precondiciones para generar dichos casos. También nos indica qué posibles valores podemos usar como datos para poder acceder a cada camino de ejecución y poder ejecutar el programa usando dichos datos mediante una librería que nos proporciona la propia herramienta.

Con esta gran posibilidad que nos ofrece la herramienta KLEE podremos usar un programa en C junto su contrato para averiguar qué elementos se puede marcar como falsos. Esto lo haremos ejecutando simbólicamente nuestro programa C del contrato en KLEE donde usaremos sus funciones para indicar ciertas precondiciones - que forman parte del contrato - usando la función **klee_assume()**. Con estas precondiciones crearemos casos de pruebas y comprobaremos la herramienta KLEE es capaz de probar que el consecuente falso.

A continuación, vamos a hacer uso del fragmento **insert()** para ilustrar el uso, paso a paso, de la herramienta KLEE para el proceso descrito.

Como acabamos de exponer, dado un fragmento de código, si deseamos generar casos de prueba, primero, hay que marcar como simbólicas las variables de entrada que deseemos. En nuestro ejemplo, hemos elegido como simbólico el elemento a insertar, la capacidad de la estructura y los elementos dentro de la estructura. Para ello hemos hecho uso de la variable **klee_make_symbolic()** (Véase en el apartado 3.2.3):

```
1 /* VARIABLES SIMBOLICAS */
2
3 klee_make_symbolic(&capacidad, sizeof(capacidad), "capacidad");
4
5 klee_make_symbolic(&elementoInsertar, sizeof(elementoInsertar), "
    elementoInsertar");
6
7 klee_make_symbolic(&lista,sizeof(lista),"lista");
```

Código 3.6: Marcación de la variables como simbólicas.

Teniendo estos tres elementos ya señalados para la ejecución, podríamos llamar a KLEE pero vamos a aprovechar la funcionalidad que nos ofrece la herramienta para asumir ciertas precondiciones (pertenecientes al contrato software de nuestro fragmento de código) y así descubrir los axiomas que son claramente falsos. Por consiguiente, vamos a mostrar dos axiomas candidatos para realizar esta tarea.

El axioma es muy sencillo y es claramente falso debido a que si la estructura del programa es igual al valor nulo, el método **insert()** no modificaría nada y todo se mantendría como está. En vista de esto, vamos a usar dicho axioma para poder demostrar cómo podemos llegar a falsificar parte de un contrato software usando la herramienta KLEE.

$$(\; isnull(s) = 1 \;) \Rightarrow \begin{pmatrix} isnull(s') = 1 \;\; \wedge & isempty(s') = 0 \; \wedge \\ contains(s', x) = 1 \;\; \wedge & \\ length(s') = length(s) + 1 \;\; \wedge \;\; ret = 1 & \end{pmatrix}$$

Figura 3.6: Primer axioma candidato del método **insert()**.

En primer lugar, usaremos la precondición de que la estructura es igual al valor nulo usando el método observador de la siguiente forma, $isnull(s) = 1$. En KLEE esto se transformará en una asunción que tomara la herramienta, para generar los distintos casos de prueba, usando el método **klee_assume()** (Véase en la sección 3.2.3):

```
/* PRECONDICION */

klee_assume(isnull(listaElementos) == 1);
```

Código 3.7: Creación de una precondición en KLEE.

Toda la preparación realizada en los apartados anteriores para poder ejecutar nuestro programa C en KLEE quedaría estructurada de la siguiente forma: la parte central del programa con los métodos y constructores; junto al método principal donde encontraremos las variables que hemos marcado como simbólicas, la precondición y una llamada al método modificador del programa.

```
/* METODOS CODIGO */
        ...
/* MAIN CODIGO */
int main(){
    int capacidad;
    int elementoInsertar;
    int lista[3];

    /* VARIABLES SIMBOLICAS */
    klee_make_symbolic(&capacidad, sizeof(capacidad), "capacidad");
    klee_make_symbolic(&elementoInsertar, sizeof(elementoInsertar), "
        elementoInsertar");
    klee_make_symbolic(&lista,sizeof(lista),"lista");

    struct set *listaElementos = new(capacidad);

    memcpy(listaElementos->elem,lista,sizeof(lista));

    /* PRECONDICION */
    klee_assume(isnull(listaElementos) == 1);

    /* MODIFICADOR */
    return insert(listaElementos,elementoInsertar);
}
```

Código 3.8: Fragmento de código preparado para la utilización de **insert().** (A) en KLEE

En segundo lugar, una vez ya preparado el programa con las variables simbólicas y las precondiciones necesarias, será el turno de compilar el archivo que

contiene el programa, para ver si hemos cometido algún error o no y poder proseguir con la utilización de la herramienta KLEE. Para realización de esta tarea vamos a usar el comando *clang*, explicado detalladamente en la Sección 3.2.4:

clang -I ../../include -emit-llvm -c -g insert.c

Una vez insertado el comando, se creará (si no ha habido un error en tiempo de compilación) un fichero con el código compilado. Este fichero se creará en el mismo directorio donde estemos ejecutando KLEE y recibirá el mismo nombre que el archivo C usado para la compilación (en este casos quedaría como **insert.bc**). Con dicho fichero será con el que ejecutemos debidamente la herramienta con el comando **klee** (sección 3.2.4), como podemos ver en el siguiente ejemplo:

```
sandra@sandra:~/Escritorio/TFG/KLEE$ klee insertSandra.bc
KLEE: output directory is "/home/sandra/Escritorio/TFG/KLEE/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: ERROR: insertSandra.c:135: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 258
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

Figura 3.7: Salida por consola de la ejecución del archivo compilado mediante KLEE.

Después de ejecutar el comando, KLEE ya nos esta avisando que la expresión que estamos asumiendo probablemente sea falsa, es decir, que nunca se dé el caso de que nuestra estructura sea igual al valor nulo por la manera en que está desarrollado el programa. Para observar mejor este hecho vamos a leer cual ha sido el único caso de prueba que se ha generado y ejecutado después con la librería libkleeRuntest (véase 3.2.7).

```
sandra@sandra:~/Escritorio/TFG/KLEE$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['insertSandra.bc']
num objects: 2
object 0: name: 'elementoInsertar'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
object 1: name: 'lista'
object 1: size: 12
object 1: data: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
object 1: hex : 0x000000000000000000000000
object 1: text: ............
```

Figura 3.8: Lectura del único caso generado para **insert.c** y la precondición $isnull(s) = 1$.

Como podemos observar se ha creado un caso de prueba un tanto extraño, ya que como hemos comentado en la sección anterior 3.2.3, no podemos crear de

forma simbólica una estructura debido a que KLEE no es capaz de hacerlo y, por lo tanto, nunca podremos asumir usando este herramienta que la estructura sea nula. Es decir, que la creación de este caso de prueba no debería haberse producido pero podemos apreciar que es incorrecto debido a que creamos una estructura y configuramos como simbólicos los elementos que hay dentro. En conclusión, cualquiera axioma configurado con la precondición $isnull(s) = 1$ y que modifique algún elemento o propiedad será falso.

Ahora vamos a realizar la ejecución del caso de prueba generado mediante la librería **libkleeRuntest**. Como podemos observar en la Figura 3.9, no es posible ejecutar el caso de prueba correctamente puesto que la herramienta nos anuncia un error en la precondición marcada.

```
sandra@sandra:~/Escritorio/TFG/KLEE$ KTEST_FILE=klee-last/test000001.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio/TFG/KLEE$ echo $?
1
sandra@sandra:~/Escritorio/TFG/KLEE$ cat klee-last/test000001.user.err
Error: invalid klee_assume call (provably false)
File: insertSandra.c
Line: 135
assembly.ll line: 417
State: 1
Stack:
        #000000417 in main () at insertSandra.c:135
```

Figura 3.9: Ejecución del caso generado.

En este segundo axioma, estableceremos como precondiciones que la estructura no este vacía, no sea nula, no este llena y que tenga un cierto tamaño. También tendremos en cuenta el observador **contains()** pero en este caso su valor será adquirido durante la ejecución. Entonces, si sucediera la poscondición, que podemos observar en el axioma, será porque no se ha podido insertar y esto equivaldría a que el valor $?c$ sea igual a uno. Por lo tanto, vamos a observar la generación de casos de prueba para observar el efecto que tendrá el valor de **contains** en la correctitud de este axioma.

$$\begin{pmatrix} isempty(s) = 0 \ \wedge & isnull(s) = 0 \ \wedge \\ isfull(s) = 0 \ \wedge & contains(s,x) = ?c \ \wedge \\ length(s) = l & \end{pmatrix} \Rightarrow \begin{pmatrix} isempty(s') = 0 \ \wedge & isnull(s') = 0 \ \wedge \\ isfull(s') = 0 \ \wedge & contains(s',x) = ?c \ \wedge \\ length(s) = l \ \wedge & ret = 0 \end{pmatrix}$$

Figura 3.10: Primer axioma candidato del método **insert()**.

Para ello, vamos a establecer las precondiciones vistas en nuestro axioma anterior estableciendo que no se cumpla con ninguna de las condiciones, que tenga cierto tamaño y jugaremos con el valor que pueda devolver el método **contains(s,x)**. Esto lo haremos usando otra vez el método que nos ofrece KLEE, **klee_assume()**, como podemos observar en el siguiente fragmento de código.

```
/* PRECONDICION */
klee_assume(isemptyl(listaElementos) == 0);
klee_assume(isnull(listaElementos) == 0);
klee_assume(isfull(listaElementos) == 0);
klee_assume(length(listaElementos,elementoInsertar) == 3);
```

Código 3.9: Creación de una precondición en KLEE.

Una vez ya preparadas las precondiciones, vamos a ejecutar KLEE con la misma preparación anterior (Código 3.8) pero modificando los **klee_assume** anteriores por estos nuevos.

```
int main(){
    int capacidad;
    int elementoInsertar;
    int lista[3];

    /* VARIABLES SIMBOLICAS */
    klee_make_symbolic(&capacidad, sizeof(capacidad), "capacidad");
    klee_make_symbolic(&elementoInsertar, sizeof(elementoInsertar), "
        elementoInsertar");
    klee_make_symbolic(&lista,sizeof(lista),"lista");

    struct set *listaElementos = new(capacidad);

    memcpy(listaElementos->elem,lista,sizeof(lista));

    /* PRECONDICION */
    klee_assume(isemptyl(listaElementos) == 0);
    klee_assume(isnull(listaElementos) == 0);
    klee_assume(isfull(listaElementos) == 0);
    klee_assume(length(listaElementos,elementoInsertar) == 3);

    /* MODIFICADOR */
    return insert(listaElementos,elementoInsertar);
}
```

Código 3.10: Creación de una precondición en KLEE.

Para diferenciar los diversos casos de pruebas generados según el valor de **contains** vamos a realizar dos ejecuciones por separado con una precondición extra. Para la primera ejecución, añadiremos la precondición **klee_assume(contains(listaElementos,x) == 0)** para generar los casos posibles cuando el elemento no está contenido; y para la segunda ejecución, **klee_assume(contains(listaElementos,x) == 1)** para generar los casos cuando el elementos esta contenido.

Casos de prueba generados si el elemento no está contenido

Vamos a empezar con la ejecución del problema asumiendo que el elemento a insertar no está previamente en la estructura. Para ello añadiremos la función mencionada anteriormente a el fragmento de código 3.10.

Una vez añadido, compilaremos y ejecutaremos el archivo como ya hemos mencionado otras veces durante este Capítulo.

Observando la Figura 3.11, vemos que se han creado diez casos de prueba en los que variará el elemento, la capacidad de la estructura y la lista de elementos dentro de la estructura, pero con la característica de que el elemento nunca estará en la estructura y que no habrá ningún impedimento para la inserción.

```
sandra@sandra:~/Escritorio$ klee noContiene.bc
KLEE: output directory is "/home/sandra/Escritorio/klee-out-8"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: ERROR: noContiene.c:131: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 1261
KLEE: done: completed paths = 13
KLEE: done: generated tests = 10
```

Figura 3.11: Casos de prueba generados si el elemento no está contenido en la estructura.

El siguiente paso, es analizar los resultados de la ejecución de nuestro programa con estos casos de prueba. A fin de realizar está tarea, usaremos la librería ofrecida por KLEE, **libkleeRuntest** para descubrir si se cumple el axioma candidato, propuesto en la Figura 3.10, con la precondición de que el elemento no este contenido.

```
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000001.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000002.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000003.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000004.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000005.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000006.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000007.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000008.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000009.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000010.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
1
```

Figura 3.12: Ejecución de los casos de prueba si el elemento no está contenido en la estructura.

Como podemos en la Figura 3.12, de los diez casos creados, KLEE ha fallado en los dos primeros, pero los ocho siguientes casos han funcionado correctamente. Por ello, solo vamos a tener en cuenta estos ocho casos.

Mirando los resultados de la ejecución, se corrobora que para las precondiciones tomadas y asumiendo que no va estar contenido el elemento, siempre se insertará el elemento en la estructura. Esto nos demuestra que en el axioma candidato propuesto, 3.10, si asumimos la precondición **contains(s,x) = 0** nunca se dará la postcondición propuesta, ya que deseamos que el resultado sea igual a cero (**ret = 0**) y hemos comprobado usando KLEE, que nunca se dará esta postcondición, es decir, que el axioma candidato propuesto es falso.

Casos de prueba generados si el elemento está contenido

Para este segundo caso, vamos a realizar los mismos pasos realizados para el caso anterior pero, esta vez, asumiendo que el elemento que queremos insertar estará en la estructura.

En primer lugar compilaremos y ejecutaremos el programa con esta nueva precondición. El resultado de dicha ejecución se muestra en la Figura 3.14 que nos muestra que se han generado cincuenta casos de prueba pero nos advierte que en algunos casos KLEE ha fallado para asumir ciertas condiciones.

```
sandra@sandra:~/Escritorio$ klee Contiene.bc
KLEE: output directory is "/home/sandra/Escritorio/klee-out-10"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: ERROR: Contiene.c:131: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 3425
KLEE: done: completed paths = 50
KLEE: done: generated tests = 50
```

Figura 3.13: Ejecución de los casos de prueba si el elemento no está contenido en la estructura.

Si miramos el directorio creado durante la ejecución, podemos visualizar cuales son los casos de prueba que tenemos que tener en cuenta para dictaminar si el axioma candidato es correcto y cuales han fallado. En la siguiente imagen podemos ver que la mayoría de casos se han generado de forma correcta.

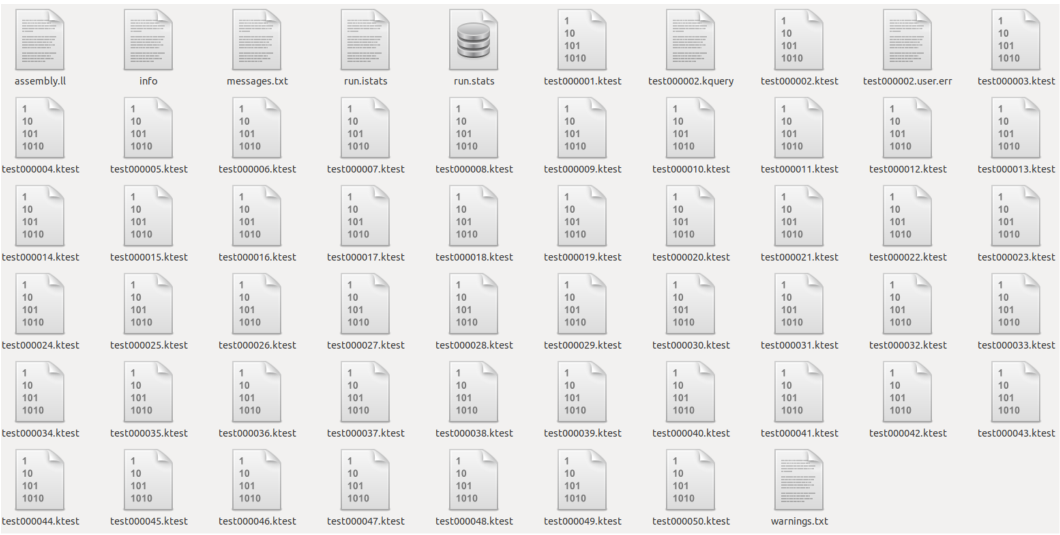


Figura 3.14: Ficheros ejecutados por KLEE.

Finalmente, comprobaremos el resultado de cada uno de los casos generados ejecutando el programa con los valores de entrada obtenidos en la generación de pruebas. Con este fin, haremos uso, una vez más, de la librería ofrecida por KLEE para la ejecución de los casos de prueba generados.

```
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000001.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000003.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000004.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000005.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000006.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000007.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000008.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000009.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000010.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000011.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000012.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000013.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000014.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000015.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
```

Figura 3.15: Resultados si se contiene el elemento parte 1.

```
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000016.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000017.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000018.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000019.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000020.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000021.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000022.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000023.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000024.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000025.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000026.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000027.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000028.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000029.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
```

Figura 3.16: Resultados si se contiene el elemento parte 2.

```
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000030.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000031.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000032.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000033.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000034.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000035.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000036.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000037.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000038.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000039.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000040.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000041.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000042.ktest ./a.out
sandra@sandra:~/Escritorio$ echo $?
0
```

Figura 3.17: Resultados si se contiene el elemento parte 3.

```
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000043.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000044.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000045.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000046.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000047.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000048.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000049.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000050.ktest ./a.out
KLEE_RUN_TEST_ERROR: invalid klee_assume
sandra@sandra:~/Escritorio$ KTEST_FILE=klee-last/test000051.ktest ./a.out
KLEE-RUNTIME: unable to open .ktest file
```

Figura 3.18: Resultados si se contiene el elemento parte 4.

En las Figuras 3.15, 3.16, 3.17 y 3.18, están ilustrados todos los resultados de las ejecuciones del programa **insert** para la precondicón de que el elemento este contenido en la estructura.

Como ya íbamos anunciando, hay ciertos casos que no se han generado correctamente mostrando como resultado el mensaje **KTEST_RUN_TEST_ERROR**, pero por suerte, solamente es un grupo reducido de casos.

Con respecto a los test generados correctamente, podemos visualizar que en cada uno de ellos el resultado de la inserción ha sido cero, es decir, que no se ha insertado ningún elemento, hecho que se esperaba.

En pocas palabras, para el caso en que contemplamos que el axioma candidato contenga la precondición de que **contains(s,x) = 0** no podemos demostrar, en este punto, que sea falso. Para poder validar este axioma se necesitarían de otras técnicas de abstracción.

Como resultado final, el hacer todo este proceso usando la herramienta KLEE nos permite detectar fácilmente qué axiomas candidatos de nuestro contrato software son incorrectos. Por tal razón, nuestro proyecto se ha dedicado a entender bien toda la funcionalidad que nos ofrece la herramienta KLEE y poder así, automatizar la mayor parte de este proceso para poder validar automáticamente contratos software, ya que usando solo la herramienta KLEE es un proceso arduo, intrincado y lento, sobre todo, si queremos aprovechar la funcionalidad de KLEE en programas más complejos.

CAPÍTULO 4
Análisis del problema

Como se ha expuesto en los Capítulos anteriores, nuestro propósito es desarrollar una herramienta cuya funcionalidad principal es validar de forma automática contratos software pertenecientes a un programa en C, encontrando asertos que son claramente falsos. Para realizar esta tarea, junto a nuestro programa a desarrollar haremos uso de la herramienta KLEE.

Debido a esto, se ha llevado a cabo un análisis del problema donde se ha estudiado diferentes puntos de vista para poder obtener las necesidades de nuestro producto así como la viabilidad de la misma para poder desarrollarla y usarla junto a la herramienta de generación de casos de pruebas KLEE.

Para ello, en el siguiente Capítulo, se expone el análisis de requisitos funcionales y no funcionales, la identificación y análisis de las posibles soluciones, y finalmente, la solución elegida.

4.1 Análisis de requisitos

Vamos a empezar nuestro análisis del problema con el análisis de requisitos de nuestra aplicación. En este análisis vamos a definir las necesidades, los servicios y restricciones de nuestra herramienta definiendo los requisitos funcionales y no funcionales.

4.1.1. Requisitos funcionales

Para poder detectar los requisitos funcionales necesarios, hemos hecho uso de un diagrama de casos de uso. Como podemos ver en la siguiente figura, disponemos de dos actores: el usuario y la herramienta KLEE (externa a nuestro sistema). Dichos actores se encargarán de realizar las acciones descritas en la figura y que podemos ver explicadas con más detalle en las tablas posteriores.

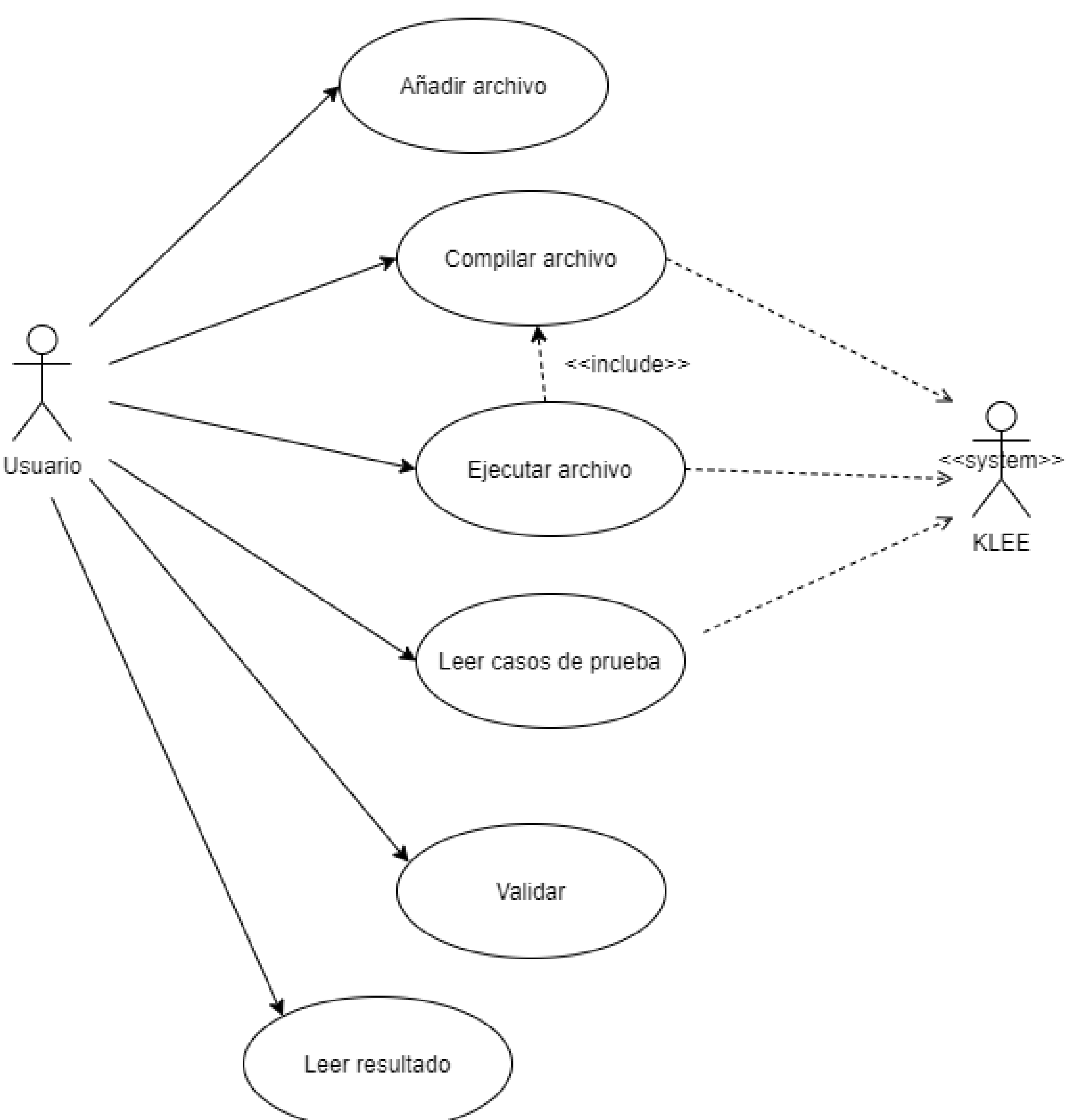


Figura 4.1: Diagrama de casos de uso de la herramienta desarrollada.

Como podemos ver en la Figura 4.1 en nuestro programa solo se contemplan seis funcionalidades, de las cuales tres de ellas implican usar la herramienta externa KLEE. En las siguientes tablas se describen cada una de ellas:

Caso de uso	Añadir archivo
Actores	Usuario
Resumen	Añadir al programa el archivo que deseemos usar como contrato
Precondiciones	Añadir un archivo existente
Postcondiciones	-
Incluye	-
Extiende	-
Hereda de	-
Flujo de eventos	
Actor	**Sistema**
1. Indicar al programa que archivo es el deseado	
	2. Comprobar la existencia de dicho archivo
	3. Si existe el archivo guardarlo en el programa
	4. Si no existe avisar al usuario con un mensaje de error

Tabla 4.1: Caso de uso Añadir archivo.

Caso de uso	Compilar archivo
Actores	Usuario y KLEE
Resumen	Compilar, mediante la herramienta KLEE, el archivo C elegido por el usuario
Precondiciones	Añadir un archivo existente
Postcondiciones	Salida del archivo compilado
Incluye	-
Extiende	-
Hereda de	-
Flujo de eventos	
Actor	**Sistema**
1. Indicar al programa que queremos compilar el archivo C	
	2. Comprobar que el archivo sea C
	3. Mandar el archivo C a la herramienta KLEE
	4. Recoger la salida de KLEE
5. Adquirir el archivo compilado por KLEE	

Tabla 4.2: Caso de uso Compilar archivo.

Caso de uso	Ejecutar archivo
Actores	Usuario y KLEE
Resumen	Ejecutar el archivo ya compilado para crear los casos de prueba mediante la herramienta KLEE
Precondiciones	Archivo compilado
Postcondiciones	Casos de prueba generados
Incluye	Compilar archivo
Extiende	-
Hereda de	-
Flujo de eventos	
Actor	**Sistema**
1. Indicar al programa que queremos ejecutar el archivo compilado para generar los casos de prueba	
	2. Comprobar si el archivo es el correcto
	3. Si es correcto redirigir el archivo a la herramienta KLEE
	4. Recoger la salida de KLEE
5. Adquirir todos los archivos relacionados con la ejecución en KLEE	

Tabla 4.3: Caso de uso Ejecutar archivo.

Caso de uso	Leer casos de prueba
Actores	Usuario
Resumen	Leer los ficheros con los distintos casos de prueba generados por KLEE
Precondiciones	Ficheros generados después de una ejecución
Postcondiciones	-
Incluye	-
Extiende	-
Hereda de	-
Flujo de eventos	
Actor	**Sistema**
1. Indicar al programa que queremos leer los casos de prueba generados	
	2. Llamar a KLEE por cada caso de prueba
	3. Transformar el dato para que sea legible
	4. Mostrar al usuario el resultado

Tabla 4.4: Caso de uso Leer casos de prueba.

Caso de uso	Validar
Actores	Usuario
Resumen	Validar el contrato software comprobando el programa con los datos obtenidos de la ejecución de los casos de prueba
Precondiciones	Casos de prueba del programa a validar
Postcondiciones	-
Incluye	-
Extiende	-
Hereda de	-
Flujo de eventos	
Actor	**Sistema**
1. Indicar al programa que deseamos la validación de nuestro programa	
	2. Recoger los datos adquiridos con los casos de prueba
	3. Ejecutar el programa con esos datos
	4. Observar la evolución del programa
	5. Si es erróneo falsear dicho contrato
	6. Si es correcto validar dicho contrato

Tabla 4.5: Caso de uso Validar.

Caso de uso	Leer resultado
Actores	Usuario
Resumen	Añadir al programa el archivo que deseemos usar
Precondiciones	Añadir un archivo existente
Postcondiciones	-
Incluye	-
Extiende	-
Hereda de	-
Flujo de eventos	
Actor	**Sistema**
1. Indicar al programa que queremos leer los resultado obtenidos	
	2. Mostrar al usuario dichos datos

Tabla 4.6: Caso de uso Leer resultado.

4.1.2. Requisitos no funcionales

Los requisitos no funcionales más notorios para la realización de este programa incluyen, en primer lugar, que nuestra herramienta pueda ejecutarse en sistemas Linux debido a que una de las herramientas que vamos a utilizar, KLEE, solo es capaz de ejecutar programas complejos en Linux.

Otro requisito no funcional sería la introducción en nuestro programa sólo de archivos escritos en C.

Finalmente, el usuario que desee usar nuestro programa requerirá de la instalación de la herramienta KLEE, debido a que nuestro programa depende de esta herramienta

4.2 Identificación y análisis de soluciones posibles

Vistas la necesidades, se nos plantea varios aspectos a tener en cuenta a la hora de elegir el lenguaje de programación, el entorno de desarrollo y el tipo de aplicación que queremos ofrecer al usuario.

A la hora de elegir qué lenguaje de programación se va a usar para el desarrollo de nuestra aplicación, se han tenido en cuenta varios lenguajes de programación estudiados durante el grado o que su utilización es de las más populares. Entre esos lenguajes están, entre otros, C, C#, Python, Java y JavaScript. Para la elección, entre los distintos lenguajes, se ha tenido en cuenta su complejidad, portabilidad, eficiencia, facilidad de aprendizaje y su uso en el diseño de interfaces. También se ha tenido en cuenta que dichos lenguajes sean aptos para su desarrollo en entornos aptos para sistemas operativos Linux. Otro aspecto bastante importante a tener en cuenta es la necesidad de poder conectar durante la ejecución del programa con la consola de Linux, ya que la herramienta KLEE funciona a través de la utilización de comandos vía terminal de Linux. Finalmente, como último aspecto a tener en cuenta, se ha tenido en cuenta el tipo de aplicación que deseemos ofrecer al usuario.

Con respecto al entorno de desarrollo a utilizar, hay que observar que se verá afectado según el lenguaje que escojamos y su portabilidad en sistemas operativos Linux. También estará afectado por el tipo de aplicación que finalmente desarrollemos.

Por último, se plantean tres tipos de aplicaciones que se verán afectadas por el uso conjunto con la herramienta KLEE: una pagina web, una aplicación de escritorio o una aplicación vía comandos por la terminal de linux.

4.3 Solución propuesta

Finalmente, la solución escogida para el desarrollo del proyecto, teniendo en cuenta todos los aspectos comentados en el apartado anterior, ha sido la siguiente:

- El lenguaje de programación elegido ha sido Java, debido a su familiaridad con él, poca complejidad y su apto uso para la utilización vía consola de la herramienta KLEE.
- Con respecto al entorno, se ha optado por la herramienta de desarrollo Eclipse (analizada en el siguiente Capítulo 5.3) ya que da soporte al lenguaje Java y es apto para entornos Linux.
- Finalmente, se ha optado por realizar una aplicación con interfaz y uso vía terminal de Linux, en vista de que la herramienta a usar funciona también a través de la terminal de Linux.

Teniendo claro ya la solución a nuestro problema, podemos pasar ya a diseñar nuestra aplicación para poder realizar posteriormente su desarrollo.

CAPÍTULO 5
Diseño de la solución

En este Capítulo detallamos el diseño que hemos seguido para implementar nuestra herramienta de validación automática de contratos software y la tecnología utilizada para su desarrollo.

5.1 Arquitectura del Sistema

El programa se ha desarrollado siguiendo el patrón software de **Arquitectura basada en capas**, en el cuál cada componente del sistema software queda desacoplado pudiendo retocar una capa sin interferir con las demás. Dichos componentes están clasificados en presentación, lógica y datos:

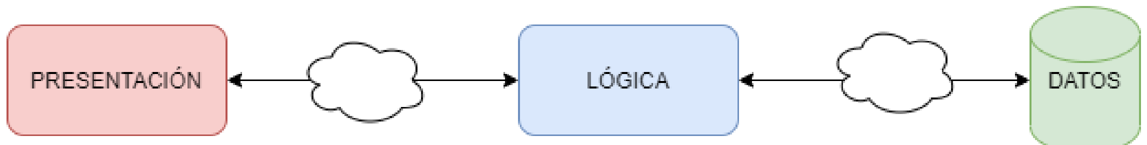


Figura 5.1: Diagrama de Arquitectura por capas.

- La capa de **presentación** es la única capa que puede verse y comunicarse con el usuario, es decir, en esta capa está representada lo que conocemos como interfaz gráfica. Esta capa comunica información y captura toda interacción del usuario para comunicarla a la capa de lógica que se encargará de procesarla y comunicar nueva información.
- En la capa de **lógica** localizaremos todas las funciones principales de nuestra aplicación. Es la capa más importante, encargada de recibir las peticiones del usuario y ejecutarlas mediante la comunicación con la capa de presentación; así como de solicitar y gestionar la base de datos mediante la comunicación con la capa de datos.
- Finalmente, en la capa de **datos** encontraremos almacenados los datos del programa y podremos acceder a ellos. Está constituida por uno o más gestores de bases de datos que se encargarán de gestionar la base de datos (almacenar y recuperar) según las peticiones recibidas desde la capa de negocio.

Por otra parte, para poder implementar bien las conexiones entre las capas vamos a hacer uso del patrón de diseño *Singleton* o Instancia Única [10]. Este patrón permite asegurar que sólo se va a crear una instancia de una clase y suministra solo un acceso global a la misma.

Singleton

- instance: Singleton

- Singleton()

+ getInstance(): Singleton

Figura 5.2: Implementación del patrón *Singleton*.

En la Figura 5.2 podemos observar la implementación de este método, donde haremos privado el constructor de la clase,**Singleton()**, para evitar que se creen objetos de la clase sin ningún control; y un método estático de acceso o creación que actúe como constructor, **getInstance()**.

En nuestro proyecto implementaremos el patrón de instancia única en la capa de presentación y en la capa de datos para poder conectar únicamente con la capa de lógica a través de esta instancia. Esto nos facilitará el desacoplamiento entre las capas pudiendo modificarlas sin afectar a los otros componentes.

5.2 Diseño Detallado

Como hemos adelantado, para organizar el programa, hemos basado todo el diseño en la arquitectura de tres capas, mencionada anteriormente.

El producto software está formado por: una interfaz de línea de comandos a la que se accede a través de la consola de mandos; una parte con toda la funcionalidad de nuestra aplicación, que será la encargada de comunicar todos los componentes del sistema; y finalmente, un gestor de archivos en el que almacenaremos y leeremos todos los archivos generados durante la ejecución de la herramienta desarrollada. Estos archivos contendrán información relevante como los resultados obtenidos y los datos de la generación de casos de pruebas.

El esquema siguiente ilustra está organizado el proyecto. En primer lugar, como podemos ver, tendremos como elemento principal la terminal de comandos. Con ella podremos ejecutar nuestra aplicación y, a través de ésta, acceder a la herramienta KLEE, que también funciona vía uso de comandos. Por otra parte, tendremos todos los archivos necesarios para la ejecución y todos los archivos generados después de la ejecución.

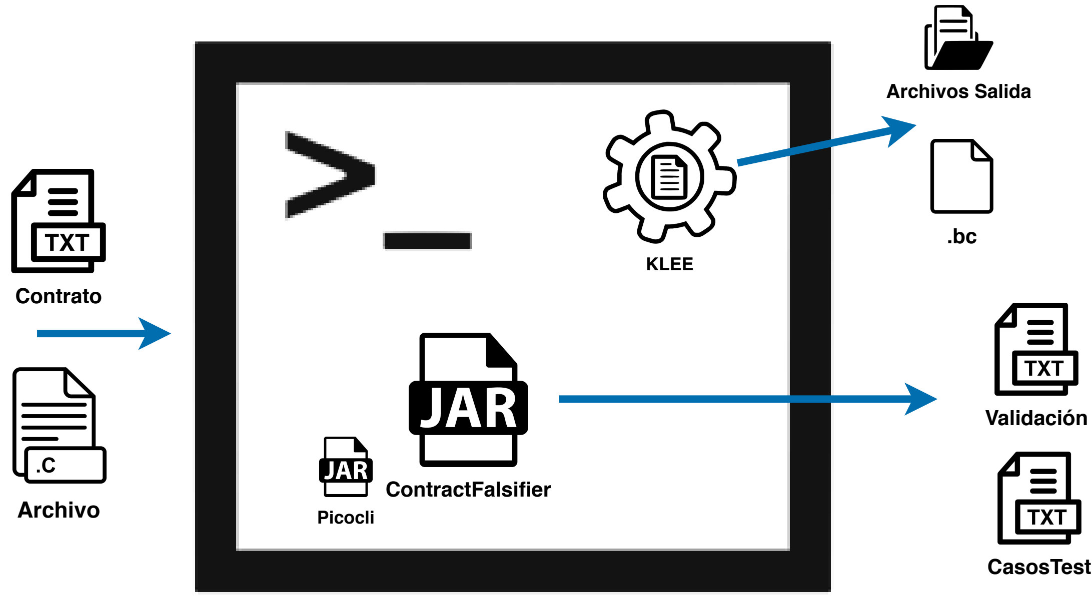


Figura 5.3: Esquema global de nuestra aplicación.

Una vez explicado este esquema vamos a mostrar en detalle la estructura del sistema. Dicha estructura está separada en tres paquetes que representarán las capas mencionadas, presentación, lógica y datos.

En el primer paquete, dispondremos de todos los elementos relacionados con la interfaz de comandos. Estos se organizan en cuatro clases que representarán los comandos que podemos ejecutar en la terminal para interactuar con nuestro programa. En una clase se implementan el patrón instancia única y también se ha desarrollado una interfaz necesaria para el funcionamiento de los comandos.

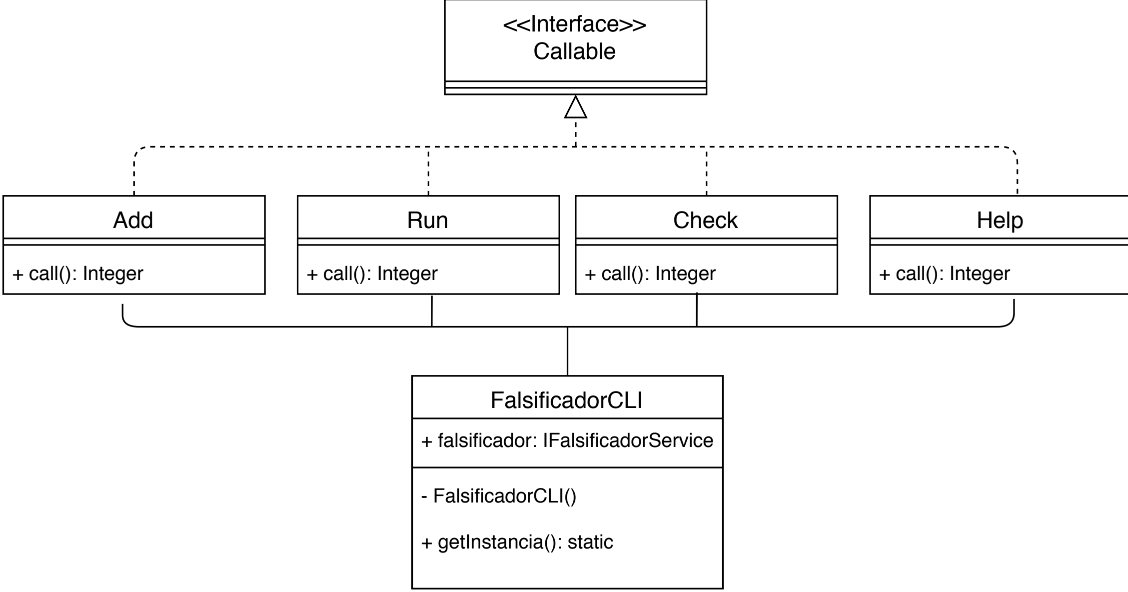


Figura 5.4: Diagrama de clases de la capa de presentación.

En el segundo paquete establecemos las clases necesarias para ejecutar la funcionalidad requerida. En este caso tendremos dos clases de las que parte de su

funcionalidad estará relacionada con la herramienta KLEE. Y, por otro lado, una interfaz que hará de puente con las otras capas (y nos facilitará el desacoplamiento permitiéndonos poder modificar en todo momento las distintas funcionalidades sin que afecte a las otras capas) y una clase que la implementará.

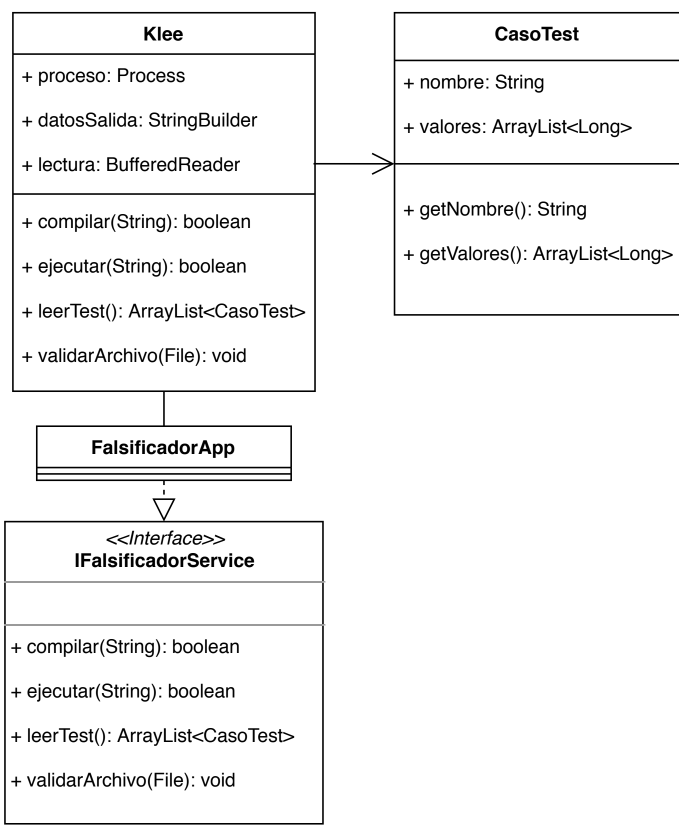


Figura 5.5: Diagrama de clases de la capa de lógica.

Por último, en el tercer paquete tenemos la funcionalidad necesaria para gestionar los archivos necesarios para el funcionamiento de nuestra aplicación. Esta funcionalidad estará recogida en una misma clase. También se ha construido una clase con la implementación del patrón instancia única que conectará con la capa de lógica.

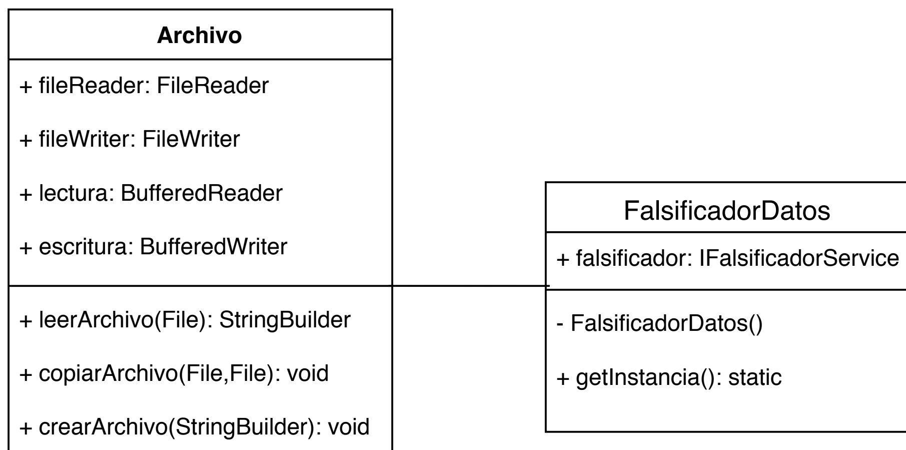


Figura 5.6: Diagrama de clases de la capa de datos.

5.3 Tecnologías Utilizadas

Para empezar, hemos desarrollado nuestro programa para su ejecución en sistemas operativos Linux, ya que como hemos comentado en el apartado 3.2.2 la herramienta KLEE solo funciona correctamente en dicho sistemas y utiliza la consola de Linux para ejecutarse.

El lenguaje de programación utilizado ha sido Java [11], debido a la gran familiaridad con este lenguaje y su poca complejidad. Para su desarrollo hemos usado la plataforma de desarrollo software Eclipse [8], junto la librería PICOCLI [16] que nos ha permitido crear varios comandos en Java para su uso en la terminal de Linux.

Finalmente hemos utilizada la herramienta KLEE, analizada ya en Capítulos anteriores 3, que formará parte clave en la funcionalidad y desarrollo de nuestro programa.

CAPÍTULO 6
Desarrollo de la solución

Durante este Capítulo vamos a describir los pasos seguidos para la realización del desarrollo de la solución descrita en los Capítulos de Análisis y Diseño, vistos anteriormente.

Observando las Figuras 6.1 y 6.2, podemos apreciar la estructura modular del proyecto destacando los tres paquetes que representarán las tres capas descritas con anterioridad: presentación, lógica y persistencia o datos. También dispondremos de un paquete con la librería PICOCLI que, como hemos comentado antes, nos servirá para implementar los comandos en Java.

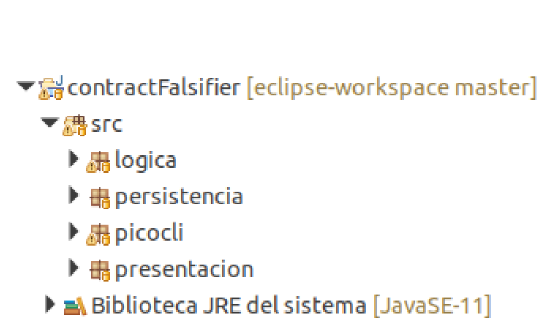


Figura 6.1: Detalle de la estructura de *ContractFalsifier*.

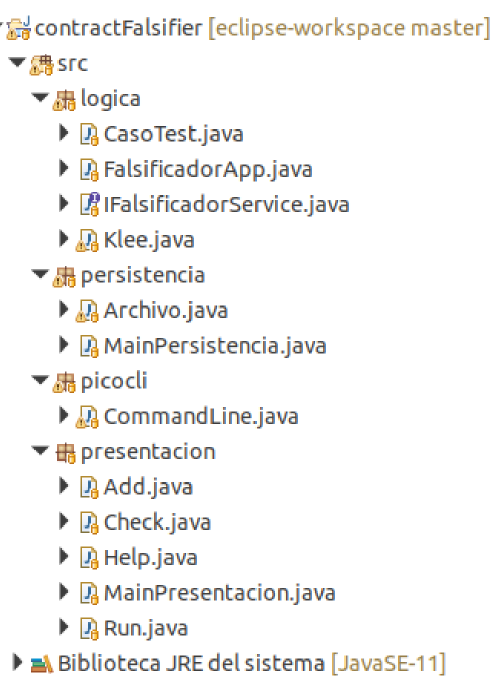


Figura 6.2: Detalle de la estructura y clases de *ContractFalsifier*.

Conforme a ello, vamos a explicar en las siguiente secciones el desarrollo de cada uno de los elementos diseñados para la aplicación propuesta.

6.1 Desarrollo de la interfaz de línea de comandos

Durante el análisis de nuestra herramienta, establecimos que la interfaz de nuestra aplicación se desarrollaría procurando que se utilizaran comandos para acceder a la terminal de Linux debido a que la herramienta KLEE se ejecuta también usando la línea de comandos.

Nuestro propósito ha sido crear varios comandos que envuelvan las seis funcionalidades que dispone nuestro programa.

Cada uno de los comandos se ha desplegado en una clase distinta. En total se han creado cuatro comandos, *Add*, *Run*, *Check* y *Help*, representados en cuatro clases. Con el comando *Add* accederemos a la funcionalidad de añadir el contrato, que estará el código preparado para la ejecución con KLEE, y el archivo C a nuestro programa y guardar una copia. Con *Run* activaremos la compilación, generación de casos de pruebas, lectura y guardado de los casos de prueba. Con *Check* accionaremos la ejecución de los casos de prueba y validación según los datos obtenidos. Y finalmente en *Help*, accederemos a una ayuda del uso de los comandos.

Para poder realizar su implementación, hemos usado PICOCLI y con el uso de sus librerías se han creado los comandos al que hemos añadido ciertas opciones. Por ejemplo en el caso de *Add*:

En primer lugar declaramos el nombre que deseamos ponerle al comando y le atribuimos una descripción.

```
@Command(
    name = "Add",
    description = "add archivo"
)
```

Código 6.1: Comando para la creación de *Add*.

En segundo lugar, asignamos las opciones que deseamos al comando. En este caso se han creado dos opciones para añadir diferentes tipos de archivo, un contrato y un programa C. El fichero adquirido se guardará en la variable creada para cada opción.

Observando el siguiente código podemos ver que *Add* usando **-c** o **- -contrato** y **-a** o **- -archivo** accederemos a las opciones para seleccionar el archivo correspondiente.

```
@Option(names = {"-c", "--contrato"}, description = "Add contrato")
    private File contrato;

@Option(names = {"-a", "--archivo"}, description = "Add archivo C")
    private File archivo;
```

Código 6.2: Código pata la creación de opciones.

Por último implementamos la interfaz **Callable** en la clase para poder ejecutar dicho comando y acceder a la capa de lógica. Con ese fin, necesitaremos incluir las siguientes lineas para su funcionamiento:

```
public class Add implements Callable<Integer> {

int exitCode = new CommandLine(new Add()).execute(args);

@Override
    public Integer call() throws Exception { /* CONEXION LOGICA */}
}
```

Código 6.3: Implementación de **Callable**.

6.2 Desarrollo de la funcionalidad

Referente al desarrollo de la funcionalidad, hemos creado la clase **Klee()** (véase Figura 5.5). En ella hemos implementado toda la funcionalidad necesaria de la aplicación.

Considérese que parte de la funcionalidad de nuestra aplicación se ha desarrollado con el uso conjunto de la herramienta KLEE. Para poder conectar con la herramienta, hemos creado un proceso para conectar con la terminal de Linux y ejecutar los diversos comandos para el uso de KLEE. También leeremos los resultados obtenidos de la inserción de dichos comandos. Por ejemplo:

En el siguiente fragmento podemos ver como hemos hecho posible la compilación de un archivo en KLEE y la creación de un búfer de lectura para poder guardar los resultados obtenidos.

```
Process process = Runtime.getRuntime().exec("clang -I ../../include -
    emit-llvm -c -g " + archivo);

BufferedReader readerExito = new BufferedReader(
                    new InputStreamReader(process.getInputStream()));
```

Código 6.4: Ejecución del comando **clang** a través de Java.

Visto esto vamos a explicar de forma resumida la funcionalidad implementada en cada uno de los métodos de la clase **Klee()**.

6.2.1. Método compilar(String)

Para los siguientes tres métodos accederemos desde la llamada vía terminal con el uso del comando *Run*.

Con este método ejecutaremos el comando de compilación de la herramienta KLEE (el mismo que hemos visto en 6.4) con un archivo C preparado ya para su uso en KLEE. Este archivo será insertado desde la terminal de comandos con la opción -a o - -archivo.

Una vez compilado correctamente el archivo, KLEE generará un archivo compilado del mismo. Este archivo recibirá el mismo nombre pero será del tipo **.bc**. Enviaremos la información de este nuevo archivo a el método **ejecutar(String)**.

Si durante la ejecución del método hubiera algún error, se proporcionará la información necesaria al usuario para que pueda actuar ante el error de forma eficaz y clara.

6.2.2. Método ejecutar(String)

Una vez recibido el archivo compilado, aplicaremos el siguiente comando (como en Figura 6.4) para ejecutar la generación de casos de pruebas.

klee archivoCompilado.bc

Si todo es correcto, nos proporcionará la información de que todo ha funcionado como corresponde y se creará un directorio con todos los archivos generados con la ejecución, entre ellos, los archivos **.ktest** que contendrán los casos de prueba.

Como el método anterior, si hubiera algún problema se avisará al usuario de inmediato.

6.2.3. Método leerTest()

En este método accederemos a todos los casos de pruebas generados automáticamente con la ejecución del archivo C en KLEE. Esto lo haremos mediante la herramienta **ktest-tool** descrita en la Sección 3.2.6.

Por cada caso de prueba generado, usaremos el siguiente comando en nuestro programa para poder acceder a la lectura del archivo, como lo hemos realizado en la Figura 6.4.

ktest-tool –write-ints klee-last/test00000X.ktest

Con la lectura, nos guardaremos los siguientes datos: el número de casos de prueba, nombre de las variables y sus valores. Esta información será llevada a la capa de datos para que gestione esa información y cree un archivo para almacenarla.

```
El test 1 está constituido por:
La variable "elementoInsertar" tiene el valor 0
La variable "lista" tiene el valor [0, 0, 0]
La variable "contiene" tiene el valor 1
El test 2 está constituido por:
La variable "elementoInsertar" tiene el valor 0
La variable "lista" tiene el valor [257, 257, 0]
La variable "contiene" tiene el valor 1
```

Figura 6.3: Archivo con los casos de prueba generados.

6.2.4. Método validarArchivo(File)

Para acceder a la funcionalidad proporcionada por este método, lo haremos a través del uso del comando *Check* y las opciones correspondientes. Una vez accedido al método con el archivo a validar, usaremos la librería de KLEE , **libkleeRuntest** para ejecutar el programa con los casos generados. Para ello, nuestro método ejecutará por cada caso de prueba generado, tres comandos que nos permitirán acceder al resultado de la ejecución con los valores generados.

```
/*.......*/
while(archivoTest.exists()) {
    System.out.println("Ejecutando el test " + numeroTest);
    String[] command =
        {
            "bash",
        };

    process= Runtime.getRuntime().exec(command);
    PrintWriter stdin = new PrintWriter(process.getOutputStream());

    stdin.println("gcc -L /home/sandra/klee/build/lib/ " + archivo.
       getName() + " -lkleeRuntest");
    stdin.println("KTEST_FILE=klee-last/" + archivoTest.getName() + "
       ./a.out");
    stdin.println("echo $?");
    stdin.close();
    /*.......*/
```

Código 6.5: Código para ejecutar varios comandos en el mismo proceso.

Mientras ejecutemos los distintos comandos nos guardaremos los resultados obtenidos para comparar los distintos datos junto al contrato inicial y los valores generados en los casos de prueba.

Con toda esta información obtenida, seremos capaces de deducir que axiomas candidatos de nuestro contrato son completamente incorrectos. Por ejemplo si nuestro axioma indica que la estructura es nula, no existirá ningún caso de prueba donde nos indique que se ha podido insertar algún elemento.

6.3 Desarrollo del almacenamiento

Finalmente, en la capa de persistencia o datos, hemos creado un gestor de archivos para leer los archivos añadidos por el usuario y guardar todos los datos obtenidos durante la generación de los casos de prueba y la validación del contrato.

Este gestor se ha creado mediante el uso de los siguientes tres métodos, **leerArchivo(File)**, **copiarArchivo(File,File)** y **crearArchivo(StringBuilder)** (véase Figura 5.6).

El primer método se encargará de leer cualquier archivo entrante a nuestra aplicación; el segundo método copiará un archivo y lo añadirá a otro archivo; y el último método, se encargará de crear un nuevo archivo con la información contenida en el **StringBuilder**.

CAPÍTULO 7
Validación de la solución propuesta

En este Capítulo se procede a probar nuestro programa, *ContractFalsifier*, usando los dos ejemplos que hemos explicado con anterioridad en el Capítulo de la herramienta KLEE (véase Capítulo 3).

7.1 Validación con el archivo get_sign()

Primero vamos a probar el archivo más sencillo analizado, **get_sign()**. Con este ejemplo, solo podemos probar eficazmente los comandos *Add* y *Run*. También podemos usar el comando *Check*, pero de forma más simple, ya que este programa es muy sencillo.

En primer lugar vamos a colocar el método **get_sign** en un archivo C, mientras que toda la preparación necesaria para ejecutar el archivo en KLEE queda escrita en un archivo **txt**.

Una vez realizada esta tarea abrimos nuestra terminal y ejecutamos el comando *Add* junto a las opciones de añadir el archivo y el fichero de texto.

```
sandra@sandra:~/Escritorio$ java -cp "picocli-4.4.0.jar:contractFalsifier.jar" presentacion.Add -a get_signContrato.c -c preparacion.txt
Archivos añadidos y nuevo archivo creado correctamente
```

Figura 7.1: Ejecución del comando *Add* en **get_sign()**.

El resultado de la ejecución se muestra por la terminal y al mismo tiempo se crea el fichero con la configuración para KLEE añadida.

Figura 7.2: Ficheros en **get_sign()** para la utilización de *ContractFalsifier*.

Con esto realizado ya podemos ejecutar la primera parte de nuestro programa con el comando *Run*. Al seleccionar este comando se llama a KLEE para realizar

las tareas de compilar, generar casos de prueba y leer dichos casos que se guardarán en un fichero de texto. Todo este proceso se muestra por pantalla como podemos ver en la Figura 7.3.

```
sandra@sandra:~/Escritorio$ java -cp "picocli-4.4.0.jar:contractFalsifier.jar" presentacion.Run -a get_signContrato.c
Compilado correctamente

Se ha ejecutado correctamente

Se ha leido correctamente el caso de prueba 1
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 2
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 3
Guardando resultado en una archivo externo...
```

Figura 7.3: Ejecución de *Run* en **get_sign()**.

Por otra parte, en el fichero generado vemos el resultado de la generación de los casos de prueba donde para cada variable, marcada como simbólica, obtenemos que valor ha adquirido en cada prueba. En la Figura 7.4, está representado uno de estos ficheros, en el cual la variable **a** adquiere tres valores: el cero, un valor positivo y un valor negativo.

```
Abrir    casosTest.txt
         ~/Escritorio
El test 1 está constituido por:
La variable "a" tiene el valor 0
El test 2 está constituido por:
La variable "a" tiene el valor 16843009
El test 3 está constituido por:
La variable "a" tiene el valor -2147483648
```

Figura 7.4: Fichero con los casos de **get_sign()** *Run*.

Finalmente nos queda la comprobación con el comando *Check* que en este ejemplo no se realiza esta funcionalidad completamente, como ya hemos mencionado.

Observando la Figura 7.5 vemos la ejecución de los tres casos de prueba generados en el comando anterior. Si comparamos los dos ficheros generados 7.4 y 7.6 podemos decir que el método **get_sign()** para el valor de entrada cero su resultado será cero, para un valor positivo será uno y para un valor negativo será menos uno[1].

```
sandra@sandra:~/Escritorio$ java -cp "picocli-4.4.0.jar:contractFalsifier.jar" presentacion.Check -a get_signContrato.c
Inicio de la validación...

Ejecutando el test 1
Ejecutando el test 2
Ejecutando el test 3
COMPLETADO
Generando fichero....
```

Figura 7.5: Ejecución de *Check* en **get_sign()**.

[1]El valor 255 en KLEE, equivale a -1

```
Abrir ▾                                    ResultadosEjecucion.txt
                                                ~/Escritorio
El resultado de la ejecución del test 1 es 0
El resultado de la ejecución del test 2 es 1
El resultado de la ejecución del test 3 es 255
```

Figura 7.6: Resultados de *Check* en **get_sign()**.

7.2 Validación con el archivo insert()

Para este segundo ejemplo, vamos a probar el archivo **insert()** con el mismo contrato configurado en la Sección 3.3. Con él, comprobaremos el axioma candidato de la Figura 3.10 con la precondición de que el elemento no esté contenido, axioma que hemos visto que puede ser falseado con la herramienta KLEE.

Primero, añadiremos el contrato establecido al código **insert()**. Dicho contrato, contiene todas las precondiciones y toda la preparación necesaria para que el código original pueda funcionar con KLEE. Usando el comando *Add*, realizamos esta tarea creando un fichero nuevo.

```
sandra@sandra:~/Escritorio$ java -cp "picocli-4.4.0.jar:contractFalsifier.jar" presentacion.Add -a noContiene.c -c contrato.txt
Archivos añadidos y nuevo archivo creado correctamente
```

Figura 7.7: Ejecución del comando *Add* en **insert()**.

En las Figuras 7.7 y 7.8 podemos ver una muestra del funcionamiento del comando *Add* y el fichero resultante.

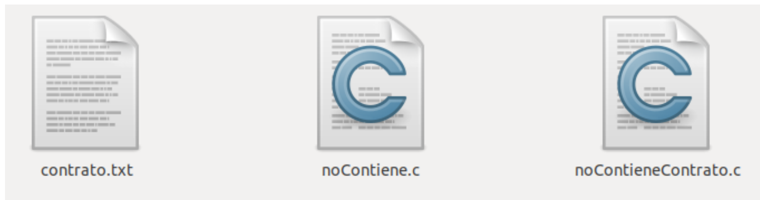


Figura 7.8: Ficheros en **insert()** para la utilización de *ContractFalsifier*.

En segundo lugar, aplicaremos al fichero generado, la funcionalidad que nos ofrece el comando *Run*. Primero compila el fichero en KLEE para después ejecutarlo, crear todos los casos de prueba y leerlos.

En la Figura 7.9 podemos ver que se han generado diez casos de prueba como en el ejemplo realizado anteriormente (Figura 3.11). También se ha generado un fichero 7.10 con la lectura de todos estos casos de prueba que nos informa por cada prueba que valor tienen las variables **elementoInsertar** y **lista**.

```
sandra@sandra:~/Escritorio$ java -cp "picocli-4.
Compilado correctamente

Se ha ejecutado correctamente

Se ha leido correctamente el caso de prueba 1
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 2
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 3
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 4
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 5
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 6
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 7
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 8
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 9
Guardando resultado en una archivo externo...
Se ha leido correctamente el caso de prueba 10
Guardando resultado en una archivo externo...
```

Figura 7.9: Ejecución de *Run* en **insert()**.

```
Abrir   casosTest.txt ~/Escritorio
El test 1 está constituido por:
La variable "elementoInsertar" tiene el valor 0
La variable "lista" tiene el valor [0, 0, 0, 0, 0, 0]
El test 2 está constituido por:
La variable "elementoInsertar" tiene el valor 0
La variable "lista" tiene el valor [257, 257, 257, 257, 257, 257]
El test 3 está constituido por:
La variable "elementoInsertar" tiene el valor 16777216
La variable "lista" tiene el valor [0, 3, 0, 0, 0, 0]
El test 4 está constituido por:
La variable "elementoInsertar" tiene el valor 50331648
La variable "lista" tiene el valor [0, 1, 0, 0, 0, 64]
El test 5 está constituido por:
La variable "elementoInsertar" tiene el valor 1
La variable "lista" tiene el valor [0, 0, 0, 0, 0, 0]
```

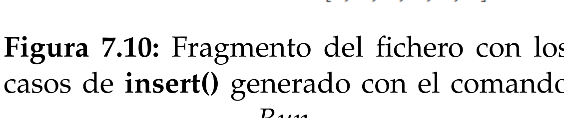


Figura 7.10: Fragmento del fichero con los casos de **insert()** generado con el comando *Run*.

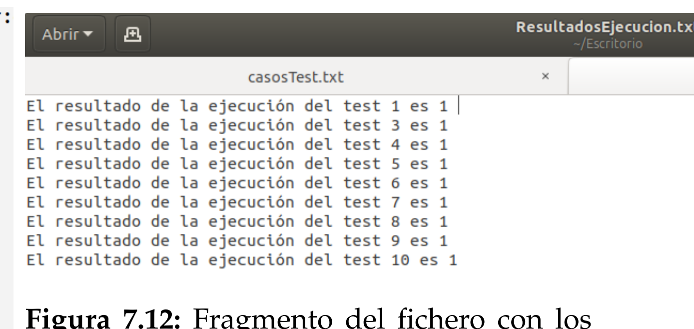
Por último, vamos a realizar la validación de nuestro contrato usando el comando *Check*. Este comando ejecutar, solamente, los casos en los que KLEE pudo generar el contrato sin ningún fallo.

En las Figuras 7.11 y 7.12 están representados los resultados obtenidos con nuestro programa. Podemos observar que para las precondiciones tomadas siempre se inserta el elemento, así que podemos concluir que el axioma candidato propuesto es claramente falso.

```
sandra@sandra:~/Escritorio$ java -cp "picocli-4.4.0.jar:
Inicio de la validación...

Ejecutando el test 1
Ejecutando el test 2
Ejecutando el test 3
Ejecutando el test 4
Ejecutando el test 5
Ejecutando el test 6
Ejecutando el test 7
Ejecutando el test 8
Ejecutando el test 9
Ejecutando el test 10
COMPLETADO
Generando fichero....
```

Figura 7.11: Ejecución de *Run* en **insert()**.



```
Abrir   ResultadosEjecucion.txt ~/Escritorio
casosTest.txt
El resultado de la ejecución del test 1 es 1
El resultado de la ejecución del test 3 es 1
El resultado de la ejecución del test 4 es 1
El resultado de la ejecución del test 5 es 1
El resultado de la ejecución del test 6 es 1
El resultado de la ejecución del test 7 es 1
El resultado de la ejecución del test 8 es 1
El resultado de la ejecución del test 9 es 1
El resultado de la ejecución del test 10 es 1
```

Figura 7.12: Fragmento del fichero con los casos de **insert()** generado con el comando *Run*.

CAPÍTULO 8
Conclusiones

Una vez finalizado el desarrollo de nuestra herramienta, *ContractFalsifier*, podemos afirmar que hemos cumplido con los cinco objetivos establecidos al comienzo de nuestro desarrollo.

Para poder lograr todos estos objetivos, primero hemos hecho un estudio exhaustivo de la herramienta de generación automática de casos de pruebas KLEE. Esta herramienta fue estudiada durante la signatura de Análisis, Validación y Depuración de software (AVD) como ejemplo práctico de la técnica de análisis mediante ejecución simbólica.

Durante su estudio, descubrimos funciones bastantes útiles como **klee_assume()**, que nos permite asumir ciertas condiciones a la hora de generar los casos de prueba. Por otra parte, tuvimos ciertos problemas con su instalación en sistemas macOs, que nos hizo darnos cuenta, que para estos sistemas la herramienta KLEE está limitada a programas muy básicos. También tuvimos problemas con el uso de las estructuras a la hora de querer marcarlas como simbólicas pero, finalmente, encontramos una solución parcial, en que, creábamos de forma simbólica los elementos deseados y los copiábamos a la estructura.

Habiendo comprendido el funcionamiento y las características ofrecidas por la herramienta KLEE, explotamos su funcionalidad para la generación de casos de pruebas utilizando como medio los contratos software, método de especificación software estudiado en la asignatura Métodos Formales Industriales (MFI). Estos contratos nos ofrecen distintos axiomas acerca de nuestro código, describiendo unas precondicones y postcondiciones que se cumplirán durante la ejecución del mismo. Como hemos mencionado durante nuestro trabajo, estos contratos se pueden inferir de forma automática, pero pueden no ser del todo correctos o precisos. En consecuencia, usamos la funcionalidad del método **klee_assume()** para adjudicar ciertas precondicones a la generación de casos de pruebas. Esto nos ofreció la oportunidad de verificar nuestros axiomas candidatos y, debido a ello, automatizar este proceso con el desarrollo de nuestra aplicación.

Nuestro propósito ha sido crear una aplicación capaz de ejecutar KLEE de forma automática adjudicando ciertas precondiciones, previamente establecidas, y generar los casos de pruebas para establecer que postcondiciones nunca se darán. Para ello, seguimos una metodología ágil, vista y ejecutada anteriormente en las asignaturas Proceso del Software (PSW) y Proyecto de Ingeniería del Software (PIN). Tomamos esta decisión, para poder controlar mejor el proceso de desa-

rrollo, sobre todo, si se producían ciertos inconvenientes. Por ejemplo, a la hora de desarrollar la interfaz de linea de comandos (implementación que la realizábamos por primera vez), poder cambiar el diseño de la aplicación sin alterar el proceso de producción.

El uso de una metodología ágil junto a las fases de análisis, diseño, desarrollo y validación, propias de un producto software, nos ha permitido guiarnos de forma correcta y manteniendo la calidad del producto durante la realización de nuestro proyecto. En cada una de estas fases hemos realizado tareas diferentes: En la fase de análisis nos encargamos de identificar los requisitos funcionales y no funcionales, mediante técnicas de especificación, vistas en varias asignaturas de la carrera pero profundizadas en Análisis y Especificación de Requisitos (AER). Además analizamos las diversas propuestas o soluciones para la realización de la herramienta. Una vez decidido que sería una aplicación vía comandos programada en Java, pasamos a diseñar la arquitectura de nuestro sistema siguiendo ciertos patrones de diseño, aprendidos en Diseño de Software (DDS), junto al diseño de clases a desarrollar. Con respecto al desarrollo, realizamos cuatro comandos que abarcaban las seis funcionalidades obtenidas durante el análisis de requisitos, siguiendo el diseño de arquitectura por capas. Finalmente, la validación donde probamos nuestra aplicación con los ejemplos desarrollados durante el trabajo.

Como resultado, se ha logrado con éxito el propósito de desarrollar una herramienta de validación automática de contratos software partiendo de un programa en C junto a su contrato, que asumimos inferido automáticamente, mediante el uso de la herramienta KLEE para la generación de casos de prueba usados posteriormente para el proceso de validación y detección de los componentes decididamente falsos del contrato.

Bibliografía

[1] María Alpuente, Daniel Pardo, y Alicia Villanueva. Symbolic Abstract Contract Synthesis in a Rewriting Framework. In Manuel V. Hermenegildo y Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, págs. 187–202. Springer, 2016.

[2] María Alpuente, Daniel Pardo, y Alicia Villanueva. Abstract Contract Synthesis and Verification in the Symbolic K Framework. *Fundamentos Informáticos 2020*, Próximamente.

[3] Douglas N Arnold. The explosion of the ariane 5. *URL: http://www. ima. umn. edu/arnold/disasters/ariane. html*, 2000.

[4] Cristian Cadar. Can't make_symbolic a struct member. https://www.mail-archive.com/klee-dev@imperial.ac.uk/msg00826.html, 2012. Comunicación privada. Acceso Enero 2020.

[5] Cristian Cadar, Daniel Dunbar, y Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves y Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, págs. 209–224. USENIX Association, 2008.

[6] Patrick Cousot y Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, y Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, págs. 238–252. ACM, 1977.

[7] Leonardo De Moura y Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, págs. 337–340. Springer, 2008.

[8] Eclipse IDE for Java Developers. https://www.eclipse.org/downloads/packages/. Herramienta. Acceso Julio 2020.

[9] Carlos Alberto Fernández y Fernández et al. Métodos formales aplicados a la industria del software. *Repositorio Nacional Conacyt*, 2011.

[10] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns Elements of reusable object-oriented sofware*. Addison Wesley, 2009.

[11] Java: The world's most popular modern development platform. https://www.oracle.com/java/. Herramienta. Acceso Julio 2020.

[12] Overview of the main KLEE intrinsic functions. https://klee.github.io/docs/intrinsics/. Documentación. Acceso Marzo 2020.

[13] Barbara Liskov, John Guttag, et al. *Abstraction and specification in program development*, volume 20. MIT press Cambridge, 1986.

[14] Corina S. Pasareanu y Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In Charles Pecheur, Jamie Andrews, y Elisabetta Di Nitto, editors, *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, págs. 179–180. ACM, 2010.

[15] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, y Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[16] Picocli - a mighty tiny command line interface. https://picocli.info/. Herramienta. Acceso Julio 2020.

[17] Nadia Polikarpova, Ilinca Ciupa, y Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In Gregg Rothermel y Laura K. Dillon, editors, *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, págs. 93–104. ACM, 2009.

[18] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, y Bertrand Meyer. What good are strong specifications? In David Notkin, Betty H. C. Cheng, y Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, págs. 262–271. IEEE Computer Society, 2013.

[19] R Pressman. Ingeniería del Software Un Enfoque Práctico. 7ma ed. University of Connecticut, 2010.

[20] Grigore Rosu. K: A semantic framework for programming languages and formal analysis tools. *Dependable Software Systems Engineering*, 50:186, 2017.

[21] Roland Schinzinger. Ethics on the feedback loop. *Control Engineering Practice*, 6(2):239–245, 1998.

[22] Nikolai Tillmann y Jonathan de Halleux. Pex-White Box Test Generation for .NET. In Bernhard Beckert y Reiner Hähnle, editors, *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, págs. 134–153. Springer, 2008.

[23] Testing a Small Function. https://klee.github.io/tutorials/testing-function/. Documentación. Acceso Marzo 2020.

[24] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8–24, 1990.

APÉNDICE A

Código de insert()

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  struct set {
5    int capacity;
6    int size;
7  };
8
9  struct set* new(int capacity) {
10   struct set *new_set;

11
12   new_set = (struct set*) malloc(sizeof(struct set));
13   if(new_set == NULL) return NULL; /* no memory left */

14
15   new_set->capacity = capacity;
16   new_set->size = 0;
17   *(new_set->elem) = malloc(capacity * sizeof(int));
18   return new_set;
19 }
20
21 int insert(struct set *s, int x) {
22   int found;
23   int i;
24
25   if(x==NULL)
26     return 0;
27   if(s==NULL)
28     return 0; /* NULL set */
29
30   if(s->size >= s->capacity) return 0; /* no space left */
31
32   if(s->elem == NULL) { /* empty set */
33     s->elem[s->size] = x;
34     s->size = 1;
35     return 1;
36   }
37
38   found = 0;
39   for(i = 0; i< s->capacity; i++) {
40     if(s->elem[i] != NULL){
41       if(s->elem[i] == x) {
```

```
42          found = 1;
43        }
44      }
45    }
46
47    if(found) return 0; /* element already in the set */
48    s->elem[s->size] = x;
49    s->size = s->size + 1;
50
51    return 1; /* element added */
52  }
53
54  int isnull(struct set *s) {
55    if(s==NULL)
56      return 1;
57    return 0;
58  }
59
60  int isempty(struct set *s) {
61    if(s==NULL)
62      return 0;
63    if(s->elem==NULL)
64      return 1; /* s is empty */
65    return 0;
66  }
67
68  int isfull(struct set *s) {
69    if(s==NULL)
70      return 0;
71    if(s->size >= s->capacity)
72      return 1; /* s is full */
73    return 0;
74  }
75
76  int contains(struct set *s, int x) {
77      int i;
78
79    if(s==NULL)
80      return 0; /* s is NULL */
81
82    for(i = 0 ; i < 3 ; i++){
83      if(s->elem[i] == x)
84        return 1; /* element found */

85    }
86
87    return 0; /* element NOT found */
88  }
89
90  int length(struct set *s) {
91    if(s==NULL)
92      return 0; /* s is NULL */
93
94    return s->size;
95  }
```

APÉNDICE B
Instalación y uso de la aplicación

En este apéndice se describe cómo se instala la herramienta desarrollada junto con una guía para usar cada uno de los comandos creados para realizar las distintas actividades dentro de nuestro programa.

B.1 Instalación

Para poder instalar correctamente la herramienta que hemos desarrollado, en primer lugar, habrá que tener un sistema operativo Linux en la cual instalarla.

En segundo lugar, tendremos que tener previamente instalada la herramienta de casos de prueba KLEE. Dicha instalación está descrita en la Sección 3.2.2. Una vez instalada, hay que guardar dos variables de entorno en nuestro ordenador para poder usar KLEE con normalidad. En ellas, pondremos la ruta absoluta, en nuestro ordenador, de KLEE y una de sus librerías como podemos ver en B.1.

```
# klee
export PATH=/home/sandra/klee/build/bin:$PATH
# libreria klee
export LD_LIBRARY_PATH=/home/sandra/klee/build/lib/:$LD_LIBRARY_PATH
```

Figura B.1: Ejemplo de las variables de entorno necesarios

Por último, tendremos que descargar dos archivos **jar**, uno con nuestro programa llamado *ContractFalsifier*, y otro con la herramienta utilizada para crear los comandos de nuestro programa, PICOCLI.

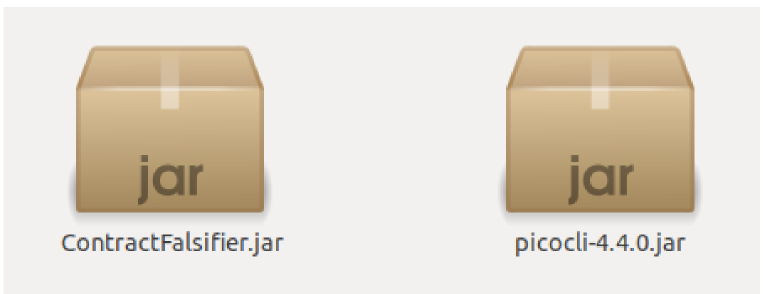


Figura B.2: Archivos necesarios para la instalación de la aplicación

Estos archivos se pueden colocar en cualquier parte de nuestro sistema pero tendremos que recordar su ruta o acceder a la terminal de Linux desde la misma ruta donde se encuentren.

B.2 Uso

Una vez instalado todo lo necesario para ejecutar el programa, solo hace falta abrir la terminal de Linux y usar el comando correspondiente con los archivos correctos.

B.2.1. Comando *Add*

Con el comando *Add* podremos añadir el archivo con nuestro porgrama C y el archivo de texto que contendrá el contrato. En el contrato estará la preparación que hay que realizar para que nuestro programa pueda funcionar con KLEE 3.2.3

Además de esto, creará un nuevo fichero que contendrá la combinación de los archivos que hemos mencionado (programa y contrato) para poder después ejecutar en KLEE directamente este archivo de tal forma que si queremos hacer diversas pruebas solo habrá que añadir un nuevo contrato.

Pulsando el siguiente comando ejecutaremos la funcionalidad adjudicada al comando *Add*.

java -cp "picocli-4.4.0.jar:ContractFalsifier.jar"presentacion.Add -c contrato.txt -a archivo.c

Como podemos ver, el comando *Add* tiene dos opciones -c o - -contrato más el nombre del archivo de texto y -a - -archivo más el nombre del archivo C donde especificaremos los archivos que deseamos que use nuestro programa.

B.2.2. Comando *Run*

Con el comando *Run* realizaremos varias acciones que conectarán con la herramienta KLEE, como la compilación del programa C seleccionado, su generación de casos de prueba y finalmente la lectura de los casos de prueba creados. Se irá transmitiendo por consola cada proceso y su evolución hasta finalizar, si no hay ningún error de compilación o de ejecución. Si los hubiera, se retransmitiría al usuario este hecho con información del problema para que pueda solucionarse de manera eficaz.

Además, con la ejecución de este comando se crearán varios archivos propios de KLEE que se guardarán en el directorio correspondiente de la ejecución, **klee-out-n**. Para más información en el apartado 3.2.5 están descritos estos archivos generados con detalle.

Por otra parte se creará un archivo con la lectura de los casos de prueba, ya de que por si el fichero generado con el test no es legible sin el uso de una he-

rramienta auxiliar de KLEE . También se almacenarán dichos casos de prueba en nuestra base de datos.

Pulsando el siguiente comando ejecutaremos la funcionalidad adjudicada al comando *Run*.

java -cp "picocli-4.4.0.jar:ContractFalsifier.jar"presentacion.Run -a archivo.c

Como en el comando anterior nuestro comando tendrá unas opciones. En este caso solo estará disponible la opción de seleccionar archivo con -a o - - archivo más nombre del archivo.

B.2.3. Comando *Check*

Con el uso del comando *Check* validaremos nuestro contrato software. Para ello llamaremos otra vez a la herramienta KLEE para ejecutar los casos de prueba creados anteriormente y observar los resultados obtenidos. Finalmente, crearemos un archivo para almacenar dichas observaciones y el resultado de la validación.

Pulsando el siguiente comando ejecutaremos la funcionalidad adjudicada al comando *Check*.

java -cp "picocli-4.4.0.jar:ContractFalsifier.jar"presentacion.Check -a archivo.c -c contrato.txt

Se puede optar solo a la opción de añadir un archivo en C, pero dicho archivo tiene que estar preparado para KLEE . Si ese no es el caso, usaremos las dos opciones preestablecidas para poder proseguir con la validación del programa.

B.2.4. Comando *Help*

Por último, el comando más simple de todos, *Help* que servirá como ayuda para saber como funcionan los comandos anteriores si el usuario por alguna razón no supiera manejarlos o tuviera algún problema.

Pulsando el siguiente comando ejecutaremos la funcionalidad adjudicada al comando *Help*.

java -cp "picocli-4.4.0.jar:ContractFalsifier.jar"presentacion.Help