UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Validación Automática de Contratos Software con Z3

Trabajo Fin de Máster

Máster Universitario en
Ingeniería y Tecnología de Sistemas Software

Departamento de Sistemas Informáticos y Computación

Autor/a: Sanz Carreres, Sergi

Tutor/a: Alpuente Frasnedo, María

Villanueva García, Alicia

Curso 2019-2020

Agradecimientos

A lo largo de este año son muchas las personas que me han ayudado, apoyado o aconsejado, tanto para el trabajo que se describe en esta memoria como en otros proyectos. Son tantas las personas a las que debería mencionar qué no se si voy a poder mencionarlas a todas.

En primer lugar, mi agradecimiento va dirigido a mis padres, por ser esos héroes sin capa que siempre te animan y apoyan, aunque muchas veces no comprendan del todo que estas haciendo, creen en ti y sabes que pase lo que pase siempre estarán a tu lado .

A mi hermana, por estar siempre a mi lado y ayudarme a levantarme cuando ni yo mismo lo creía capaz, creyendo en mí y dejándome ver día a día cómo se convierte en alguien admirable.

A los profesores y compañeros del Máster, por todo lo que he aprendido con ellos. En especial a Fernando, por mostrarme su cultura y estar siempre dispuesto a ayudar desinteresadamente. A David, por esos debates sobre *frameworks* y lenguajes de programación, pero sobre todo, porque indistintamente de la hora del día que sea, siempre está dispuesto a ayudar a elegir cuál es la mejor tecnología para un proyecto. A Marc, por enseñarme que a veces aunque vayamos más despacio, no significa que no estemos avanzando. Sin olvidar a Melary, por ser un ejemplo a seguir y demostrarme que con constancia y perseverancia se puede lograr cualquier cosa.

A los profesores y compañeros del grupo ELP, del Departamento de Sistemas Informáticos y Computación, por haberme acogido y ayudado, en especial a Nacho, Lidia y Julia, que desde el primer día han ejercido de hermanos mayores aconsejándome y ayudándome siempre que ha sido necesario.

También, me gustaría agradecer especialmente a mis tutoras María Alpuente y Alicia Villanueva, por compartir conmigo sus conocimientos, sus experiencias y su tiempo en el desarrollo de este proyecto. Además de la enorme paciencia que han tenido para transmitirme todos los conocimientos que han sido necesarios. Pero sobre todo, por confiar en mí y brindarme la oportunidad de trabajar en el grupo ELP y realizar este proyecto. Ya que, sin su ayuda todo lo logrado no hubiera sido posible.

¡Gracias a todos!

Sergi

Resumen

En Ingeniería de Software, el concepto de contrato está relacionado con una especificación del comportamiento de los programas utilizando descripciones que típicamente incluyen precondiciones y postcondiciones.

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 permite refinar dichos contratos, utilizando el sistema resolutor SMT Z3 para identificar y eliminar aquellos componentes que el proceso de validación determina que son demostradamente falsos.

Palabras clave: Generación de casos de prueba, ejecución simbólica, satisfacibilidad de restricciones,*Z3*, *SMT-Solvers*

Resum

En Enginyeria de Software, el concepte de contracte està relacionat amb una especificació del comportament dels programes utilitzant descripcions que típicament inclouen precondicions i postcondicions.

L'estat de l'art actual permet generar automàticament contractes a partir de el 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 permet refinar aquests contractes, utilitzant el sistema resolutori SMT Z3 per identificar i eliminar aquells components que el procés de validació determina que són demostradament falsos.

Paraules clau: Generació de casos de prova, execució simbòlica, satisfacibilitat de restriccions, *Z3*, *SMT-Solvers*

Abstract

In Software Engineering, the concept of contract is related to a specification of the behavior of programs using descriptions that typically include preconditions and postconditions.

The current state of the art allows contracts to be automatically generated from the source code that can be used as input for increasingly powerful analyzers. 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 said contracts, using the SMT Z3 resolving system to identify and eliminate those components that the validation process determines are demonstrably false.

Key words: Test case generation, symbolic execution, constraint satisfiability, Z3, SMT-Solvers

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En este documento se expone el trabajo de fin de máster correspondiente al Máster Universitario en Ingeniería y Tecnología de Sistemas Software de la Universitat Politècnica de Valencia.

En este capítulo se describirán los motivos para el desarrollo de este trabajo, los objetivos que se esperan lograr, el impacto que se espera conseguir con el cumplimiento de estos objetivos, la metodología empleada para el desarrollo del proyecto, la planificación seguida y la estructura utilizada durante el proceso de desarrollo de esta memoria.

1.1 Motivación

En los últimos años, la industria del desarrollo de software ha crecido a un ritmo sin precedentes, desarrollándose herramientas y aplicaciones para prácticamente todos los aspectos de nuestras vidas. Al mismo tiempo, la ingeniería del software ha evolucionado en paralelo con el fin de controlar las necesidades del desarrollo de software, que abarca desde el análisis de los requisitos que un producto software debe satisfacer hasta el diseño de la solución que será clave para una posterior implementación y mantenimiento del software.

Este aumento significativo del volumen de software ha supuesto también un aumento en la complejidad del mismo. Esto supone que la probabilidad de introducir un error en el código es mucho mayor. Los errores en el software pueden causar todo tipo de consecuencias negativas para los clientes y usuarios.

Por esta razón se han desarrollado herramientas que pueden analizar el código implementado y verificar el rendimiento del software, las cuales son de gran importancia no solo porque ayudan a obtener un producto de calidad, sino porque también ayudan a reducir el tiempo de mantenimiento requerido para resolver errores potenciales y/o futuros .

Por lo tanto, es muy importante utilizar métodos que puedan eliminar o evitar errores en el código antes de su implementación. Uno de los enfoques mejor establecidos para lograr esto es utilizar métodos formales.

Los métodos formales son un conjunto de técnicas para describir, analizar y garantizar las propiedades del sistema, haciendo uso de modelos matemáticos rigurosamente especificados con el objetivo de mejorar la calidad del software mediante el uso de técnicas de verificación automática. De esta forma podemos conocer si el software cumple con las especificaciones definidas, lo que supone una reducción significativa de los problemas ocasionados por los errores no detectados.

En ocasiones, el desarrollo matemático necesario para formalizar las especificaciones requeridas por los métodos formales puede resultar laborioso. Para mitigar este problema de especificación de los requisitos del software, en este trabajo se plantea una solución basada en la automatización del proceso.

1.2 Objetivos

En Ingeniería de Software, el concepto de contrato está relacionado con una descripción del comportamiento de los programas utilizando precondiciones, postcondiciones e invariantes.

Este proyecto se centra en el proceso de analizar y verificar contratos software, los cuales son esenciales para automatizar estrategias de prueba y ayudar a filtrar entradas o salidas no válidas de forma automatizada. También es posible generar contratos automatizados utilizando, entre otras, la herramienta de síntesis KindSpec 2.0 [17], basada en técnicas de ejecución simbólica y subsunción abstracta.

Sin embargo, debido al uso de abstracciones para garantizar que el proceso de generación de los contratos es finito, los contratos generados automáticamente pueden no ser completamente precisos o correctos, conteniendo algunos elementos que no están verificados.

El objetivo principal de este proyecto es desarrollar un sistema software que pueda ayudar a validar los contratos de programas que se infieren automáticamente o que sean proporcionados manualmente por el ingeniero de software.

Para lograr esto, se proporcionará una traducción adecuada entre el lenguaje de salida de la herramienta de generación de contratos KindSpec 2.0[1] y el *SMT Solver* Z3 [2], que será la herramienta utilizada para validar los contratos.

1.3 Impacto esperado

Con el desarrollo de este proyecto se desea conseguir un nuevo software que pueda ayudar a garantizar la validez de los contratos software, en particular los que se infieren automáticamente con una aceptable precisión.

[1] KindSpec 2.0: http://safe-tools.dsic.upv.es/kindspec2
[2] z3: https://ericpony.github.io/z3py-tutorial/guide-examples.htm

1.4 Metodología

Una vez identificados y definidos los objetivos del proyecto es necesario elegir una correcta metodología para el desarrollo de éste. En este caso, se ha decidido elegir una metodología basada en un desarrollo iterativo e incremental.

El desarrollo iterativo e incremental se caracteriza por ser un proceso de desarrollo software que agrupa un conjunto de tareas en etapas que se repiten durante el ciclo de vida del proyecto. Se diferencia del ciclo de vida clásico en que este se organiza en fases consecutivas, donde cada una de las fases se centra en una actividad concreta. Siendo muy diferente cada fase del resto.

Cada vez que finalizamos una etapa, obtenemos un producto funcional que se mejora en cada iteración. Por tanto, podemos apreciar que esta metodología es iterativa que consiste en un número distinto de tareas que se repiten (proceso iterativo) de forma que aportan o incrementan la calidad o las funcionalidades del proyecto (incremental) [26].

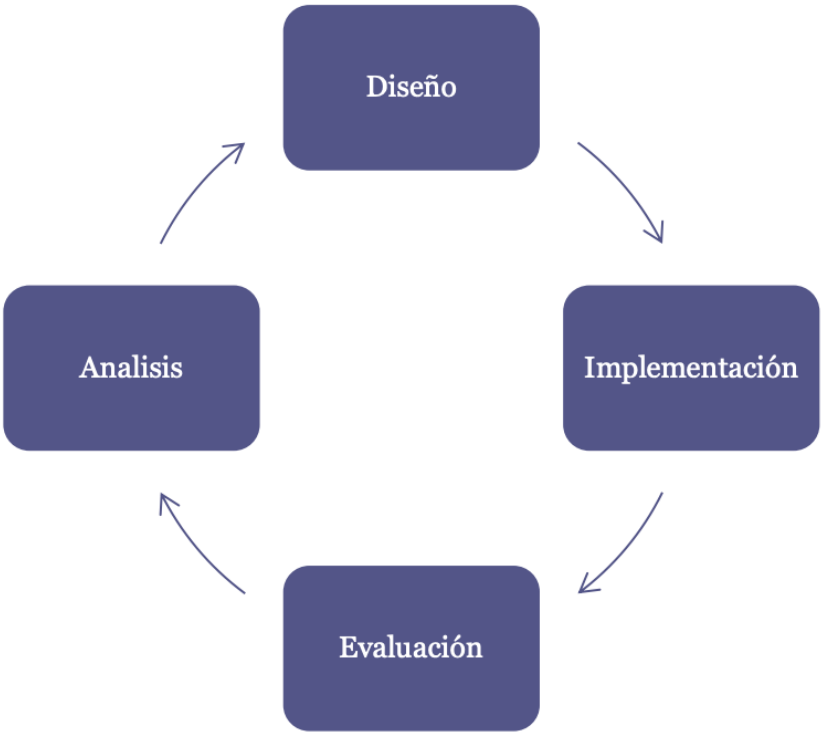

Figura 1.1: Ciclo de vida iterativo e incremental.

Como se ilustra en la Figura 1.1 podemos observar las distintas fases que componen el ciclo de vida iterativo e incremental:

- **Fase de análisis de requisitos:** Se debe analizar las necesidades de los usuarios finales para determinar los objetivos.
- **Fase de diseño:** El objetivo es describir la estructura interna del software y las relaciones entre los distintos componentes que lo formen.

- **Fase de implementación:** Se implementan las distintas soluciones obtenidas en las etapas anteriores, dando como resultado un programa operativo.
- **Fase de evaluación:** Nos encargaremos de comprobar que el programa obtenido en la fase de implementación cumple con los requisitos especificados en la primera fase.

Continuando con el ciclo iterativo e incremental, se vuelve a empezar una nueva iteración donde se evalúa de nuevo el estado del proyecto, identificando nuevos riesgos y requisitos, siendo posible repetirlo tantas veces como sea necesario.

El motivo de la utilización de este ciclo de vida ha sido el desconocimiento a priori de las dificultades y desafíos, dividiendo el proyecto en varias partes, aportando una solución para cada uno de los requisitos identificados.

1.5 Planificación

Uno de los aspectos más importantes en el momento de realizar un proyecto es seguir una planificación adecuada que permita satisfacer de forma correcta todas las partes de éste.

La planificación aplicada a este proyecto es la siguiente:

- **Investigación previa:** Tiempo necesario para comprender de forma correcta todas las tecnologías necesarias para el desarrollo del proyecto.
- **Preparación de la memoria:** A lo largo de todo el proyecto se ha ido desarrollando la memoria al mismo tiempo que se realizaban las distintas fases del trabajo.
- **Caso de estudio:** Planteamiento del caso de estudio que se va a realizar y el escenario a desarrollar para diseñar y evaluar la solución propuesta.
- **Diseño:** En esta fase se ha abordado cómo realizar el diseño de las soluciones propuestas.
- **Implementación:** El objetivo principal de esta fase es el de implantar la solución que se ha diseñado en la fase anterior.
- **Pruebas:** Esta fase consiste en realizar el testing de cada módulo que compone el proyecto con el fin de certificar su correcto funcionamiento.
- **Revisión general:** En esta fase se realiza una revisión general de todo el proyecto.

Para la planificación temporal del proyecto se ha optado por la utilización de un diagrama de Gantt, que es una herramienta gráfica cuyo objetivo es mostrar la dedicación realizada a las distintas partes del proyecto durante el tiempo total determinado del mismo. La Figura 1.2 describe la planificación propuesta.

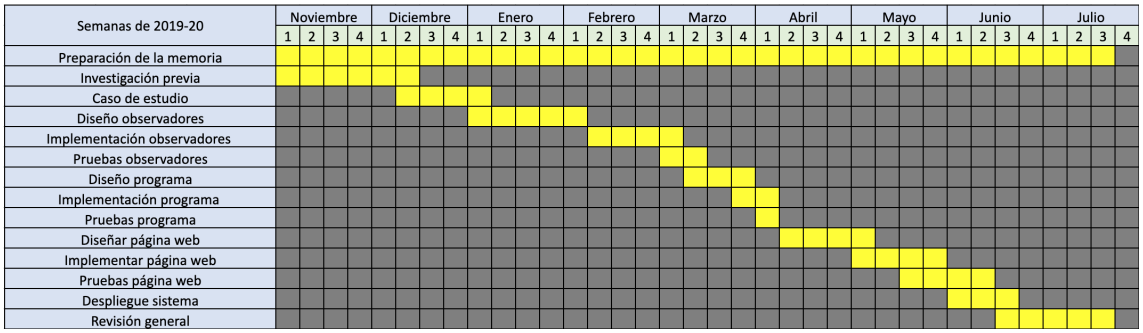

Figura 1.2: Diagrama de Gantt.

1.6 Estructura de la memoria

Este documento se ha dividido en ocho capítulos además de la bibliografía.

El primer capítulo es la introducción donde se describirán los motivos para la realización de este trabajo, los objetivos que se esperan lograr, el impacto que se espera conseguir con el cumplimiento de estos objetivos, la metodología empleada para el desarrollo del proyecto, la planificación seguida y la estructura utilizada durante todo el proceso de desarrollo de esta memoria.

En el segundo capítulo se revisa el estado del arte del campo en el que se enmarca este proyecto.

Partiendo del análisis del estado del arte, en el tercer capítulo se profundizará el contenido sobre los *SMT Solver*.

El cuarto capítulo partirá del conocimiento adquirido en el capítulo anterior, para realizar una comparativa en detalle entre los diversos y más utilizados *SMT Solver* del mercado.

El quinto capítulo detallará en profundidad la distinta sintaxis y lenguaje utilizada en Z3.

En el sexto capítulo se expondrá la implementación de este proyecto, además de algunos aspectos clave como la arquitectura software, las tecnologías y herramientas utilizadas, además del despliegue.

En el séptimo capítulo se realizarán las pruebas pertinentes para la comprobación del correcto funcionamiento de la solución implementada.

Finalmente, en el octavo capítulo se efectuará un resumen del trabajo realizado durante el proyecto.

A continuación se listará la bibliografía utilizada para la realización del proyecto y la memoria.

CAPÍTULO 2
Estado del arte

En este capítulo se describe el contexto del estado del arte en el que se enmarca este proyecto, que no es otro que los *SMT Solvers*, describiendo los problemas que son necesarios resolver y los requisitos necesarios para poder diseñar una solución.

2.1 Orígenes

Una vez finalizada la creación de un programa, aparece la tarea de hacer pruebas de ejecución. Esta tarea implica comprobar si el programa se comporta como debe, lo cual ayuda a aumentar la confiabilidad en el mismo. Para ello se realizan una serie de casos de prueba, para posteriormente realizar la comparación de los resultados obtenidos en dichas pruebas [3].

Normalmente, el proceso de especificar y preparar manualmente estos casos de prueba presenta el inconveniente de que es poco fiable porque no siempre se consideran todos los casos necesarios y puede producirse la omisión de errores que no hemos detectado. Otro aspecto inconveniente es el coste, ya que las pruebas son costosas porque requieren idear casos que se ajusten a nuestro código, realizando numerosas ejecuciones, determinando cuál es la respuesta correcta para cada caso y comprobar los resultados.

Las herramientas automáticas pueden aliviar la mayoría de las actividades tediosas y propensas a errores relacionadas con las pruebas. Una de estas herramientas son las *test-case generation* (TCG) que se presenta en dos variantes, caja negra y caja blanca.

Las pruebas de caja negra consisten en generar casos de prueba basándose exclusivamente en la especificación del programa sin tener ningún conocimiento del funcionamiento interno de la aplicación [27]. Por contra, que las pruebas de caja blanca tiene en cuenta el código del programa y su propósito es generar casos de prueba que recorran los distintos caminos que puede seguir la ejecución del código [13]. Ambos tipos de pruebas se complementan entre si para la realización de pruebas exhaustivas [35].

El problema principal relacionado con la automatización de pruebas es verificar la validez de los resultados devueltos por el programa para cada caso de prueba. Para automatizar este proceso, los programadores deben escribir postcondiciones formales o aserciones formales, y tener versiones ejecutables de ellos, el problema que supone esto es que si estas comprobaciones son realizadas por humanos, entonces aumenta la posibilidad de errores. Para resolver este problema, surgen los *SMT Solver* que son herramientas se encargan de comprobar automáticamente la validez de los casos de prueba [35].

2.2 *SMT Solvers*

La satisfacción de restricciones también conocida como satisfacibilidad proposicional, o SAT, es decidir si una fórmula sobre variables booleanas, formada usando conectivas lógicas, puede ser cierta eligiendo valores verdaderos o falsos para sus variables [22].

La satisfacción módulo teorias (SMT) generaliza la satisfacción booleana (SAT) al agregar razonamiento de igualdad, aritmética, vectores de bits de tamaño fijo, matrices, cuantificadores y otras teorías útiles de primer orden. Ambos, SAT y SMT, se emplean para demostrar la satisfacción de las fórmulas a distintos niveles de satisfacibilidad.

A menudo se da el caso de que no solo existen una sino varias soluciones satisfactorias para una fórmula dada y cada una de ellas tiene un coste que se refiere a algún parámetro relacionado con el problema en sí. Por lo tanto, es posible convertir el problema de satisfacción en uno de optimización al establecer una función objetivo donde no solo se quiere encontrar una solución satisfactoria, sino la óptima con respecto al coste que asignamos a las variables [34].

Un *SMT Solver* es una herramienta para decidir la satisfacción (o validez) de las fórmulas en estas teorías. Los *SMT Solver* son claves para el desarrollo de aplicaciones como la verificación automática, la abstracción de predicados, la generación de casos de prueba y la verificación de modelos acotados sobre dominios infinitos, por mencionar algunos [20].

Los *SMT Solvers* hunden sus raíces en algunas de las áreas más fundamentales de la informática, como por ejemplo en la lógica simbólica, donde se combina el problema de satisfacción booleana con dominios avanzados que pueden involucrar estructuras dinámicas.

En la última década, los *SMT Solvers* han atraído una mayor atención debido a los avances tecnológicos y a sus aplicaciones industriales, de manera que soportan un conjunto más amplio de teorías, y permiten que los programas más complejos puedan ser verificados. Esto ha llevado a un enorme progreso ya que los problemas de satisfacción de restricciones pueden resolverse debido a las innovaciones en los algoritmos clave, los tipos de datos algebraicos, las estructuras de datos, heurísticas y el uso de microprocesadores modernos.

2.2.1. Aplicaciones en el desarrollo de software

Los desarrolladores de software utilizan fórmulas lógicas para describir los estados de los programas y las transformaciones entre estados. Estás fórmulas lógicas se encuentran en el núcleo de la mayoría de herramientas de ingeniería del software que analizan, verifican o testean programas.

Algunas de las aplicaciones del uso de las fórmulas lógicas son [22]:

- ***Dynamic symbolic execution:*** Los *SMT Solver* juegan un rol central en la ejecución simbólica dinámica, ya que están diseñados para recopilar rutas de programas explorados como fórmulas, utilizando *Solvers* para identificar nuevos datos de entrada y casos de prueba que pueden dirigir la ejecución hacia nuevas ramas de la ejecución. Los *SMT Solver* son un excelente soporte para la ejecución simbólica porque la semántica de la mayoría de las declaraciones de datos en los programas se modelan fácilmente utilizando teorías compatibles con estos *Solver*. Un gran número de herramientas utilizadas en la industria están basadas en ejecución simbólica dinámica con *SMT Solving* como son CUTE[1], Klee[2], DART[3], SAGE[4], PEX[5] y Yogi[6].

- ***Program model checking:*** El objetivo de las herramientas de *model checking* es verificar que los programas se encuentren libres de errores. La idea es explorar todas las ejecuciones posibles utilizando una abstracción finita y suficientemente pequeña del espacio de estados del programa. Un gran número de herramientas utilizadas en la industria están basadas en *model checking* como son BLAST[7], SDV[8], y SMV[9] .

- ***Static program analysis:*** Las herramientas de análisis estático de programas no requieren la ejecución de los programas y pueden analizar las librerías software y las utilidades independientemente de como sean usadas. Una de las ventajas de utilizar los *SMT Solver* modernos es que el análisis estático de programas puede capturar con precisión la semántica de la mayoría de las operaciones básicas utilizadas por los lenguajes de programación convencionales.

- ***Program verification:*** Robert Floyd y Anthony Hoare introdujeron (a finales de la década de 1960) la verificación de programas mediante la manipulación de aserciones lógicas. Una de las herramientas mas utilizada para la verificación de programas es Why3. Why3 es una plataforma para la verificación deductiva de programas, que cuenta con un lenguaje rico para la

[1] CUTE:http://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf
[2] Klee:https://klee.github.io/
[3] DART:https://web.eecs.umich.edu/~weimerw/2014-6610/reading/p213-godefroid.pdf
[4] SAGE:https://www.microsoft.com/en-us/research/publication/automated-software-testing-using-program-analysis/
[5] PEX:https://www.microsoft.com/en-us/research/project/pex4fun/
[6] Yogi:https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ieeesw2008.pdf
[7] BLAST:http://cseweb.ucsd.edu/~rjhala/blast.html
[8] SDV:https://www.microsoft.com/en-us/research/project/slam/
[9] SMV:http://nusmv.fbk.eu/

especificación y programación llamado Why3ML que esta basado en demostradores de teoremas externos, tanto automatizados como interactivos [7].

- ***Modeling:*** Los SMT Solvers representan una interesante oportunidad para las herramientas de modelado software de alto nivel. En algunos contextos estas herramientas utilizan dominios matemáticos (como tipos de datos algebraicos, arrays, sets, y maps) y también han sido objeto de extensas investigaciones en el contexto de los SMT Solvers.

2.3 Contratos *software*

Los contratos son anotaciones que documentan y especifican formalmente el comportamiento de un programa para la verificación formal de los mismos [38], proporcionando una base para las técnicas de verificación de programas y siendo esenciales para las estrategias de prueba automatizadas al ayudar a filtrar entradas no válidas actuando como oráculos automatizados. También admiten la depuración al proporcionar información sobre la ubicación de los fallos, lo cual sirve como ayuda de documentación a la vez que mejora el proceso de análisis y diseño [32].

Inferir contratos puede resultar útil para fortalecer las especificaciones escritas por el programador, aunque hay que destacar que la calidad de los contratos inferidos disminuye con el crecimiento del acoplamiento entre clases, de manera que cuantas más consultas de argumento cero tenga una clase, mayor será el porcentaje de cláusulas de aserciones inferidas incorrectamente.

2.4 Z3

El problema de *Satisfiability Modulo Theories* (SMT) es un problema de decisión para fórmulas lógicas con respecto a combinaciones de teorías de base tales como aritmética, vectores de bits, matrices y funciones no interpretadas.

Por tanto, dada una fórmula F y aplicando una serie de restricciones a la fórmula, se deberá determinar si F es satisfacible [13]. Bajo este contexto los *SMT Solvers* proporcionan una relación sinérgica con las herramientas de análisis de software, verificación y ejecución simbólica necesarias para estos objetivos [31].

Para el desarrollo de este proyecto, (como se ha enunciado anteriormente) se hará uso de Z3. Se ha decidido optar por la utilización de Z3 por las ventajas significativas que aporta, que se analizarán en detenimiento en el capítulo 4.

2.4.1. Modo de uso

Una de las formas más convencionales en que el usuario puede utilizar la herramienta Z3, es en dos modos distintos, por un lado podemos determinar la satisfacibilidad de una serie de restricciones y en base a ello obtener un modelo que las verifique, o bien realizar una comprobación de la validez de una fórmula lógica [13].

El primer modo consiste en trabajar con una serie de declaraciones de variables y funciones sobre las cuales se les aplican una serie de restricciones para cumplir ciertos requisitos. Para definir las restricciones se hace uso de asertos, donde se pide a Z3 que compruebe si existe alguna combinación de valores para las variables de manera que puedan ser ciertos estos asertos. En caso de que se cumpla, podemos solicitar que nos devuelva un modelo (que no es más que una asignación de valores a cada variable que hace que todas las restricciones se verifiquen). Una de las limitaciones que se produce al pedir el modelo, es que únicamente se nos devuelve una posible combinación, cuando podríamos estar interesados en obtener más de una [13].

Para el segundo modo, se quiere trabajar en determinar si una cierta fórmula lógica es valida, es decir, si siempre es cierta para cualquier combinación de valores. Para ello se tendrá que negar nuestra fórmula y volver a comprobar la satisfacibilidad, de manera que si Z3 nos devuelve que esta fórmula negada es insatisfacible, entonces podremos asegurar que la original era válida, puesto que se hacía cierta para valores cualquiera [13].

Entre los dos modos de utilización explicados, nosotros utilizaremos el segundo modo. Como ya se ha comentado anteriormente, se busca desarrollar un software con el fin de validar los contratos de programas. Para ello será necesario realizar una serie de declaraciones de variables y funciones sobre las que se aplicará una serie de restricciones con el fin de cumplir ciertos requisitos, y posteriormente se negaran estas formulas y se comprobará la satisfacibilidad para asegurar que la fórmula original se cumple para cualquier asignación de variables [13].

CAPÍTULO 3
Satisfiability Modulo Theories Solver

En este capítulo se describe con mayor profundidad el funcionamiento de los resolutores de restricciones modulo teorías (*Satisfiability Modulo Theories Solvers*) describiendo las tecnologías básicas de este área y el tipo de problemas para los que están diseñados.

3.1 Introducción

Como sugiere el nombre de *Satisfiability Modulo Theories Solver*, los *SMT solvers* tienen una cercana relación a la *Boolean Satisfiability* (SAT). De hecho, la mayoría de *SMT solvers* utilizan un *SAT solver* para evaluar si una instancia de SMT es satisfacible o no.

Debido a este motivo, los recientes avances en el campo de los *SAT solvers* repercutieron directamente en un gran número de avances en los *SMT solvers*. Esto llevó al desarrollo de muchas aplicaciones industriales diferentes en los campos de la verificación de software, pruebas basadas en modelos, verificación de modelos, generación de casos de pruebas y mucho más [39].

3.2 Lógica de primer orden

Los *SMT solvers* determinan si una fórmula, en un lenguaje lógico sin cuantificadores como es la *First-Order Logic* (FOL), es satisfacible o no. Dicha lógica extiende la lógica proposicional para permitir el razonamiento sobre los elementos del dominio .

La lógica de primer orden tiene variables que pueden tomar valor sobre 'individuos', pero no sobre funciones o predicados; tales variables sí se permiten en la lógica de segundo orden o de orden superior. También utiliza los cuantificadores $\forall$ ('para todo') y $\exists$ ('existe') [37].

Conectiva lógica	*Símbolo*	*Ejemplo*	*Significado*
Negación	$\neg$	$\neg q$	No q
Conjunción	$\wedge$	$p \wedge q$	p y q
Disyunción	$\vee$	$p \vee q$	p o q
Condicional	$\rightarrow$	$p \rightarrow q$	p implica q
Bicondicional	$\Leftrightarrow$	$p \Leftrightarrow q$	p sí y solo si q

Tabla 3.1: Conectivas lógicas.

3.2.1. Lógica proposicional

En esta sección se va a enunciar con detalle una de las lógicas en las que se basan los *SMT Solver*, la lógica de orden cero o lógica proposicional.

Una proposición es cualquier enunciado del que se puede afirmar sin ambigüedad (y de forma excluyente) si es verdadero o falso. Las proposiciones más simples son llamadas atómicas y se representan con letras minúsculas como pueden ser p, q o r.

Una proposición molecular es una proposición compuesta mediante el uso de las conectivas lógicas que aparecen en en la Tabla 3.1:

Así pues, una fórmula proposicional compuesta es cualquier expresión formada por símbolos (normalmente letras, que representan otras formas proposicionales), conectivas lógicas y pares de paréntesis. Un ejemplo de fórmula preposicional podría ser:

$$p \rightarrow (q \wedge r)$$

Hay que resaltar el hecho de que una fórmula proposicional se convierte en una proposición cuando sustituimos sus variables por proposiciones concretas. Una proposición, a diferencia de una fórmula proposicional compuesta (ya que en este caso puede ser una cosa o la otra según el valor de verdad de las expresiones que la forman) siempre es verdadera o falsa [23]. Por tanto:

- **Tautología:** Es una fórmula proposicional que siempre es verdadera.
- **Contradicción:** Es una fórmula proposicional que siempre es falsa.
- **Contingencia:** Es una fórmula proposicional que no es ni tautología ni contradicción.

3.2.2. Lógica de predicados:

La lógica de predicados surge para superar algunas limitaciones presentes en la lógica proposicional, como es el caso de que no resulta suficientemente flexible para poderse aplicar tanto a individuos como a colectividades de individuos.

En la lógica de predicados se distingue entre las propiedades (también llamadas predicados) y los objetos a los que dichas propiedades se refieren (también llamados términos). Para referirnos a predicados utilizaremos letras mayúsculas, mientras que las letras minúsculas simbolizarán términos de carácter concreto y las variables simbolizaran términos de carácter arbitrario o genérico.

Llamaremos universo de discurso a la clase de objetos o términos sobre los que estamos afirmando algo. El símbolo $\forall$ denota el cuantificador universal, que significa que todos los elementos del universo se ven afectador por esa condición, mientras que el símbolo $\exists$ denota el cuantificador existencial que significa que en ese universo existe un elemento que se ve afectado por esa condición.

3.3 Problema SMT

Un problema SMT consiste en determinar si una fórmula expresada por un cuantificador FOL (*First-Order Logic*, es una extensión de la lógica proposicional de primer orden) es satisfacible con respecto a una teoría de base. Los ejemplos típicos de estás teorías son: la teoría de las funciones no interpretadas con igualdad, la teoría de la aritmética lineal sobre enteros o reales, y teorías de diferentes estructuras de datos como listas, matrices y vectores de bits.

Los problemas en SMT se nutren de los principales problemas de la lógica simbólica del siglo pasado, a saber, el problema de decisión, la teoría de la complejidad y la completitud e incompletitud de las teorías lógicas. Como ya se mencionó, los *SMT solvers* dependen de los *SAT solvers* y SAT es una clase de problemas NP-completos.

Además, FOL (*First-Order Logic*) es indecidible y como la complejidad computacional de la mayoría de problemas SMT es muy alta, la mayoría de *solvers* se centran en resolver problemas prácticos de forma eficiente, como fórmulas producidas por herramientas de verificación y análisis, ya que la mayoría de estas fórmulas se pueden resolver de manera eficiente [25].

Adicionalmente, SMT también busca encontrar algoritmos que manejen eficientemente diferentes teorías y funcionen bien cuando se combinan entre sí. Sin embargo, incluso con todos estos problemas y limitaciones, ha habido un gran progreso en el campo de los SMT en los últimos años. Muchos problemas pueden ser resueltos, por los *SAT solvers* modernos gracias a la constante mejora de los algoritmos y a sus eficientes implementaciones eficientes [21].

3.4 Teorías

Una parte central de SMT se compone de las teorías, o más específicamente, de los *solvers* de teorías. La mayoría de *SMT solvers* modernos siguen el paradigma Davis-Putnam-Logemann-Loveland *modulo theories* (DPLL(T)), el cual sugiere una implementación separada para cada teoría.

Esto indica que no todos los *SMT solver* implementan las mismas teorías, solo aquellas necesarias para su campo de aplicación. Algunas teorías se hicieron muy populares debido a su amplio rango de aplicaciónes, como la teoría de las funciones no interpretadas con igualdad, aritmética lineal sobre enteros y reales, teorías de matrices o vectores de bits.

Básicamente, una teoría T es un conjunto de sentencias donde nosotros afirmamos que si una fórmula φ es satisfacible entonces existe un modelo M que satisface φ bajo la teoría T, denotado como $M \models_T \varphi$.

Además, si hay algún procedimiento δ que compruebe si una fórmula libre de cuantificador es satisfacible o no, entonces el problema de satisfacibilidad para la teoría T es decidible.

Debido a la creciente popularidad de este campo, nació el sitio web *Satisfiability Modulo Theories Library* (SMT-LIB) que comenzó a proporcionar descripciones rigurosas estándard de las teorías más utilizadas [25].

3.4.1. STM-LIB v2

Todos los *SMT Solvers* hacen uso de un lenguaje estándar conocido como STM-LIB, el cual utiliza una sintaxis definida por medio de reglas de producción de estilo BNF, que así mismo es un metalenguaje usado para expresar gramáticas libres de contexto [5].

El SMT-LIB fue creado por Cesare Tinelli y Silvio Ranise convirtiéndose en una iniciativa internacional destinada a facilitar la investigación y el desarrollo de los *SMT Solvers*. Esta iniciativa tiene como objetivo proporcionar descripciones estándar de las teorías de base utilizadas en los sistemas SMT, y normalizar lenguajes de entrada y salida comunes para los *SMT Solvers*.

EL SMT-LIB versión 2.0 fue especificado por Cesare Tinelli, Clark Barett y Aaron Stump. Esta nueva versión fue desarrollada con el objetivo de crear un lenguaje común que los *SMT Solvers* pudieran expresar [15].

Además, se pone a disposición de la comunidad de investigación una gran biblioteca de *benchmarks* para *SMT Solvers*, recopilando y promoviendo herramientas software útiles para la comunidad SMT [1]. La biblioteca contiene más de 100,000 *benchmarks* y continúa creciendo. Las fórmulas en formato SMT-LIB son aceptadas por la gran mayoría de los *SMT Solver* actuales. Por otro lado, gran parte del trabajo experimental publicado en el area de los SMT se basa significativamente en los *benchmarks* SMT-LIB [5].

Clark Barrett y Aaron Stump junto con Leonardo DeMoura también iniciaron la SMT-COMP (*International Satisfiability Modulo Theories Competition*), una competición donde diferentes *SMT solver* compiten unos con otros en diferentes categorías. Además, al utilizar el lenguaje SMT-LIB como lenguaje de entrada estándar para la competición produjo que muchos desarrolladores adaptaron su implementación de SMT actual al nuevo formato, para poder participar en la SMT-COMP [25].

Las diferentes categorías se correlacionan con una lógica SMT-LIB específica como puede apreciarse en la Figura 3.1.

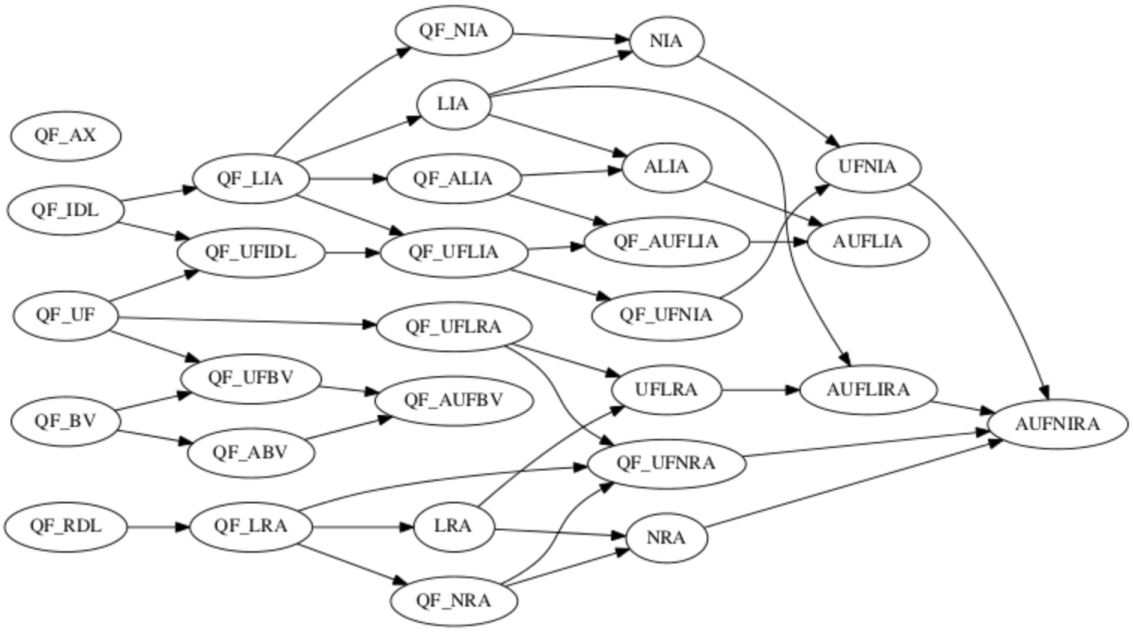

Figura 3.1: Ejemplo lógica SMT-LIB.

Abreviación	Significado
QF	Sin cuantificadores
A o AX	Teoría de arrays
BV	Teoría de bit-vectores de tamaño fijo
IA	Teoría de aritmética de enteros
RA	Teoría de aritmética real
IRA	Teoría de aritmética real de enteros mixtos
IDL	Teoría de lógica de diferencias entera
RDL	Teoría de lógica de diferencias real
L antes de IA, RA, IRA	lineal
N antes de IA, RA, IRA	no lineal
UF	Funciones no interpretadas con igualdad

Tabla 3.2: Explicación lógica SMT-LIB.

En la Tabla 3.2 podemos observar el significado de las abreviaciones utilizados por la lógica de SMT-LIB.

En el siguiente ejemplo (Figura 3.2) se ilustra un ejemplo de la sintaxis utilizada por SMT-LIB, donde se declara la constante s y se hace un *assert* (la función *assert* se encarga de añadir la fórmula al *STM Solver*) solicitando que el valor de la constante sea mayor de 9. Posteriormente se comprueba si el modelo es satisfacible y, en caso de que se cumpla, se muestra el modelo.

```
1  (declare-const s Int)
2  (assert (> s 9))
3  (check-sat)
4  (get-model)
```

Figura 3.2: Ejemplo SMT-LIB.

En este caso, este ejemplo es satisfacible, y el modelo resultante se muestra en la figura 3.3.

```
1  sat
2  (model
3    (define-fun s () Int
4      10)
5  )
```

Figura 3.3: Respuesta ejemplo SMT-LIB.

3.4.2. Funciones no interpretadas con igualdad

Una función no interpretada es una función con un nombre y aridad pero, como su nombre indica, no hay interpretación. Además, la teoría permite conectivas booleanas ($\wedge, \vee$), igualdad (=) y desigualdad ($\neq$).

Los procedimientos de decisión para esta teoría tienen una gran importancia, ya que el problema de decisión para otras teorías puede reducirse a ella. Muchos *solvers* para las teorías de funciones no interpretadas se basan en el método de cierre de congruencia.

Si consideramos una fórmula, consistente en conjunciones de igualdades entre términos que usan funciones libres, se puede aplicar el cierre de congruencia para encontrar una representación del conjunto más pequeño de igualdades implícitas [21].

Finalmente, una vez aplicado el cierre de congruencia se debe realizar una última verificación que compruebe si los términos en ambos lados de cada desigualdad están en diferentes clases de equivalencia.

La Figura 3.4 ilustra un ejemplo de cómo funciona el cierre de congruencia, aplicado para $a = b \wedge b = c \wedge f(a, g(a)) = f(b, g(c)$.

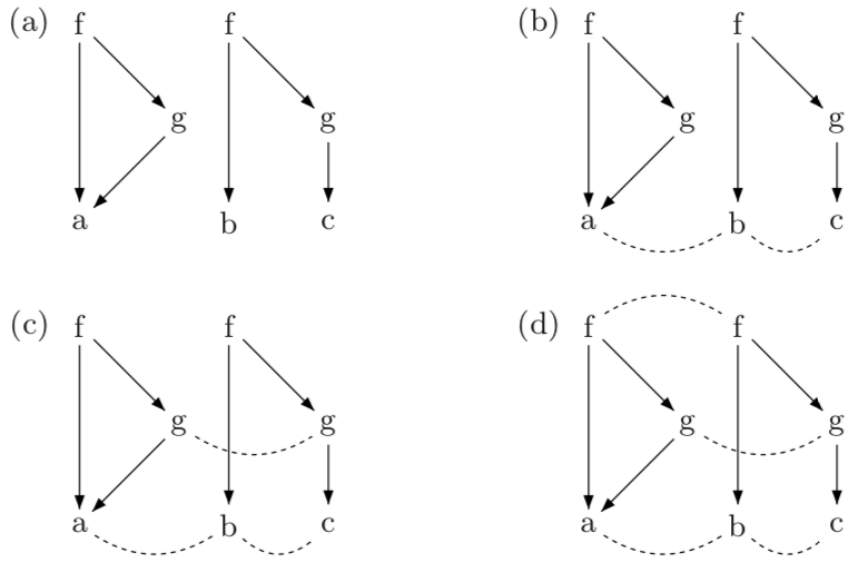

Figura 3.4: Ejemplo cierre de congruencia [21].

3.4.3. Aritmética lineal

La aritmética lineal es una teoría matemática que soporta entre sus funciones las operaciones +,- y * . Sin embargo, el operador * esta restringido a ser utilizado de forma $c * x$ siendo c una constante y x una variable, mientras que los símbolos de igualdad y desigualdad ($=, \leq, <$) son usados en predicados atómicos . El proceso de decisión de restricciones de aritmética lineal utilizado por muchos *SMT solvers* es el *simplex algorithm* que se explicará con más detalle en la sección 3.6 [21].

3.4.4. Diferencia aritmética

Se trata de un fragmento de aritmética lineal donde los predicados están restringidos a la forma $x - y \leq c$, para x, y variables y c una constante numérica. Las conjunciones de desigualdades aritméticas diferenciales pueden verificarse de manera muy eficiente para la satisfacción buscando ciclos negativos en grafos dirigidos ponderados.

En la representación gráfica, cada variable corresponde a un nodo, y una desigualdad de la forma $x - y \leq c$ corresponde a una arista de y a x con peso c. La Figura 3.6 muestra una conjunción de desigualdades de diferencia y el grafo correspondiente [25].

3.4.5. Aritmética no lineal

Los procedimientos de decisión para la aritmética no lineal sobre los reales usan algoritmos de álgebra computacional, como calcular una base de Gröbner (un tipo particular de conjunto generador de un ideal en un anillo polinomial K [x1, .., xn] sobre un cuerpo K) a partir de igualdades.

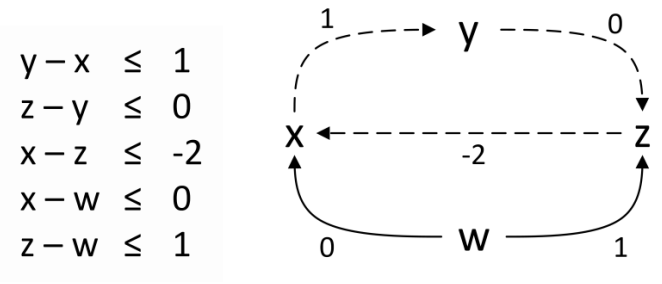

Figura 3.5: Ejemplo de desigualdades de diferencia [21].

Sin embargo, el problema de decidir la satisfacción de la aritmética de enteros no lineales es indecidible. Es decir, no existe un algoritmo que pueda resolver cada instancia de este problema.

Agregar cuantificadores a la teoría lo hace aún peor, ya que ni siquiera hay un conjunto computable de axiomas para caracterizar la aritmética de enteros no lineales con cuantificados.

No hay muchos *SMT Solvers* que admitan aritmética no lineal, lo cual representa un problema ya que en el área de la verificación se precisan aritmética real no lineal [21].

3.4.6. Bit-vectors

Los vectores de bits representan cada número como una secuencia de bits de tamaño fijo. Además de las operaciones aritméticas estándar, la teoría de los vectores de bits también permite mezclar operaciones de bits conocidos, como *and, xor, or y not*. Los procedimientos de decisión eficientes para los vectores de bits utilizan métodos como *lazy bit-blasting* y *approximating long bit-vectors by short bit-vectors* [21].

3.4.7. Arrays

Los arrays tienen dos posibles funciones, escritura o lectura. La definición de la teoría de arrays es ambigua respecto a permitir diferentes extensiones o restricciones. Por ejemplo, algunas teorías restringen qué tipo de matriz están permitidas, al restringir su dimensión máxima.

El enfoque más común para lidiar con la teoría de matrices es usar una reducción a la teoría de funciones no interpretadas con igualdad a través de la instanciación de axiomas de matriz perezosa [25].

3.4.8. Teorías cuantificadas

Si consideramos que los cuantificadores son parte del lenguaje de FOL (*First-Order Logic*), el problema de decidir la satisfacción se vuelve significativamente más difícil. De hecho, no muchos *SMT solvers* admiten teorías que permitan cuantificadores.

Uninterpreted Functions	*Linear Arithmetic*
$f(e_1) = a$	$e_2 - e_3 = e1$
$f(x) = e_2$	$e_4 = 0$
$f(y) = e_3$	$e_5 = a + 2$
$f(e_4) = e_5$	
$x = y$	
variables compartidas: $e_1, e_2, e_3, e_4, e_5, a$	

Tabla 3.3: Ejemplo del método de combinación Nelson-Oppen [25].

3.4.9. Teoría combinada

Como ya se mencionó anteriormente, una dificultad importante en el desarrollo de los *SMT solvers* radica en encontrar algoritmos que no solo sean capaces de manejar de manera eficiente teorías especiales, sino que también puedan combinarse modularmente entre sí.

Existen varios métodos diferentes para combinar teorías para los *SMT Solver*. En la práctica, el método de *Nelson-Oppen combination*, el método de *Delayed Theory Combination* y el método de *Model-based Theory Combination* son los más populares [10].

- ***Nelson-Oppen Combination*:** Asumamos que tenemos una formula de entrada SMT φ, de la forma:

  $$f(f(x) - f(y)) = a \wedge f(0) = a + 2 \wedge x = y.$$

  Para verificar la satisfacción de esta fórmula, tenemos que combinar las teorías de las funciones no interpretadas y la aritmética lineal. La *Nelson-Oppen Combination* divide la fórmula φ en $\varphi_1 \wedge ... \wedge \varphi_n$, tal que, $\varphi_i \in \sum_i$ no tienen ninguna función o símbolos de predicados en común donde los distintos $\varphi_i \in \sum_i 's$, sin embargo, pueden tener variables compartidas.

  La división se realiza de acuerdo con la siguiente regla de transformación de preservación de la satisfacción:

  $$f(x) \rightarrow f(e) \wedge e = x,$$

  donde e es una variable fresca. Para nuestro ejemplo anterior, esto dividiría nuestra fórmula en una parte solucionable por la teoría del *solver* para funciones no interpretadas con igualdad y una parte solucionable por el *solver* de teoría aritmética lineal, como se puede ver en la Tabla 3.3.

  Con eso, los dos *solvers* de teorías pueden verificar la satisfacción de su parte de la fórmula, mientras propagan la igualdad de sus variables compartidas entre ellos. Esto se hace hasta que se alcanza una convergencia, lo que significa que la fórmula es satisfactoria o, en caso contrario, hasta que el *solver* devuelva que el resultado es insatisfactorio [25].

- **Delayed Theory Combination:** El método *delayed theory combination* es un refinamiento para el método de Nelson-Oppen. En lugar de intercambiar directamente las igualdades entre los dos *solvers* de teorías, el método de *delayed theory combination* adopta un enfoque diferente.

  Los *solvers* de teorías trabajan aisladamente unos de otros. Todas las igualdades implicadas entre las variables compartidas se agregan a ambas partes de la fórmula antes de entregarlas al *SAT solver* para encontrar una asignación satisfactoria.

  Esta asignación se divide en diferentes subasignaciones: una asignación para cada teoría, que contiene literales puramente de dicha teoría, y una asignación para las igualdades compartidas. Si ambos *solvers* de teorías devuelven un resultado satisfactorio, la fórmula es satisfactoria.

  De lo contrario, el conjunto de conflictos se agrega a la fórmula, para evitar que ocurran las mismas asignaciones de verdad. Si no es posible encontrar un modelo T-consistente, la fórmula no es satisfactoria [10].

- **Model-based Theory Combination:** El método de combinación de teoría basado en modelos también se basa en la combinación Nelson-Oppen. Para este enfoque cada teoría T_i necesita mantener su propio modelo M_i. Cuando se encuentra una igualdad, la teoría crea un nuevo literal de decisión de igualdad $(u \simeq v)^i$ y la propaga a todas las teorías que comparten u y v. Cada uno de estos modelos M_i necesita ser adaptado para satisfacer el nuevo literal. En caso de que la igualdad no se mantenga dentro de uno de los modelos, se debe verificar la satisfacción para el literal negado. Si esto nuevamente conduce a una inconsistencia, la fórmula es insatisfactoria. De lo contrario, el proceso continúa hasta que se encuentren modelos que satisfagan toda la fórmula [10].

3.5 Resolutor SMT

Como ya se mencionó anteriormente, los *SMT solvers* de última generación utilizan *SAT solvers* eficientes para decidir la satisfacción de una fórmula. Sin embargo, los *SAT solver* trabajan en *propositional logic* (PL) y por lo tanto es necesario realizar, una conversión de FOL en PL es necesario. Además, PL (*propositional logic*)tiene una expresividad más baja que FOL (*first-order logic*) y es por eso que se necesitan varios pasos para una traducción exitosa.

Una vez que la fórmula se tradujo con éxito, se puede pasar al *SAT solver* para decidir si es satisfactoria. En los *SMT solvers* modernos, hay dos enfoques comunes sobre cómo interactúan los *SMT solvers* con el *SAT solver* [25].

3.5.1. *Eager approach*

Los *SMT solver* que implementan la *eager approach* traducen la fórmula FOL en una fórmula PL *Conjunctive Normal Form* (CNF) usando un algoritmo, que preserva la satisfacibilidad. Esto se hace considerando cada átomo como una variable booleana y agregando inconsistencias a la fórmula. En la *eager approach* se derivan todas las inconsistencias antes de llamar al *SAT solver*.

Esto permite una configuración fácil ya que el *SAT solver* funciona como una especie de caja negra. Sin embargo, puede surgir el problema de que se produzcan demasiadas inconsistencias, lo que podría convertir un problema fácil en uno intratable. La mayoría de los *SMT solver* para vectores de bits se basan en la *eager approach*, ya que existe un *eager encoding*, que impide la generación de demasiadas inconsistencias [9].

Además, para una traducción correcta de las fórmulas FOL en fórmulas PL se necesitan procedimientos eficientes para cada teoría. Aunque se hizo un gran esfuerzo para crear algoritmos de este tipo, la *lazy approach* es en muchos casos tremendamente más rápido [22].

Seguir la metodología de resolución da una idea general de cómo un *SMT solver* que interactúa con su *SAT solver* de acuerdo con el enfoque impaciente, decide la satisfacción de una fórmula FOL, donde:

- Asumimos que cada átomo es una variable booleana.
- Se buscan todas las inconsistencias entre los átomos.
- Se traducen la fórmula a una fórmula booleana.
- Se pasa la fórmula SAT resultante a un *SAT solver* y se devuelve el mismo resultado.

En la Figura 3.6 se aprecia como el codificador agrega inconsistencias a la fórmula SMT y la traduce en una fórmula proposicional. La fórmula traducida posteriormente se pasa al *SAT solver* para decidir su satisfacción.

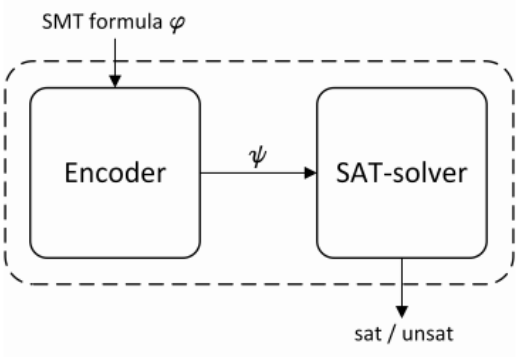

Figura 3.6: Ejemplo eager approach [9].

3.5.2. *Lazy approach*

El *lazy approach* deriva inconsistencias durante el *SAT solving*. Es decir, agrega inconsistencias bajo demanda y, por lo tanto, generalmente requiere menos inconsistencias para encontrar una solución. Sin embargo, para que el *lazy approach* funcione correctamente, necesita interactuar con el *SAT solver* para decidir la T-consistencia de los modelos encontrados.

Esto lleva a una configuración más difícil que con el *eager approach*. No obstante, el *lazy approach* es, debido a su flexibilidad, el procedimiento más utilizado en los *SMT solver* existentes.

Seguir la metodología de resolución da una idea general de cómo un *SMT solver* interactúa con su *SAT solver* de acuerdo con la aproximación precisa, decide la capacidad de una fórmula FOL (*first-order logic*) donde: [22]

- Asumamos que cada átomo es una variable booleana.
- Se pasa la fórmula SAT resultante a un *SAT solver*.
- Si el *SAT solver* se devuelve resultados insatisfactorios, devuelve el mismo resultado.
- Si el *SAT solver* encuentra un modelo, se verifica la T-consistencia del modelo.
- Si el modelo es T-consistente se devuelve que el modelo es satisfactorio.
- Si el modelo es inconsistente en T, se agregan lemas a la teoría, introduciéndolos como input al *SAT solver* y se comienza de nuevo con la decisión de la satisfacción de la fórmula.

En la Figura 3.7 se observa cómo la fórmula se traduce en una fórmula proposicional y se pasa al *SAT solver* para decidir su satisfacción. El *SAT solver* y el solver de teorías interactúan entre sí para decidir la T-consistencia de los modelos candidatos.

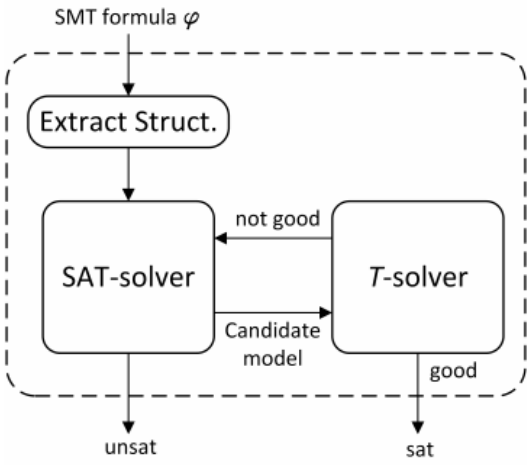

Figura 3.7: Ejemplo *lazy approach* [9].

3.6 DPLL y DPLL(T)

La mayoría de los *SAT solvers* modernos se basan en el paradigma Davis-Putnam-Logemann-Loveland (DPLL) que describe diferentes procedimientos para resolver eficientemente los problemas del SAT. Los *SMT solver* también hacen uso de este paradigma, ya que dependen de un *SAT solver* para decidir la satisfacción.

Sin embargo, como ya se mencionó, la mayoría de los *SMT solver* interactúan con el *SAT solver* de acuerdo con la *lazy approach*. Por lo tanto, es necesario adaptar ligeramente el paradigma DPLL para que pueda interactuar con el *solver* de teorías y trabajar módulo a una combinación de teorías. En las siguientes secciones se presenta un resumen del funcionamiento tanto del paradigma DPLL como un paradigma abstracto DPLL (T) [18].

3.7 Paradigma DPLL

El procedimiento DPLL fue introducido en 1962 por Martin Davis, Hilary Putnam, George Logemann y Donald Loveland para decidir la satisfacibilidad de las fórmulas proposicionales en CNF, conocidas como SAT. Hoy en día, 50 años después, las diferentes variaciones del procedimiento DPLL constituyen la base de la mayoría de los *SAT solver* de última generación [19].

El método consiste en las siguientes siete reglas de transición, que describen de manera general cómo funcionan los *SAT solver* modernos basados en DPLL :

UnitPropagate

Para que una fórmula CNF sea satisfactoria, todas sus cláusulas deben ser verdaderas. Por lo tanto, la unidad de propagación busca cláusulas a cuyos literales se les haya asignado el valor falso, a excepción de un literal, cuyo valor aún no estaba definido por M. La única forma de que la cláusula sea verdadera en M es extender M con el literal restante como verdadero [25].

Decide

Decide realiza una división de casos. Elige un literal indefinido l, le asigna un valor de verdad y lo agrega a M. Adicionalmente, l es denotado como un *decision literal* l^d. En caso de que $l \in M$ no se pueda extender a un modelo de F, pese a considerarse que $\neg l \in M$ [30].

Fail

Si se detecta un *conflicting clause* y M no contiene literales de decisión, es decir, todos los literales en M tienen un valor fijo, DPLL produce un estado fallido devolviendo que F es insatisfacible.

Backjump

Backtracking vuelve cronológicamente al último literal de decisión l^d y lo cambia en $\neg l$. El *Backjumping* se hace impulsado por conflictos, evaluando por qué se produjo tal conflicto y luego, si es necesario, retrocede varios niveles de decisión a la vez, donde agrega algunos literales nuevos a ese nivel inferior [30].

Learn

Learn utiliza las clausulas *backjump* $(C' \vee l')$ agregándolas a la cláusula establecida como cláusulas aprendidas. Teóricamente, permite añadir cualquier clausula C a F siempre y cuando todos los átomos de C están incluidos en F o M. Es decir, no solo se pueden agregar lemas, sino cualquier consecuencia producida por F [25].

Forget

Al contrario que la regla de aprendizaje *Learn*, el *Forget* elimina lemas con niveles de relevancia o actividad por debajo de un cierto umbral. Teóricamente puede eliminar no solo aquellas cláusulas agregadas por 'learn', sino cualquier cláusula, si está implicada por el resto de F. Dado que producir consecuencias y determinar implicaciones es muy costoso, su uso es muy limitado en la práctica [25].

Restart

La idea detrás de *restart* es que el conocimiento adicional de los lemas aprendidos conducirá a que la regla 'decide' se comporte de manera diferente y encuentre una solución más rápida que mediante *backtracking* [30].

3.8 Paradigma DPLL(T)

El paradigma DPLL (T) se basa en DPLL. Sin embargo, antes de que podamos adaptar nuestro modelo abstracto DPLL para trabajar módulos teorías, debemos considerar que en lugar de tratar con literales proposicionales, DPLL (T) trata con literales de primer orden sin cuantificadores [25].

3.8.1. *T - Learn*

La vinculación entre fórmulas se convierte en implicación en T. También las cláusulas T aprendidas pueden pertenecer a M y F, en lugar de solo a F. De lo contrario, la regla se comporta igual que 'learn' [25].

3.8.2. *T - Forget*

El único cambio en la regla T-Forget es que la vinculación entre fórmulas se convierte en vinculación en T [30].

3.8.3. *T - Backjump*

T-Backjump hace uso tanto de la noción proposicional de vinculación ($\models$), como de la noción de primer orden de implicación módulo teorías $(\models)_T$. Las teorías del *solver* esperan que el *SAT solver* encuentre un modelo M para la fórmula. Si se encuentra dicho modelo y no se puede aplicar ninguna de las reglas "Decide", 'Fail', 'UnitPropagate' y "T-backjump', el T-solver verifica la consistencia de los modelos.

Si es T-consistente, la fórmula es satisfactoria con respecto a la teoría. De lo contrario, si M es inconsistente en T, entonces existe un conjunto de literales $\{l_1, ..., l_n\}$ en M, que son inconsistente con la teoría. T-Learn aprenderá el lema de la teoría $\neg l_1 \vee ... \vee \neg l_n$ y 'Restart' se aplicará.

Este proceso es repetido hasta que se encuentre un modelo T-consistent o se alcanza el estado fallido. Teóricamente, así es como funcionan los *SMT solver* para resolver problemas SMT. Sin embargo, la mayoría de los *SMT solver* modernos implementan diferentes métodos para mejorar el rendimiento. Los métodos más utilizados se describen a continuación [25].

3.8.4. *Incremental T-solver*

La mayoría de los *SMT solvers* de última generación implementan el concepto de T-solving incremental. Esto significa que, en lugar de esperar a que el *SAT solver* encuentre un modelo, la T-consistencia de la asignación se verifica de forma incremental mientras se construye usando el procedimiento DPLL, detectando T-inconsistencias tan pronto como se producen, o en ciertos intervalos.

Para que esto funcione de manera eficiente, el *solver* de teorías tiene que ser más rápido en el procesamiento de un literal adicional que en el reprocesamiento de todo el conjunto de literales desde el principio. Esto es, de hecho, factible para muchas teorías pero no todas [30].

3.8.5. *Online SAT solvers*

Después de que se detecta una T-inconsistencia y se aprende como un lema teórico, en lugar de comenzar de nuevo la búsqueda, se aplicará el procedimiento T-Backjump, para volver a un punto donde la asignación aún era T-consistente, o producir un estado fallido a través de la regla 'Fail' [25].

3.8.6. *Theory propagation*

Las técnicas descritas hasta ahora permiten que el *solver* de teorías proporcione información solo después de alcanzar un estado T-inconsistente. La regla de teorías permite detectar literales l verdaderos en la asignación parcial M (denotados como $M \models_T l$) y añade estos literales a M. La propagación de teorías es un tipo de verificación hacia adelante y juega un papel importante en DPLL (T) [30].

3.8.7. *Exhaustive theory propagation*

Es conveniente aplicar la propagación de la teoría con una prioridad más alta que la regla 'Decide'. Las técnicas que no utilizan la propagación de la teoría, sino que aprenden los lemas de la teoría, tienen que agregar muchas consecuencias de la teoría al conjunto de cláusulas y, por lo tanto, duplicar la información de la teoría. Con la propagación exhaustiva de la teoría , el proceso de duplicar la información de la teoría se vuelve innecesario, y su propagación no exhaustiva lo reduce en gran medida [30].

CAPÍTULO 4
Comparativa *SMT Solvers*

En este capítulo se analizan las características que ofrecen los resolutores de restricciones modulo teorías (*Satisfiability Modulo Theories Solvers*) más utilizados y se realiza una comparación entre estos.

4.1 Z3

Z3 es un *SMT Solver* propiedad de Microsoft Research y desarrollado en el lenguaje de programación C++. Destaca por ser muy eficiente y contar con algoritmos especializados para resolver teorías de base.

Z3 utiliza algoritmos novedosos para la instanciación de cuantificadores y la combinación de teorías, siendo uno de los más completos ya que es capaz de soportar la teoría de funciones no interpretadas, aritmética lineal, aritmética no lineal, bit-vectors, arrays, y tipos de datos algebraicos, entre otros, convirtiéndose en el *SMT solver* que mas se ajusta a nuestras necesidades [20] .

Una de las principales ventajas que ofrece Z3 es que es uno de los pocos resolutores capaces de manejar todas las lógicas SMT-LIB (que se han expuesto en la sección 3.4.1). Ha participado en un gran número de ediciones de la competición SMT-COMP, dónde en 2011 ganó las categorías de QF_BV, QF_UF, QF_LIA, QF_LRA, QF_UFLIA, QF_UFLRA, QF_AUFLIA, QF_IDL, AUFLIA y AUFNIRA, entre otras. Además, 15 pruebas fueron resueltas únicamente por Z3.

El año siguiente Z3 no participó en la SMT-COMP, y sin embargo, ningún otro *solver* pudo mejorar las puntuaciones obtenidas por Z3 en 2011 en ninguna categoría, excepto QF_BV. En 2013 se volvió a presentar y, el número de *benchmarks* que sólo pudieron ser resueltos por Z3 se incrementó a 21, mientras que en la SMT-COMP de 2014 el Z3 participó como no competitivo, y únicamente como referencia para los otros competidores. Sin embargo, todavía ganó en 15 categorías de 32, lo que demuestra la enorme capacidad de este solver [25].

4.1.1. Características técnicas

Entre las características técnicas que ofrece Z3, existe una gran cantidad de características diferentes, entre las que destacan las siguientes[25]:

- ***User Interaction:*** Es posible interactuar con Z3 a través de *scripts* de SMT-LIB v2.0 suministrados como un archivo de texto o a través de Z3, o usando llamadas a la API desde un lenguaje de programación de alto nivel como C,C++,.NET, Python o Java, entre otros. También soporta el formato de entrada *DIMACS* y *simplify*.
- ***Core technology:*** Z3 implementa un núcleo llamado *simplify* que se encarga de introducir fórmulas aplicando reglas de reducción algebraica estándar y simplificaciones especificas de la herramienta.
- ***Theory Solver:*** Z3 integra un solucionador de teorías básicas que maneja igualdades y funciones no interpretadas junto con diferentes solucionadores satélites para aritmética lineal, vectores de bits, matrices y otros. Además, Z3 hace uso de un manejador de *matching* ecuacional. El solucionador de teorías para funciones no interpretadas con igualdades se basa en el algoritmo de cierre de congruencia. Como base para el solver de aritmética lineal se utiliza el algoritmo simplex. Para combinar teorías, Z3 utiliza el método de combinación de teorías basado en modelos [6].
- ***Models and Proofs:*** Z3 es capaz de devolver modelos para satisfacibles fórmulas. Además, puede generar pruebas de de insatisfacibilidad de fórmulas y extraer núcleos insatisfacibles.

4.1.2. Arquitectura

Z3 integra un moderno *SAT Solver* basado en DPLL (algoritmo completo de backtracking para decidir la satisfacibilidad de fórmulas de la lógica proposicional), un *Solver* de teorías básicas que maneja igualdades y funciones no interpretadas, un *Solver* satélite (para aritmética, matrices, etc) y una máquina abstracta de *E-matching* (para cuantificadores) [20].

En la Figura 4.1 se muestra un esquema de la arquitectura interna de Z3:

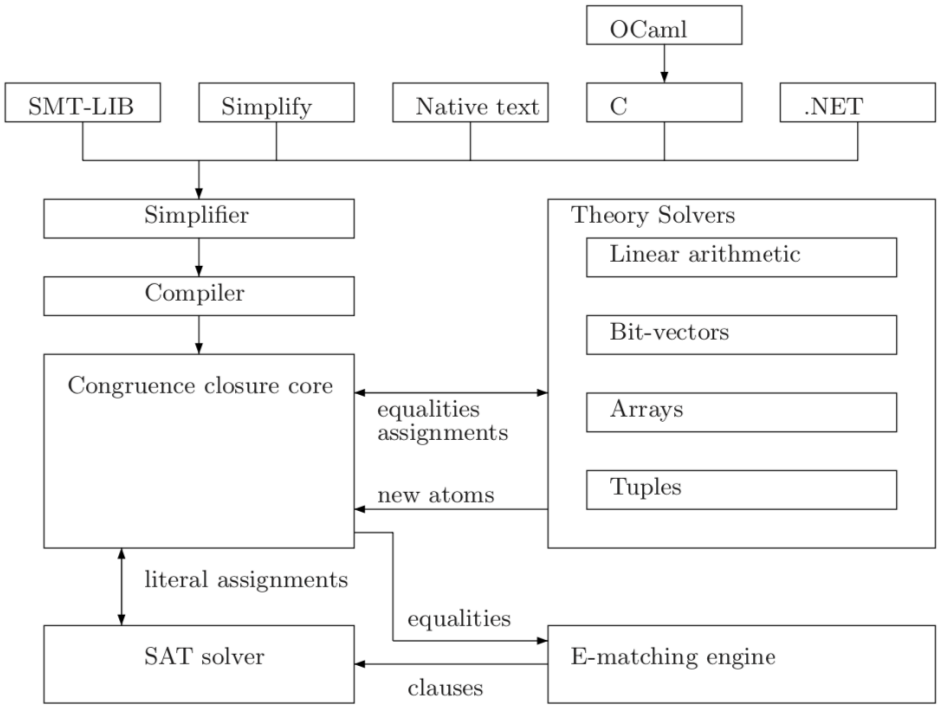

Figura 4.1: Arquitectura interna de Z3.

Las partes que conforman dicha arquitectura son las siguientes:

- **Simplifier:** Las fórmulas de entrada se procesan primero utilizando una simplificación incompleta pero eficiente. El simplificador aplica reglas estándar de reducción algebraica, como $p \wedge true \rightarrow p$, pero también realiza una simplificación contextual limitada ya que identifica definiciones ecuacionales dentro de un contexto y reduce la fórmula considerada utilizando la correspondiente definición.
- **Compiler:** La representación de árbol de sintaxis abstracta simplificada de la fórmula se convierte en una estructura de datos diferente que comprende un conjunto de cláusulas y nodos de cierre de congruencia.
- **Congruence Closure Core:** Este núcleo recibe asignaciones del SAT Solver en forma de átomo. Este átomo se propaga a través del núcleo utilizando estructuras de datos de tipo E-Graph.
- **Theory Combination:** Z3 utiliza un nuevo método de combinación de teorías que concilia gradualmente los modelos mantenidos por cada teoría.
- **SAT Solver:** El *SAT Solver* integra métodos estándar de poda en la búsqueda (como pueden ser el usar dos literales de observación), para una propagación de restricciones booleanas eficiente, aprendizaje de lemas mediante cláusulas de conflicto y el almacenamiento de fases en caché para guiar divisiones de casos y realizar un seguimiento no cronológico.

- **Deleting clauses:** El efecto secundario que tiene la instanciación de cuantificadores es la creación de nuevas cláusulas que contienen nuevos átomos en el espacio de búsqueda. El recolector de basura de Z3 recoge las cláusulas que fueron inútiles, junto con sus átomos y términos y los reutiliza.
- **Relevancy propagation:** Los *solvers* basados en DLPP asignan un valor booleano a todos los átomos potenciales que aparecen en un objetivo. En la práctica, Z3 ignora estos átomos para teorías costosas.
- **Quantifier instantiation using E-matching:** Z3 utiliza un enfoque bien conocido para el razonamiento con cuantificadores que funciona sobre un *E-graph* para instanciar variables cuantificadas. Z3 utiliza nuevos algoritmos que identifican coincidencias en *E-graphs* de manera incremental y eficiente.
- **Theory Solvers:** Z3 utiliza un *solver* aritmético lineal basado en el algoritmo utilizado en Yices. La teoría del array utiliza instanciación perezosa de los axiomas de la matriz, mientras que la teoría de los vectores de bits de tamaño fijo aplica el *bit-blasting* (donde la fórmula de entrada se traduce a la lógica proposicional y se entrega a un *SAT Solver*) a todas las operaciones de vectores de bits.
- **Model generation:** Z3 tiene la capacidad de producir modelos como parte de la salida. Los modelos asignan valores de entrada a las constantes y generan funciones parciales para predicados y símbolos de función.

4.2 CVC4

Las siglas CVC4 significan *Cooperating Validity Checker*, es un proyecto conjunto de la universidad de Nueva York y la universidad de Iowa. Es un software open-source escrito en C++ y publicado bajo los términos de la licencia de *Berkeley Software Distribution (BSD)*.

No obstante, algunas compilaciones enlazan con bibliotecas publicadas bajo el *GNU General Public License (GPL)* y, por lo tanto, el uso de estas compilaciones está permitido solo para proyectos de código abierto. CVC4 acepta tres lenguajes de entrada distintos, llamados SMT-LIB v1.0, SMT-LIB v2.0 y el propio lenguaje nativo CVC4 [4].

Además, CVC4 al igual que Z3 es capaz de manejar cada lógica SMT-LIB, siendo uno de los pocos que soporta cuantificadores. En 2012, en la SMT-COMP tan solo CVC4 y su predecesor CVC3 compitieron en la categoría de teorías con cuantificadores [16]. Es por eso que sólo se ejecutaron como una demostración. Sin embargo, el resultado mostró que ninguno de los dos mejoró con respecto a Z3, el ganador del año anterior.

4.2.1. Características técnicas

CVC4 ofrece una gran cantidad de características diferentes, como son [25]:

- ***User Interaction:*** Como se ha mencionado, CVC4 soporta como lenguaje de entrada SMT-LIB v1, SMT-LIB v2 y su propio lenguaje nativo. Lee la entrada de un archivo externo y reconoce el lenguaje de entrada por la extensión del archivo (.cvc para el lenguaje nativo de CVC4, .smt2 para SMT-LIB v2, y .smt para SMT-LIB v1). Además, el usuario puede especificar el tipo de idioma de entrada mediante un indicador de línea de comando. También se puede acceder a CVC4 a través de la API de C ++.

- ***Core technology:*** CVC4 esta basado en la *lazy approach*. Para comprobar la satisfacibilidad, teóricamente CVC4 permite conectar diferentes *SAT Solvers*, sin embargo hasta ahora solo se admite MiniSAT.

- ***Theory Solver:*** CVC4 usa enfoques basados en modernos paradigmas DPLL(T) implementando las diferentes teorías de los *solvers*. La teoría del *solver* para funciones no interpretadas esta basada en el algoritmo de cierre de congruencia. En el caso de la teoría del *solver* para aritmética lineal, la implementación esta basada en el método simplex.

- ***Models and Proofs:*** Su sistema de prueba está diseñado para tener una huella absolutamente nula en memoria y tiempo. También soporta la *Logical Framework with Side Conditions (LFSC)* que es un lenguaje declarativo de alto nivel para definir sistemas de prueba y objetos de prueba para casi cualquier lógica. IFSC admite condiciones computacionales secundarias en las reglas de demostración, que facilitan el diseño de sistemas de prueba. Desafortunadamente, CVC4 aún no admite extracción de núcleo insatisfacible .

- ***Parallel solving:*** CVC4 brinda la oportunidad de ejecutar múltiples instancias de CVC4 en diferentes hilos. Aunque los lemas no se comparten entre subprocesos de forma predeterminada, existe una opción para hacerlo. Con esta opción activada, CVC4 puede compartir lemas de n literales, excluyendo los literales que son locales en un hilo y, por lo tanto, no son elegibles para ser compartidas. Las operaciones se pueden interrumpir si los resultados de otro hilo las hacen irrelevantes. Aunque ésta es una gran característica, todavía está en un estado experimental y, por lo tanto, debe usarse con precaución.

4.3 MathSAT 5

MathSAT 5 es un proyecto conjunto de *Fondazione Bruno Kessler* (FBK-irst) y *University of Trento*. Esta implementado en C++ y disponible gratuitamente para fines de investigación y evaluación académica. El formato por defecto en MatSAT 5 es SMT-LIB v2, aunque adicionalmente, soporta SMT-LIB v1.2 y formato DIMACS.

Es compatible con la mayoría de lógicas SMT-LIB expuestas en la sección 3.4.1, como la de igualdad y las funciones no interpretadas, aritmética lineal sobre enteros y reales, arrays, bitvectors y números reales de coma flotante, así como sus combinaciones.

Como CVC4 o Z3, MathSAT 5 y sus predecesores han participado en cada SMT-COMP desde 2005 hasta 2014. En 2005 MathSAT ganó las categorías de QF_UFLRA y QF_UFLIA. Dos años después, MathSAT volvió a ganar QF_UFLIA [11].

4.3.1. Características técnicas

MathSAT 5 ofrece una gran cantidad de características diferentes, como son[25]:

- ***User Interaction:*** Los usuarios pueden interaccionar con MathSAT 5 a través de linea de comandos o mediante el uso de archivos en formato SMT-LIB, siendo aceptados tanto v1.2 como v2.0 . El *solver* también acepta una entrada en formato DIMACS. Además, MathSAT 5 proporciona una API de C, que es similar a los comandos del lenguaje SMT-LIB 2.0 .
- ***Conjunctive Normal Form Converter:*** El codificador de restricciones de MathSAT 5 convierte cada fórmula de entrada en su CNF (*Conjunctive Normal Form*).
- ***Core technology:*** Por defecto, el núcleo de MathSAT 5 consiste en un *solver* SAT de estilo MiniSAT.
- ***Theory solver:*** MathSAT 5 consiste en *solvers* de teorías individuales basados en algoritmos de última generación. El *solver* para las funciones no interpretadas esta basado en el algoritmo de cierre de congruencia. En cuanto a la aritmética lineal en enteros y racionales, se utiliza un enfoque por capas basado en *simplex y branch & bound*. Para la teoría de coma flotante, MathSAT 5 implementa dos enfoques diferentes. Uno es basado en *lazy* o *eager bit-blasting*. El segundo y más reciente se basa en una combinación de *Interval Constraint Propagation* para números de coma flotante y modernos *Conflict-Driven Clause Learning (CDCL) SAT solvers*. Para la teoría de arrays, MatSAT 5 utiliza instanciación de axiomas. Para la teoría de bit-vectors, MatSAT 5 utiliza *lazy* o *eager bit-blasting* y la combinación de teorías es manejada por el marco de combinación de teorías de MathSAT 5.

- ***Models and Proofs:*** Además de decidir la satisfacción, Math-SAT 5 puede enumerar modelos con diferentes valores de verdad para fórmulas satisfacibles, o una prueba de insatisfacibilidades de fórmulas no satisfacibles. Además, puede extraer núcleos insatisfactorios o interpoladores de Craig [14].

  El teorema de interpolación de Craig describe una cierta relación entre dos fórmulas lógicas. Desde que este teorema se introdujo en el mundo de SMT, se hizo muy popular. Si consideramos un problema SMT para la teoría de *background* T y un par ordenado de fórmulas (A, B), tal que A y B no son satisfacibles bajo la teoría T, entonces un interpolante M Craig es una fórmula I para la cual [14]:

  - A satisface I bajo la teoría $T : A(\models)_T I$.
  - A y B no son satisfacibles bajo la teoría $T : A \wedge B(\models)_T \perp$.
  - I precede o es igual que A e I precede, o es igual a $B : I \preceq A$ y $I \preceq B$.

- ***AllSMT and Predicate Abstraction:*** MathSAT 5 implementa una funcionalidad AllSMT. Es decir, para una fórmula satisfacible, puede generar eficientemente un conjunto completo de asignaciones parciales coherentes con la teoría que satisfagan la fórmula.

- ***Pluggable SAT solvers:*** MathSAT 5 permite a sus usuarios integrar un *SAT solver* externo de su elección.

4.4 SMTInterpol

SMTInterpol es un *SMT solver* desarrollado en la universidad de Freiburg. Está implementado en Java y disponible con una licencia de software *GPU Lesser General Public License* (LGPL) v3. Como lenguaje de entrada, SMTInterpool soporta el lenguaje SMT-LIB v1.2 y v2 al igual que el formato DIMACS. El *solver* soporta las teorías de funciones no interpretadas con igualdad, aritmética lineal sobre enteros y reales y la combinación de estas teorías.

SMTInterpol también participo en el SMT-COMP en 2011 tanto en la modalidad principal como en el *application track* y fue capaz de resolver tantos problemas como el *solver* ganador pero con un tiempo de ejecución inferior. En 2012, SMT-COMP SMTInterpol fue ganador de código abierto para la teoría de QF_UFLIA y único competidor en el *proof generation track* [16].

Además, SMTInterpool y MathSAT 5 fueron los únicos participantes en 2012 en el *unsat core track*. Es decir, en el campo de la generación de pruebas y extracción de núcleos sin saturación, SMTInterpol es uno de los principales *SMT solver* disponibles.

4.4.1. Características técnicas

SMTInterpool es un *SMT solver* muy reciente. Sin embargo, provee un amplio rango de características para sus usuarios[25].

- ***User Interaction:*** Los usuarios pueden interaccionar con SMTInterpol a través de linea de comandos, o mediante el uso de archivos en formato SMT-LIB, siendo aceptados tanto v1.2 como v2. Además SMTInterpool incluye un parser para aceptar el formato DIMACS. Al mismo tiempo, proporciona una API Java modelada a partir de los comandos de este lenguaje. Los usuarios pueden emitir comandos a través de un archivo SMT-LIB, usar el canal de entrada estándar del *solver* o usar la API de Java.
- ***CNF Converter:*** SMTInterpol convierte cada fórmula de entrada en su CNF.
- ***Core technology:*** SMTInterpol esta basado en el paradigma DTLL (T) e interacciona con el SAT solver usando la lazy approach. El *SAT solver* implementado utiliza un motor CDCL. Es decir, el *SAT solver* se basa en el algoritmo DPLL, pero con la única diferencia de que su *backtracking* no funciona cronológicamente.
- ***Theory solver:*** SMTInterpol combina *solvers* para dos teorías, una para funciones no interpretadas, el cual esta basado en el algoritmo de cierre de congruencia y uno para aritmética lineal basada en el algoritmo simplex. Además, utiliza el procedimiento de combinación de teorías basado en combinación de modelos.
- ***Models and Proofs:*** El solver puede retornar modelos de fórmulas que son satisfacibles. Para fórmulas insatisfacibles, puede producir pruebas de resolución de las cuales puede extraer núcleos insatisfacibles o interpolantes Craig.

4.5 veriT

VeriT fue creado en un trabajo conjunto de la Universidad de Nancy, el Instituto Nacional de Investigación en Informática y Automática (INRIA) y la Universidad Federal de Río Grande del Norte (UFRN). Es una herramienta de código abierto escrita en C y distribuida bajo la licencia BSD.

VeriT admite SMT-LIB v2.0 y DIMACS como formato de entrada válido. Hasta 2011, VeriT proporcionó un procedimiento de decisión para la lógica de las fórmulas libres de cuantificadores sobre símbolos no interpretados, lógica de diferencia sobre números enteros y reales, y la combinación de los mismos. Desde entonces, el programa ha sido completamente reescrito para admitir también capacidades de razonamiento aritmético lineal y con cuantificador [8].

En 2014, en la SMT-COMP VeriT participó en las categorías de QF_IDL, QF_LIA, QF_LRA, QF_RDL, QF_UF y en sus combinaciones QF_AUFLIA, QF_UFIDL, QF_UFLIA, QF_UFLRA, así como en las categorías que permiten cuantificadores ALIA, AUFLIA, AUFLIRA, LIA, LRA, UF, UFLIA, UFLRA.

VeriT aún no ha ganado una categoría de un SMT-COMP, si bien esto se justifica por el hecho de que VeriT todavía está en sus primeras etapas.

4.5.1. Características técnicas

Las diferentes características de VeriT se analizan a continuación [25].

- ***User Interaction:*** VeriT soporta SMT-LIB v2.0 aunque pase el caso de que uno quiera utilizar VeriT como *SAT solver*, el lenguaje de entrada recomendado sería el DIMACS. Al mismo tiempo, también se puede acceder a VeriT a través de una API de C.
- ***Core technology:*** Esta basado en el paradigma DTLL (T) e interacciona con el *SAT solver* usando la *lazy approach*.
- ***Theory solver:*** VeriT utiliza un motor de razonamiento para la aritmética lineal basado en el método *Simplex*. El solver maneja funciones no interpretadas basadas en el método de cierre de congruencia. Además, VeriT integra cierto nivel de razonamiento cuantificadores a través *E-Maching*. Por otra parte, para combinar teorías VeriT utiliza el método de *Nelson-Oppen*.
- ***Models and Proofs:*** Se utiliza MiniSAT solver para producir modelos para la abstracción booleana de fórmulas de entrada. También puede producir trazas de prueba para fórmulas libres de cuantificadores que contienen funciones no interpretadas y aritmética. Desafortunadamente, VeriT aún no admite la extracción de núcleos insatisfactorios.

4.6 Yices 2

Yices 2 es un *SMT solver* eficiente desarrollado por el *Stanford Research Institute (SRI International)*. Es un software de código cerrado escrito en C y distribuido gratuitamente para uso personal bajo los términos de la licencia de Yices. Yices 2 puede decidir la satisfacción de fórmulas que consisten en combinaciones libres de cuantificadores de funciones no interpretadas con igualdad, aritmética lineal sobre enteros y reales, bit-vectors, arrays y lógica de enteros y diferencia real [24].

Como lenguaje de entrada, Yices 2 soporta SMT-LIB v1.2 y SMT-LIB v2.0. Al igual que su predecesor Yices 2 ha participado en la SMT-COMP en las ediciones de los años 2005 a 2009, volviendo a participar en la edición de 2014 [24].

Además, Yices 2 ganó a Z3 en 2008 en las categorías de QF_UF y QF_LRA. También, Yices 2 ha ganado las categorías de QF_AX, QF_UFLRA, QF_AUFLIA, QF_UFLIA, QF_UF, QF_RDL and QF_LRA. En 2014, Yices 2 ganó la categoría de QF_ALIA, QF_AUFLIA, QF_AX, QF_RDL, QF_UF y QF_UFBV.

4.6.1. Características técnicas

A continuación se analizan las diferentes características de Yices 2 [25]:

- ***User Interaction:*** Para interaccionar con Yices 2 se puede usar como lenguaje de entrada tanto SMT-LIB v1.2 como 2.0. Al mismo tiempo, es accesible mediante el uso de una API en C.
- ***Core technology:*** Debido a que es de código cerrado no se facilita esta información, aunque se intuye que por las funciones que desenvuelve esta basado en el paradigma DTLL (T) e interacciona con el SAT solver usando la lazy approach.
- ***Theory solver:*** Yices 2 actualmente implementa cuatro solucionadores de teorías diferentes configurados para funciones no interpretadas, aritmética lineal, arrays y vectores de bits, respectivamente. Es posible acoplar manualmente estos componentes con el *SAT solver* o eliminarlos individualmente, si no son necesarios, para optimizar el tiempo de ejecución para problemas específicos. El solver para funciones no interpretadas esta basado en el métodos de cierre de congruencia. Las teorías de aritmética lineal son manejadas por el solucionador de teorías basado en el algoritmo simplex. El procedimiento de decisión para la teoría de matrices utiliza la instanciación *lazy* de axiomas de matrices, mientras que los bit-vectors son manejados usando bit-blasting. Para combinar teorías, Yices 2 hace uso del método de combinación *Nelson-Oppen*.
- ***Models and Proofs:*** La API de Yices 2 proporciona funciones para crear modelos que asignan los símbolos de la fórmula a valores concretos. Desafortunadamente, no admite comandos para obtener pruebas o núcleos insatisfacibles.

4.7 Comparativa

En esta sección se va a proceder a realizar una comparativa entre las características que componen los distintos *SMT solver* analizados y las teorías que ofrecen a modo de resumen.

En la Tabla 4.1 podemos apreciar las distintas características que conforman los *SMT solvers* más utilizados, mientras que en la Tabla 4.2 se aprecia una comparación entre las teorías soportadas.

SMT-Solver	Propietario	Desarrollo	Licencia	Lenguaje entrada	API
Z3	Microsoft Research	C++	MSR-LA	SMT-LIB v2, DIMACS	C,C++,Python,Java,.NET
CVC4	Univ. de Nueva York e Iowa	C++	BSD	SMT-LIB v1.0/2, leng. nativo	C++
MathSAT5	FBK y Univ. de Trento	C++	Propietaria	SMT-LIB v1.2/2, DIMACS y leng. nativo	C
SMTInterpol	Univ. de Freiburg	Java	LGPL v3	SMT-LIB v1.2/2, DIMACS	Java
veriT	Univ. de Nancy, INRIA y UFRN	C	BSD	SMT-LIB v2.0, DIMACS	C
Yices2	Stanford Research Institute	C	Propietaria	SMT-LIB v2.0, lenguaje nativo	C

Tabla 4.1: Comparativa características SMT-Solvers.

SMT-Solver	Cuantificadores	Funciones no interpretadas	Aritmética lineal	Bit-vectors	Arrays	DataTypes	Aritmética no lineal
Z3	Si	Si	Si	Si	Si	Si	Si
CVC4	Si	Si	Si	Si	Si	Si	No
MathSAT5	No	Si	Si	Si	Si	No	Si
SMTInterpol	No	Si	Si	No	No	No	No
veriT	No	Si	Si	No	No	No	No
Yices2	No	Si	Si	Si	Si	No	No

Tabla 4.2: Comparativa teorías SMT-Solvers.

CAPÍTULO 5
El lenguaje de asertos

En este capítulo se presenta cómo transformar predicados a restricciones Z3. También se analizaran los distintos tipos algebraicos que ofrece Z3 y las operaciones que se pueden realizar sobre ellos.

Se muestran dos formas de trabajar, mediante el uso de lenguajes de alto nivel como Python (sección 5.2) y mediante la librería SMT-LIB v 2.0 (sección 5.3).

5.1 Introducción

Un aserto es una restricción que debe cumplir nuestro axioma. Para ello, se hace uso de expresiones básicas sobre booleanos y enteros. Las expresiones booleanas se relacionan entre sí mediante el uso de los operadores lógicos *and, or y not*. Por su parte, las expresiones sobre enteros se relacionan con los operadores $<$, $>$, $=$, $\leq$, $\geq$, etc. Estas expresiones se pueden utilizar conjuntamente con los cuantificadores $\exists$ y $\forall$ [13].

Los asertos pueden aplicarse a funciones específicas de ciertos tipos de datos como pueden ser listas, arrays, árboles, etc. Por ejemplo para expresar restricciones como que una lista no tiene una longitud mayor que 10 o que los elementos de un array deben estar ordenados.

5.2 Lenguajes de alto nivel

Como se ha enunciado en la sección 4.1.1, Z3 puede ser utilizando en lenguajes de alto nivel haciendo uso de su API disponible, en esta sección se ejemplificarán las distintas teorías de Z3 para el lenguaje de alto nivel Python.

5.2.1. Satisfacción y validez

Una fórmula o restricción F es válida si F siempre se evalúa como verdadero para cualquier asignación de valores apropiados a sus símbolos no interpretados. Por otro lado, una fórmula o restricción F es satisfacible si hay alguna asignación de valores apropiados a sus símbolos no interpretados bajo los cuales F se evalúa como verdadero.

La validez trata de encontrar una prueba a una declaración, mientras que la satisfacción trata de encontrar una solución a un conjunto de restricciones.

La satisfacibilidad de una fórmula o restricción puede comprobarse en Z3 de diferentes formas. Por un lado se puede definir un $Solver()$ al que se añaden las restricciones mediante el uso de la función $.add()$, y a continuación se usará $check()$ que nos devolverá si es satisfacible o no. En caso de desear ver el modelo usaremos $model()$.

En la Figura 5.1 se observa un ejemplo donde añadimos la restricción $x! = 0$ al *solver*, y luego comprobamos la satisfacibilidad. En caso de ser satisfacible también pedimos que nos devuelva un modelo.

```
s = Solver()
x = Const('x',IntSort())
s.add(x != 0)
s.check()
s.model()
```

Figura 5.1: Ejemplo satisfacibilidad Z3.

Como se observa en la Figura 5.2, ha dado como resultado un modelo satisfacible donde ha definido el valor de $x = 1$ para cumplir la restricción requerida.

```
[x = 1]
```

Figura 5.2: Ejemplo output satisfacibilidad Z3.

Una segunda manera de realizar este procedimiento es la utilización de *solve* donde simplemente definiendo $solve(x != 0)$ devolvería el mismo resultado que en la Figura 5.2.

A continuación veremos un ejemplo donde el *solver* devolverá que el modelo no es satisfacible ya que la restricción es imposible de cumplir. En la Figura 5.3 se pide al *solver* que cumpla con que x sea distinto de 0 y a la vez sea igual a 0, por tanto devolverá *unsat*.

```
s = Solver()
x = Const('x',IntSort())
s.add(x != 0,x==0)
s.check()
s.model()
```

Figura 5.3: Ejemplo no-satisfacibilidad Z3.

5.2.2. Aritméticas

Z3 nos permite representar y utilizar números enteros y reales con precisión, utilizando vectores de bits que podrán ser utilizados en nuestras formulas. Además podremos utilizar operadores aritméticos como son +, -, <, >, etc.

En la Figura 5.4 podemos apreciar un ejemplo de cómo sería su representación. En la primera línea se realiza la declaración de las variables de tipo real con nombres x e y. La función *solve* tiene por objetivo resolver un sistema de restricciones, de manera que se obligaría a que nos devolviera un valor de $x >= 1$, o que la suma de $x + y <= 3$.

```
x,y = Reals('x y')
solve(x>= 1,Or( x + y <= 3))
```

Figura 5.4: Ejemplo aritmético Z3.

Con este ejemplo *solve* generará un modelo que cumpla con las restricciones definidas anteriormente dando como resultado el modelo de la Figura 5.5, donde $y = 0, x = 1$.

```
[y = 0, x = 1]
```

Figura 5.5: Resultado obtenido del ejemplo aritmético Z3.

5.2.3. Arrays

Z3 cuenta con una teoría de arrays, los cuales se representan internamente como una función no interpretada de índices de valores de dominio infinito (aunque también podemos definir arrays con un tamaño fijo), caracterizada por las operaciones *select*, *store* , *k* , *map y ext*.

De las operaciones descritas anteriormente las más utilizadas son *Select* y *Store*, $Select(a, x)$ devuelve el elemento en la posición x del array a. La operación $Store(a, x, y)$ nos devuelve un array idéntico a a salvo en la posición x en la que ahora se almacenará el valor y.

En la Figura 5.6 se observa un ejemplo de la representación de arrays en z3:

```
x = Int('x')
y = Int('y')
array = Array('array', IntSort(), IntSort())

s = Solver()
s.add(Select(array, x) == x, Store(array, x, y) == array,x>0)
print (s.check())
print (s.model())
```

Figura 5.6: Ejemplo arrays Z3.

Cuyo resultado es el modelo de la Figura 5.7.

```
sat
[y = 1, array = Store(K(Int, 2), 1, 1), x = 1]
```

Figura 5.7: Resultado ejemplo arrays Z3.

Como se observa en la Figura 5.7, Z3 crea una variable auxiliar K con el fin de proporcionar una interpretación al array, y le da el valor 1 tanto a x como a y, cumpliendo con la condición de que sean iguales.

En caso de querer definir un array con un tamaño fijo de elementos se utilizará una declaración similar a $vec = IntVector('vec', 10)$.

Donde se ha definido la creación de un array de tipo entero con un tamaño de 10 elementos. Este tamaño no podrá ser modificado posteriormente.

5.2.4. Tipos algebraicos

Una de las principales ventajas que ofrece Z3 es la posibilidad de especificar algunas estructuras de datos más comunes como pueden ser los árboles, listas o tuplas; también se posibilita la opción de declarar tipos recursivos.

En la Figura 5.8 se observa un ejemplo de la representación de tipos algebraicos en Z3, en concreto de la definición del tipo lista, donde en el método *.declare(..)* se expone la funcionalidad de la lista, que en este caso es la posibilidad de insertar un elemento en la misma o de que esté vacía (*nil*) .

```
def funList(sort):
    List = Datatype('List')
    #Constructor insert: (Int, List) -> List
    List.declare('insert', ('head', sort), ('tail', List))
    List.declare('nil')
    return List.create()

List = funList(IntSort()) #Se crea una lista
x    = Const('x', List)

solve(x!=List.nil)
```

Figura 5.8: Ejemplo tipos algebraicos Z3.

La variable x es una constante de tipo lista, que se utilizará en el ejemplo para comprobar si x puede ser distinto de una lista vacía.

Como se observa en la Figura 5.9, Z3 devuelve el modelo donde el elemento 0 está en cabeza de la lista.

```
[x = insert(0, nil)]
```

Figura 5.9: Resultado ejemplo tipos algebraicos Z3.

5.2.5. Lógica proposicional:

Las expresiones booleanas pueden utilizarse en Z3 gracias al tipo predefinido $Bool$, en el caso de Z3 se soporta los operadores $And, Or, Not, Implies$(implicación), If(representa la estructura if-then-else).

En la siguiente Figura 5.10 se observa un ejemplo de la representación de la lógica proposicional en z3 utilizando la implicación entre otros operadores:

```
p = Bool('p')
q = Bool('q')
r = Bool('r')
solve(Implies(q, p), r == Not(p), Or(Not(q), r))
```

Figura 5.10: Ejemplo tipos booleanos con Z3

5.2.6. Funciones:

En los lenguajes de programación convencionales, las funciones pueden producir efectos laterales, como lanzar excepciones o caer en una ejecución infinita. Las funciones en Z3 no tienen estos efectos laterales.

Más concretamente, las funciones en Z3 son no interpretadas, lo que significa que son muy flexibles: permiten cualquier interpretación que sea consistente con las restricciones sobre la función. En contraste, un ejemplo de función interpetada sería la función $+$, la cual tiene una interpetación estándar fija (realiza la suma de dos números).

En la Figura 5.11 vemos un ejemplo de cómo funcionan las funciones en Z3. En el ejemplo primero se declaran dos variables de tipo entero x e y. Posteriormente se declara la función no interpretada de nombre $funcion$ que toma como argumento un entero y devuelve un valor entero. En Z3 los tipos se denominan sorts, por lo que $IntSort$ denota un tipo entero.

```
x = Int('x')
y = Int('y')
funcion = Function('funcion', IntSort(), IntSort())
solve(funcion(x) == y, x != y)
```

Figura 5.11: Ejemplo función en Z3.

El resultado de ejecutar el ejemplo genero el modelo de la Figura 5.12 en cual se da el valor 0 a x e el valor 1 a y cumpliendo la restricción definida en que la $funcion(x) == y$ & $x! = y$.

```
[x = 0, y = 1, funcion = [else -> 1]]
```

Figura 5.12: Resultado obtenido función en Z3.

5.2.7. Cuantificadores:

Z3 puede resolver problemas sin cuantificadores que contengan aritmética, vector de bits, booleanos, matrices, funciones y tipos de datos. Además, Z3 también puede trabajar con fórmulas que usan cuantificadores.

En la Figura 5.13 se observa un ejemplo de la representación de los cuantificadores en Z3, el cual define $\forall x, f(x, x) == 0$.

```
f = Function('f', IntSort(), IntSort(), IntSort())
x, y = Ints('x y')
a, b = Ints('a b')
solve(ForAll(x, f(x, x) == 0))
```

Figura 5.13: Ejemplo cuantificadores en Z3.

La Figura 5.14 muestra el modelo devuelto donde f tiene asignado el valor 0.

```
[f = [else -> 0]]
```

Figura 5.14: Resultado obtenido cuantificadores en Z3.

5.3 SMT-LIB v2.0

Ahora vamos a proceder a ejemplificar las distintas teorías de Z3 utilizando el lenguaje SMT-LIB v2.0 .

5.3.1. Satisfacibilidad y validez

Para comprobar la satisfacibilidad de una fórmula o restricción en Z3. Se definen las restricciones usando el comando *assert*, el cual se encargará de añadir la restricción a la pila interna de Z3. Haciendo uso del comando *(check-sat)* se determina si las fórmulas actuales en la pila Z3 son satisfacibles o no. Si las fórmulas son satisfacibles, Z3 devolverá *sat*, en caso contrario devolverá *unsat*. En caso de desear obtener el modelo, usaremos el comando *(get-model)*.

En la Figura 5.15 se observa un ejemplo donde añadimos la restricción $x! = 0$ al *solver*, y luego comprobamos la satisfacibilidad y en caso de ser satisfacible nos devuelve un modelo.

```
(declare-const x Int)

(assert (not(= x 0)))

(check-sat)
(get-model)
```

Figura 5.15: Ejemplo satisfacibilidad y validez Z3.

La Figura 5.16 muestra el modelo donde ha definido $x = 1$ para cumplir la restricción requerida.

```
sat
(model
  (define-fun x () Int
    1)
)
```

Figura 5.16: Ejemplo output satisfacibilidad y validez Z3.

La Figura 5.17 muestra un ejemplo de fórmula no satisfacible. Se pide al *solver* que cumpla con que x sea distinto de 0 y a la vez igual a 0, dando como resultado *unsat*.

```
(declare-const x Int)

(assert (not(= x 0)))
(assert (= x 0))
(check-sat)
```

Figura 5.17: Ejemplo unsat satisfacibilidad y validez Z3.

5.3.2. Aritméticas

Al igual que si utilizamos Z3 haciendo uso de un lenguaje de alto nivel, Z3 nos permite representar e utilizar números enteros y reales con precisión, utilizando vectores de bits que podrán ser utilizados en nuestras fórmulas.

En la Figura 5.18, se observa el ejemplo visto anteriormente donde se define dos constantes de tipo entero y se define dos restricciones de manera que $x >= 1$ o que la suma de $x + y <= 3$.

```
(declare-const x Int)
(declare-const y Int)
(assert (>= x 1))
(assert (<= (+ y x) 3))
(check-sat)
(get-model)
```

Figura 5.18: Ejemplo aritmético usando SMT-LIB v2.0 .

El comando *(get-model)* generará un modelo en caso de que se cumplan las restricciones definidas en el ejemplo del modelo de la Figura 5.19, donde $y = 2$ e $x = 1$.

```
sat
(model
  (define-fun x () Int
    1)
  (define-fun y () Int
    2)
)
```

Figura 5.19: Output ejemplo aritmético usando SMT-LIB v2.0 .

5.3.3. Arrays

Ya sabemos que los arrays en Z3 son infinitos. Aquí también se dispone de $Select(a, x)$ y $Store(a, x, y)$ con el mismo comportamiento que el visto en la sección 5.2.3.

La Figura 5.20 muestra un ejemplo de la representación de un array.

```
(declare-const x Int)
(declare-const a1 (Array Int Int))

(assert (= (select a1 x) x))
(check-sat)
(get-model)
```

Figura 5.20: Ejemplo arrays usando SMT-LIB v2.0 .

Obteniendo el modelo de la Figura 5.21 como resultado:

```
sat
(model
  (define-fun a1 () (Array Int Int)
    (_ as-array k!0))
  (define-fun x () Int
    0)
  (define-fun k!0 ((x!0 Int)) Int
    (ite (= x!0 0) 0
      0))
)
```

Figura 5.21: Output ejemplo arrays usando SMT-LIB v2.0 .

En la Figura 5.21 se aprecia como Z3 asigna el valor 0 a la variable x, además Z3 crea la función auxiliar k!0 para asignar una interpretación a la constante de matriz a1.

En caso de querer definir un array con un tamaño fijo de elementos, se utilizará la siguiente declaración:

```
(declare-const all1 (Array Int Int))
(declare-const i Int)
(assert (= all1 ((as const (Array Int Int)) 1)))
(assert (= i (select all1 i)))
(check-sat)
(get-model)
```

Figura 5.22: Ejemplo arrays de tamaño estático usando SMT-LIB v2.0 .

En la Figura 5.22 $((as\ const\ (Array\ Int\ Int))\ 1)$ define un array con un tamaño estático, en este caso con un solo elemento. Obteniendo el modelo resultante en la Figura 5.23:

```
sat
(model
  (define-fun all1 () (Array Int Int)
    (_ as-array k!0))
  (define-fun i () Int
    1)
  (define-fun k!0 ((x!0 Int)) Int
    (ite (= x!0 1) 1
      1))
)
```

Figura 5.23: Output ejemplo arrays de tamaño estático usando SMT-LIB v2.0 .

En este modelo, al igual que en la Figura 5.21 Z3 crea la función auxiliar k!0 para asignar una interpretación a la constante del array all1. Por otra parte, se ha asignado a *i* el valor de 1, por tanto, el valor del elemento que conforma el array será 1.

5.3.4. Tipos de datos

Con Z3 podemos especificar estructuras de datos como árboles o listas. Además permite la recursividad.

En la Figura 5.24 se ejemplifica la definición del tipo lista, utilizando el tipo genérico T, exponiendo los constructores de la lista, que en este caso es la posibilidad de acceder a la cabeza de la lista (hd), a la cola (tl) o que este vacía (nil) .

```
(declare-datatypes (T) ((Lst nil (cons (hd T) (tl Lst)))))
```

Figura 5.24: Ejemplo definición lista usando SMT-LIB v2.0 .

5.3.5. Lógica proposicional

La definición y utilización de expresiones booleanas se definen gracias al tipo Bool, que presenta soporte para los operadores *and*, *or*, *xor*, *not*, =>(implicación), *ite* (if-then-else).

En la siguiente Figura 5.25 se muestra un ejemplo de la lógica proposicional en Z3:

```
(declare-const p Bool)
(declare-const q Bool)
(assert (=> p q))
(check-sat)
(get-model)
```

Figura 5.25: Ejemplo lógica proposicional usando SMT-LIB v2.0 .

Donde el axioma de esta fórmula proposicional sería $(p \rightarrow q)$ donde al declarar que q implica p lo que estamos indicando es que en caso de que q sea cierto p también lo será, pero en caso de q sea falso, no tiene por que serlo p.

En este caso, como no se ha producido ninguna restricción de que valor tiene que tener p o q Z3 generará un modelo con un valor aleatorio que cumpla la restricción de la implicación como se observa en la Figura 5.26.

```
sat
(model
  (define-fun q () Bool
    false)
  (define-fun p () Bool
    false)
)
```

Figura 5.26: Output ejemplo lógica proposicional usando SMT-LIB v2.0 .

5.3.6. Funciones

En la Figura 5.27 vemos un ejemplo de definición de función en Z3 con la notación de SMT-LIB. Se ha definido dos funciones *min* y *max* que dados dos valores devuelven el mínimo y el máximo respectivamente.

```
(declare-const b Int)
(define-fun min ((a Int) (b Int)) Int
     (ite
       (< a b)
        a
        b
     )
)
(define-fun max ((a Int) (b Int)) Int
     (ite
       (> a b)
        a
        b
     )
)

(assert (= b (min 10 1)))

(check-sat)
(get-model)
```

Figura 5.27: Funciones min y max usando SMT-LIB v2.0 .

Al ejecutar el ejemplo tenemos como resultado un modelo (Figura 5.28) en cual asigna el valor 1 a la constante *b* que es el mínimo entre 10 y 1, cumpliendo con la restricción definida.

```
sat
(model
  (define-fun b () Int
    1)
)
```

Figura 5.28: Output funciones min y max usando SMT-LIB v2.0 .

5.3.7. Cuantificadores

Ya sabemos que Z3 también acepta y puede trabajar con fórmulas que usan cuantificadores.

En la Figura 5.29 se ejemplifica la utilización de los cuantificadores en Z3 con la notación SMT-LIB v2.0 .

```
(declare-fun f (Int Int) Int)
(assert (forall ((x Int)) (= (f x x) 0)))
(check-sat)
(get-model)
```

Figura 5.29: Ejemplo cuantificadores usando SMT-LIB v2.0 .

Obteniendo como resultado un modelo donde se observa (Figura 5.30) que se ha asignado a f el valor de 0 para cumplir con la restricción.

```
sat
(model
  (define-fun f ((x!0 Int) (x!1 Int)) Int
    0)
)
```

Figura 5.30: Output ejemplo cuantificadores usando SMT-LIB v2.0 .

CAPÍTULO 6
Validación automática de contratos software

En este capítulo se describe el proceso de la implementación de nuestro proyecto para la validación automática de contratos software usando Z3. El sistema desarrollado está accesible a través de una página web donde se han integrado los distintos servicios propuestos. Al mismo tiempo se ha llevado a cabo un despliegue de esta página web, utilizando Google Cloud Platform (GCP). La dirección de acceso es la siguiente: `http://104.155.40.246:8080/`.

6.1 Arquitectura del sistema

Uno de los aspectos más importantes en la implementación de un proyecto software es utilizar una arquitectura software adecuada.

La arquitectura software es el diseño de más alto nivel de la estructura de un sistema encarnado en sus componentes, las relaciones entre ellos y el ambiente, además de los principios que orientan su diseño y evolución [36].

Existen un gran número de arquitecturas software, siendo las más comunes las siguientes [33]:

- **Arquitectura cliente-servidor**: Es un modelo de arquitectura en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores y los demandantes, llamados clientes.
- **Arquitectura pipeline**: Este tipo de arquitectura se centra en que los sistemas se encadenan para ofrecer una funcionalidad donde cada parte tiene una responsabilidad limitada y clara, reduciendo el número de dependencias entre los sistemas.
- **Arquitectura en capas**: Este modelo de arquitectura se centra en un modelo de desarrollo de software donde el objetivo es la separación en partes de los componentes del sistema.
- **Arquitectura modelo-vista-controlador**: Este tipo de arquitectura separa los datos de una aplicación, la interfaz de usuario y la lógica de control en tres componentes distintos.

Para la utilización de este proyecto se ha decidido utilizar la arquitectura cliente-servidor debido a que es el tipo de arquitectura que más encaja en el sistema que se desea implementar.

Tendremos por un lado el *client*, que actuará como un *frontend*, siendo la parte del software donde interactúan los usuarios, mientras que el *server* será el *backend* que se encargará de procesar los datos introducidos por el usuario y realizar las acciones necesarias.

6.2 Programas

Un programa está formado por una o más funciones modificadores y un conjunto de observadores como los definidos en la sección 6.2.1. En esta sección, se definirá un programa *insert* que se encargue de añadir un elemento a una lista en caso de cumplirse unas condiciones concretas (que no este llena y que el elemento no pertenezca ya a la lista).

La Figura 6.8 evidencia el programa, donde para poder insertar un elemento en la lista, debe satisfacer que ese elemento no forme ya parte de la lista (para evitar elementos repetidos) y que la lista no exceda de su capacidad máxima.

```
(define-fun-rec len((l (Lst Int))) Int
    (ite
        (= l nil) 0
        (+ (len (tl l)) 1)
    )
)

(define-fun isFull  ((l (Lst Int))) Bool
  (ite (< (len l) 10)
    false
    true
  )
)

(define-fun-rec isMember((i Int) (l (Lst Int))) Bool
  (ite
    (= l nil) false
    (or (= i (hd l)) (isMember i (tl l)))
  )
)

(define-fun insert ((i Int) (l (Lst Int))) (Lst Int)
  (ite (or (isMember i l) (isFull l))
       l
       (cons i l))
)
```

Figura 6.1: Programa insert .

6.2.1. Observadores

Los *observadores* son funciones que permiten construir un contrato para un programa o sistema. En este proyecto se pretende verificar o falsificar los contratos. Por lo que, será de gran interés caracterizar de forma precisa como se han diseñado.

Para la definición de los observadores se ha tenido que definir el tipo lista, que no existe de forma nativa en Z3, por lo que se ha definido como se indica en la Figura 6.2.

```
(declare-datatypes (T) ((Lst nil (cons (hd T) (tl Lst)))))
```

Figura 6.2: Definición tipo lista .

A continuación procedemos a mostrar los distintos observadores desarrollados.

Length

La función length, devuelve la longitud de una lista dada. Para ello se realiza el calculo de forma recursiva (como se aprecia en la Figura 6.3). En el caso base, la función devuelve cero y la lista está vacía; en el caso general incrementa en 1 la longitud del resto de la lista.

```
(define-fun-rec len((l (Lst Int))) Int
    (ite
        (= l nil) 0
        (+ (len (tl l)) 1)
    )
)
```

Figura 6.3: Observador length .

isMember

Como se observa en la Figura 6.4, está función recursiva devuelve *true* en caso de que un valor dado pertenezca a una lista (también dada) o $false$ en caso contrario.

El caso base de está función es cuando la lista está vacía y devuelve *false*, ya que un elemento no puede pertenecer a una lista vacía. Por otro lado, en el caso general, se comprueba si la cabeza de la lista coincide con el valor dado, o bien si el elemento pertenece al resto de la lista (haciendo esta comprobación mediante el uso del *or*).

```
(define-fun-rec isMember((i Int) (l (Lst Int))) Bool
  (ite
    (= l nil)
    false
    (or (= i (hd l)) (isMember i (tl l)))
  )
)
```

Figura 6.4: Observador isMember .

isFull

La función isFull se detalla en la Figura 6.5 donde se comprueba si una lista tiene una capacidad determinada de valores, para ello se hará una llamada a la función *len* para obtener la cantidad de elementos que forman parte de la lista de manera que en caso de que se sobrepase la cantidad delimitada (en este caso 10) devolverá *true*, mientras que en caso contrario devolverá *false*.

```
(define-fun isFull  ((l (Lst Int))) Bool
  (ite (< (len l) 10)
    false
    true
  )
)
```

Figura 6.5: Observador isFull .

isEmpty

Esta función (que se observa en la Figura 6.6) se encarga de comprobar si una lista está vacía o no. Para ello hace una llamada a la función *len* que recursivamente calcula el número de elementos que forman la lista. Devolviendo *true* en caso de que ese número de elementos sea 0, y *false* en caso contrario.

```
(define-fun isEmpty ((l (Lst Int))) Bool
  (ite
    (= 0 (len l))
    true
    false
  )
)
```

Figura 6.6: Observador isEmpty .

sortedList

La función recursiva *sortedList* tiene como objetivo comprobar si una lista está ordenada ascendentemente o no (como se aprecia en la Figura 6.7).

En este caso, esta función tiene dos casos base. Por un lado, en caso de que la lista este vacía se devolverá *true*, ya que estará ordenada, por otro lado, en caso de que la cola de la lista sea *nil* (nulo), significará que la lista solo tiene un elemento y por tanto está ordenada.

En el caso general, se comprueba si el elemento en la cabeza de la lista es menor que el siguiente y en caso de cumplirse se comprueba recursivamente con la lista que contiene ese segundo elemento y el resto de elementos de la lista. En caso de no cumplirse, se devuelve $false$.

```
(define-fun-rec sortedList ((l (Lst Int))) Bool
  (ite(= l nil)
  true
  (ite(= (tl l) nil)
    true
    (and (< (hd l) (hd (tl l))) (sortedList (tl l))))
  )
)
```

Figura 6.7: Observador sortedList .

6.3 Contratos Software

Los contratos son anotaciones que documentan y especifican formalmente el comportamiento de un programa para la verificación formal de los mismos. Inferir contratos resulta útil para fortalecer las especificaciones escritas por el programador.

En la Figura 6.8 se observa un ejemplo de contrato completo donde se aprecia un axioma cierto obtenido de la herramienta KindSpec que es una herramienta correcta de síntesis de contratos.

Un axioma está formado por una implicación cuyo antecedente y consecuente son conjunciones de ecuaciones *observador* = *valor*. Los elementos que forman un axioma son los observadores.

Es importante resaltar, que las variables que tienen un '?' delante, son variables simbólicas, por tanto desconocemos el valor que pueden adquirir.

```
**********************************************
RESULTING INFERRED CONTRACT
**********************************************
-----------------------------------------------
PRECONDITION P:
(length(x)=0 ^ isNull(x)=1) ||
(isNull(x)=0 ^ length(x)=?l0 ^ ?l0 >= 1)
-----------------------------------------------
-----------------------------------------------
POSTCONDITION Q:
A1: (length(x)=0 ^ isNull(x)=1) => (length(x)=0 ^ isNull(x)=1 ^ ret=0) ^
A2: (isNull(x)=0 ^ length(x)=?l0 ^ ?l0 >= 1) => (length(x)=0 ^ isNull(x)=1 ^ ret=1)
-----------------------------------------------
-----------------------------------------------
LOCATIONS L:
n
y
x
-----------------------------------------------
-----------------------------------------------
CANDIDATE AXIOMS Q#:

-----------------------------------------------
```

Figura 6.8: Ejemplo contrato correcto .

En la Figura 6.9 tenemos otro ejemplo de contrato, pero a diferencia del anterior los axiomas se entregan como 'candidatos' lo que significa que no son correctos por construcción sino pendientes de verificar o falsificar.

```
**********************************************
RESULTING INFERRED CONTRACT
**********************************************
-----------------------------------------------
PRECONDITION P:
(isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isFull(s)=1 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=1)
-----------------------------------------------
-----------------------------------------------
POSTCONDITION Q:
A1: (isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A2: (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A3: (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)=3 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A4: (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0 ^
ret=0) ^
A5: (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0 ^
ret=0) ^
A6: (isFull(s)=1 ^ isNull(s)=0) => (isFull(s)=1 ^ isNull(s)=0 ^ ret=0) ^
A7: (isEmpty(s)=0 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=1) => (isEmpty(s)=0 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=1 ^
ret=0)
-----------------------------------------------
-----------------------------------------------
LOCATIONS L:
new_node
new_node->value
new_node->next
s->elems
s->lsize
n
found
-----------------------------------------------
-----------------------------------------------
CANDIDATE AXIOMS Q#:
C1: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2) => (isEmpty(s)=0 ^ length(s)=?l0 + 2 ^ contains(s,x)=1 ^
isNull(s)=0 ^ ?l0 >= 2 ^ ret=1) ^
C2: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2) => (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=1 ^
isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=0) ^
C3: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=?c ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2) => (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=?c ^
isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=0)
-----------------------------------------------
```

Figura 6.9: Ejemplo contrato con axiomas candidatos .

A continuación, se va a proceder a especificar el axioma candidato C3 (Figura 6.10).

$$\begin{pmatrix} \texttt{isNull(list) = False ^ isEmpty(list) = False ^} \\ \texttt{isFull(list) = False ^ isMember (list, x) = ?b ^} \\ \texttt{len(list) = ?n + 1} \end{pmatrix} \Rightarrow \begin{pmatrix} \texttt{isNull(copy) = False ^ isEmpty(copy) = False ^} \\ \texttt{isFull(copy) = False ^ isMember (copy, x) = ?b ^} \\ \texttt{len(copy) = ?n + 1} \end{pmatrix}$$

Figura 6.10: Axioma candidato C3.

Hay que destacar que el antecedente es un estado anterior de la ejecución del modificador y el consecuente es una descripción del estado posterior a la ejecución del modificador. Siendo en este caso el modificador el programa *insert*.

Hay que tener en cuenta un aspecto importante de Z3 y es que las variables no se pueden referenciar, por lo que cuando Z3 asigna un valor a una variable o lista está no puede cambiar, para solucionar este problema se crea una lista copia de nombre *copy*.

Por otro lado, ?*b* es una variable simbólica, por lo que se desconoce si Z3 la evaluará a verdadero o falso. Mientras que, ?*n* al igual que ?*b* es una variable simbólica donde se desconoce el valor que Z3 le otorgará (siendo en este caso de tipo entero y no booleano).

La manera de transformar la Figura 6.10 a la sintaxis de SMT-LIB v2.0 es la que se aprecia en la Figura 6.11. Donde el primer *assert* corresponde a la precondición del axioma, en el segundo *assert* se aplicará el modificador y finalmente en el tercer *assert* será el consecuente del estado posterior de la ejecución del modificador.

```
(assert (and (= false (isNull listInsert))(= false (isEmpty listInsert)) (= false (isFull listInsert))
(= b (isMember x listInsert))  (= (+ n 1) (len listInsert)))  )

(assert (= copy (insert x listInsert)))

(assert (and (= false (isNull copy))(= false (isEmpty copy)) (= false (isFull copy))  (= b
(isMember x copy))  (= (+ n 1) (len copy)) ))
```

Figura 6.11: Axioma candidato usando el lenguaje SMT-LIB v2.0.

6.4 Página Web

Para la integración de los observadores y posibilitar que los usuarios puedan crear los suyos propios para escribir y validar sus contratos software, se ha optado por el desarrollo de una página web que permita está funcionalidad.

6.4.1. Diagrama de Casos de uso

Los diagramas de casos de uso generalmente se denominan diagramas de comportamiento y se utilizan para describir un conjunto de acciones (casos de uso) que algunos sistemas (sujetos) deben o pueden realizar en colaboración con uno o más usuarios externos del sistema (actores) [2].

Cada caso de uso debe proporcionar algún resultado observable y valioso para los actores u otras partes interesadas del sistema [2]. En la Figura 6.12 se representa el diagrama de casos de uso de nuestro proyecto.

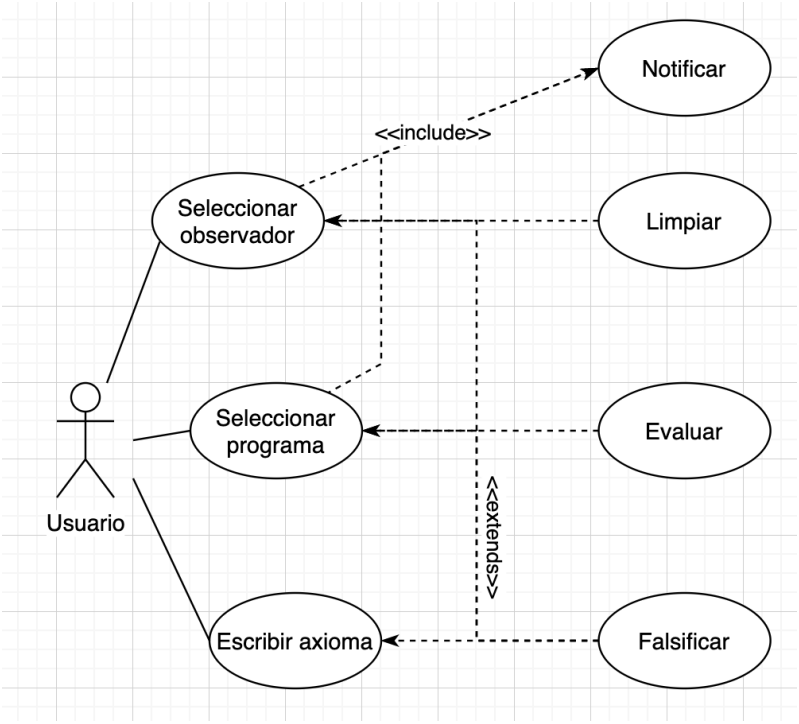

Figura 6.12: Diagrama de casos de uso .

6.4.2. Tecnologías utilizadas

Para el desarrollo de esta web se han utilizado diferentes tecnologías y lenguajes. Principalmente se ha utilizado HTML y CSS para el aspecto visual, mientras que para controlar el comportamiento de la web desde la parte del cliente se ha utilizado jQuery (JavaScript). En el lado del servidor se ha optado por la utilización de Flask, el cuál es un framework de Python que combinándolo con la API de Z3 nos permite evaluar los axiomas obteniendo modelos resultantes.

A continuación se describe cada una de estas tecnologías:

HTML

El HTML también conocido como 'Lenguaje de marcación de Hipertexto' (*Hyper Text Markup Language*) es un estándar utilizado en la WWW (*World Wide Web*) que utiliza un sistema de etiquetas que no requiere ser compilado. Fue creado en el año 1980 por el físico Tim Berners-Lee con el fin de proponer un nuevo hipertexto para compartir documentos.

Los sistemas de hipertexto habían sido desarrollados unos años antes. En el ámbito de la informática, el hipertexto permitía que los usuarios accedieran a la información relacionada con los documentos electrónicos que estaban visualizando[1].

[1]HTML: https://uniwebsidad.com/libros/xhtml/capitulo-1/breve-historia-de-html

CSS

Hojas de Estilo en Cascada (*Cascading Style Sheets*), es un mecanismo simple que describe cómo se va a mostrar un documento en la pantalla, o cómo se va a imprimir, o incluso cómo va a ser pronunciada la información presente en ese documento a través de un dispositivo de lectura. Esta forma de descripción de estilos ofrece a los desarrolladores el control total sobre estilo y formato de sus documentos[2].

JavaScript

JavaScript es un lenguaje de programación interpretado desarrollado por Netscape que suele utilizarse en el lado del cliente, de forma que se implementa como parte de un navegador con el fin de mejorar la interfaz de usuario y páginas web dinámicas. Para interactuar con una página web, JavaScript provee de una implementación del *Document Object Model* (DOM)[3].

jQuery

jQuery es una biblioteca de JavaScript creada inicialmente por John Resig rápida, pequeña y rica en funciones. Hace que cosas como el desplazamiento y la manipulación de documentos HTML (DOM), el manejo de eventos, la animación y la agregación de interacción mediante Ajax a páginas web sean mucho más simples con una API fácil de usar que funciona en una multitud de navegadores. Con una combinación de versatilidad y extensibilidad, jQuery ha cambiado la forma en que millones de personas escriben JavaScript[4].

JSON

JSON es un formato de texto sencillo para el intercambio de datos. Originalmente formaba parte de JavaScript, pero con el paso del tiempo debido a su amplia aceptación como una opción a XML, se constituyo como un formato independiente [5]. En nuestro caso se utilizará la librería jsonify para interaccionar con objetos de tipo JSON en Python.

Python

Python es un lenguaje de programación interpretado, orientado a objetos y de alto nivel con semántica dinámica. Uno de los grandes atractivos que ofrece Python es la sintaxis simple y fácil de aprender de enfatizando la legibilidad y, por lo tanto, reduce el costo del mantenimiento del programa. También admite mó-

[2]CSS: https://www.w3c.es/Divulgacion/GuiasBreves/HojasEstilo

[3]JavaScript:https://developer.mozilla.org/es/docs/Web/JavaScript/Acerca_de_JavaScript

[4]jQuery:https://jquery.com/

[5]JSON: https://www.json.org/json-es.html

dulos y paquetes, lo que fomenta la modularidad del programa y la reutilización de código [6].

Flask

Flask es un framework de Python cuyo objetivo principal es crear aplicaciones web con el menor número de líneas de código. Está basado en la especificación WSGI (*Web Server Gateway Interface*) de Werkzeug y el motor de plantillas Jinja2[7].

Z3Py

Z3Py es la librería de Z3 para Python, permitiendo la interacción del *SMT Solver* Z3 con el lenguaje de programación Python. Ofreciendo la capacidad para utilizar las distintas teorías que ofrece Z3 adaptadas a una sintaxis propia en Python, o el uso del lenguaje SMT-LIB v1.0 / v2.0 para la utilización de estas teorías[8].

Argparse

El módulo argparse incluye herramientas para construir procesadores de argumentos y opciones de línea de comando. Las acciones admitidas incluyen almacenar el argumento (individualmente o como parte de una lista), almacenando un valor constante cuando se encuentra un argumento (incluido un manejo especial para verdadero/falso valores para modificadores booleanos). Además, el módulo argparse también genera automáticamente mensajes de ayuda e uso y emite errores cuando los usuarios le dan argumentos inválidos al programa [9].

REST

Los servicios REST a diferencia de lo que comúnmente se suele interpretar no son un estándar sino un patrón de diseño. Una de las particularidades de REST es el ser *stateless*, lo que significa que cada solicitud del cliente al servidor debe incluir toda la aclaración imprescindible para poder entender la solicitud y no puede aprovechar ningún contexto almacenado en el servidor [28].

Para referirnos a términos de información en REST se emplea la palabra recursos. Estos recursos se organizan mediante una URI (*Universal Resource Identifier*). De esta forma, para interactuar con los servicios REST, se accederá a la URI del recurso y se utilizará la operación HTTP pertinente (utilizándose comúnmente, GET, PUT, POST o DELETE)[10].

[6] Python: https://www.python.org/
[7] Flask: https://flask.palletsprojects.com/en/1.1.x/
[8] Z3Py: https://ericpony.github.io/z3py-tutorial/guide-examples.htm
[9] Argparse: https://docs.python.org/3/library/argparse.html
[10] REST:https://restfulapi.net/

Logging

Esta librería tiene como objetivo definir funciones y clases que implementan un sistema flexible de registro de eventos para aplicaciones y bibliotecas. Una de las particularidades es que logging contiene todos los registros importantes del historial de eventos. De manera, que dependiendo del tipo de seguimiento que queramos hacer, solo se registran ciertas acciones o eventos de un proceso o, por el contrario, se comprueban todas las acciones[11].

6.4.3. Implementación detallada

La arquitectura del sistema se ilustra en la siguiente Figura:

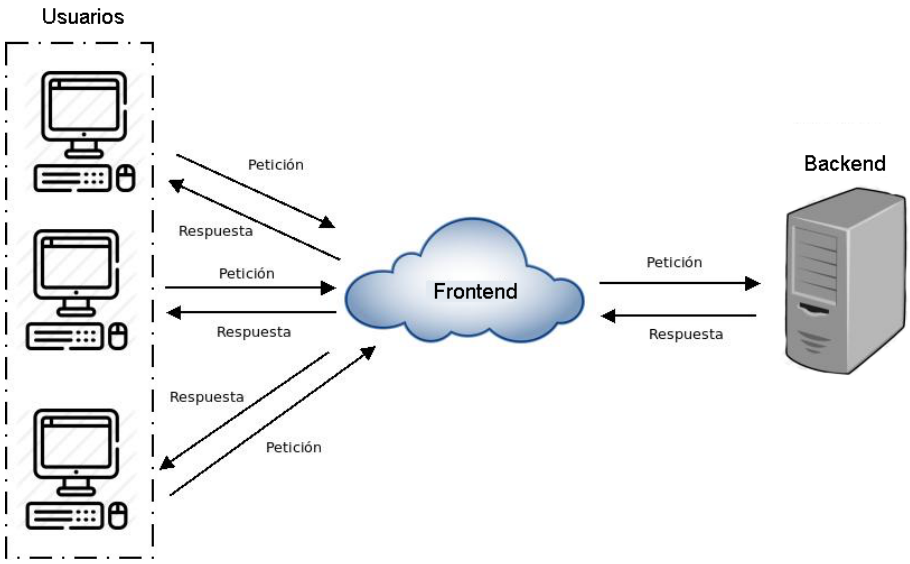

Figura 6.13: Arquitectura del sistema .

A continuación se mostrará una descripción más detallada de las partes más significativas de la implementación del proyecto.

Backend

El *backend* del proyecto esta conformado por un archivo *server.py* donde, por un lado, en el método main (Figura 6.14) se definen los argumentos que necesitará la aplicación para lanzarse. Profundizando en más detalle, se utiliza la librería *Argparse* para añadir funcionalidad en la linea de comandos, habilitando opciones al lanzar *server.py* como por ejemplo *pyhon3 server.py -p 900* que haría que el puerto que se lanzara la aplicación fuera el 900 (en nuestro caso se habilita que por defecto se utilice el puerto 8000). Una vez definidas, estás opciones se encapsulan utilizando *parse_args()* y se invoca el método *init_server(args)*. Finalmente se ejecuta el archivo en la dirección 0.0.0.0:8000, que es la dirección y puerto definidos por defecto.

[11] Logging: https://docs.python.org/3/library/logging.html

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Process texts.')
    parser.add_argument('-o', '--observators', default='./observadores')
    parser.add_argument('-r', '--programs', default='./programas')
    parser.add_argument('-p', '--port', default=8000)

    args = parser.parse_args()
    app = init_server(args)
    app.run(debug=True, host='0.0.0.0', port=args.port)
```

Figura 6.14: Método main .

El método *init_server(args)* utiliza el *framework Flask* y la librería *logging*, como se observa en la Figura 6.15. Primero se define la lógica de la aplicación haciendo la llamada *Flask(__name__)* siendo necesario para que *Flask* sepa dónde buscar plantillas, archivos estáticos, etc. Por otra parte, *logging* se encargará de crear un sistema flexible de registro de eventos para aplicaciones y bibliotecas, donde en las siguientes líneas se ha configurado el nivel y el formato deseado para el registro de eventos de la aplicación.

Finalmente se definen los métodos *main()*, *get_observadores()*, *get_observador(id)*, *get_programas()*, *get_programa(id)*, *evaluation()*, *falsification()* que utilizan *route()* para decirle a Flask qué URL debería activar la función en evaluación y que tipo de llamada REST utiliza.

```
def init_server(args):
    # Init flask stuff
    app = Flask(__name__)

    app.logger.addHandler(logging.StreamHandler())  # pylint: disable=E1101
    app.logger.setLevel(logging.DEBUG)  # pylint: disable=E1101

    logger = app.logger
    logging.basicConfig(format='%(name)s:%(levelname)s:%(asctime)s %(message)s',
                        datefmt='%H:%M:%S')

    @app.route("/")
    def main():…

    @app.route('/observator', methods=['GET'])
    def get_observadores():…

    @app.route('/observator/<id>', methods=['GET'])
    def get_observador(id):…

    @app.route('/programas', methods=['GET'])
    def get_programas():…

    @app.route('/programas/<id>', methods=['GET'])
    def get_programa(id):…


    @app.route('/evaluation', methods=['POST'])
    def evaluation():…

    @app.route('/falsification', methods=['POST'])
    def falsification():…

    return app
```

Figura 6.15: Método init_server(args) .

En el método *main()* de la Figura 6.16 se hace una llamada al método *render_template* donde se cargará nuestro archivo HTML de la página web.

```
@app.route("/")
def main():
    return render_template('main.html')
```

Figura 6.16: Método main() del init_server(args) .

En el método *get_observadores()* de la Figura 6.17 se recorre todos los archivos que hay en la ruta */observator* y se devuelve un json (haciendo uso de la librería jsonify) con el listado de los mismos.

```
@app.route('/observator', methods=['GET'])
def get_observadores():
    """
    Devuelve la lista de observators
    """
    paths = pathlib.Path(args.observators).glob('**/*')
    fileNames = [x.name for x in paths if x.is_file()]

    return jsonify(fileNames)
```

Figura 6.17: Método get_observadores() del init_server(args) .

El método *get_observador(id)* de la Figura 6.18 permite cargar un observador concreto indicando su nombre en el campo id. Para ello, se utiliza la llamada REST de tipo GET, para conocer qué observador ha sido seleccionado en el frontend. Una vez leído el archivo de ese observador se devolverá en un json.

```
@app.route('/observator/<id>', methods=['GET'])
def get_observador(id):
    """
    Devuelve el observator
    """
    with open(f"{args.observators}/{id}") as f:
        res = f.read()

    return jsonify(res)
```

Figura 6.18: Método get_observador(id) del init_server(args) .

En el método *get_programas* de la Figura 6.19 se listan todos los archivos que hay en la ruta */programas* devolviéndolos en un json.

```python
@app.route('/programas', methods=['GET'])
def get_programas():
    """
    Devuelve la lista de programas
    """
    paths = pathlib.Path(args.programs).glob('**/*')
    fileNames = [x.name for x in paths if x.is_file()]

    return jsonify(fileNames)
```

Figura 6.19: Método get_programas() del init_server(args) .

Por otro lado, al igual que se ha hecho con el método *get_observadores(id)* en el método *get_programas(id)* (Figura 6.20) se hace uso de una llamada REST de tipo GET para conocer el programa que ha sido seleccionado en el *frontend* y devolver un json del mismo.

```python
@app.route('/programas/<id>', methods=['GET'])
def get_programa(id):
    """
    Devuelve el programa
    """
    with open(f"{args.programs}/{id}") as f:
        res = f.read()

    return jsonify(res)
```

Figura 6.20: Método get_programas(id) del init_server(args) .

El método *evaluation()* (Figura 6.21) se encarga de evaluar el contenido escrito por el usuario en el 'Cuadro de texto' para ello se obtiene en la variable *origCode* el código escrito en el *frontend*. Una vez obtenido ese código, se almacena en formato smt2, creando el archivo 'modelo.smt2'. Utilizando Z3Py se inicializa el *Solver()* y se leerá el código almacenado para comprobar su satisfacibilidad y en caso de ser satisfacible se devuelve en un json con el modelo resultante. En caso de no ser satisfacible se devolverá un json con *unsat*. Además si se produce un error, se devolverá un json con el error producido.

```
@app.route('/evaluation', methods=['POST'])
def evaluation():
    body = request.json

    # Sent code from javascript
    origCode = body.get("code")
    print(origCode)
    f = open("modelo.smt2","w")
    f.write(origCode)
    f.close()
    s = Solver()
    try:
        s.from_file("modelo.smt2")
        if s.check() == sat:
            print(s.check());
            print(s.model())
            m = s.model()
            modelo = m.__repr__ ()
            code = evaluate(modelo)
        else:
            code = evaluate("unsat")
    except Exception as e:
        code = evaluate(str(e))
    return jsonify(code)
```

Figura 6.21: Método evaluation() del init_server(args) .

En el método *falsification()* (Figura 6.22) nos encontramos con que primero se recibe los valores cargados en el *frontend* y se crea un archivo smt2. Posteriormente se inicializan dos atributos, donde *dato* se encargará de leer el archivo smt2 creado anteriormente, mientras que *d* creará uno nuevo con permisos para escribir. A continuación, se recorrerá mediante un bucle for cada linea del documento que estamos leyendo, y si la linea empieza por la cadena '*(assert*' se formará una nueva cadena añadiendo '*(not(*' y un ')' de cierre, escribiéndose en *d* el nuevo axioma negado y saliendo del bucle.

Finalmente, utilizando Z3Py se inicializa el *Solver()* y se carga el archivo modificado creado anteriormente para comprobar su satisfacibilidad. En caso de ser satisfacible se devuelve en un json el modelo resultante. En caso de no ser satisfacible se devolverá un json con *unsat*. En caso de error, al igual que en el método *evaluation()* se devolverá un json con el error producido para que el usuario lo pueda corregir.

```
@app.route('/falsification', methods=['POST'])
def falsification():
    body = request.json

    # Sent code from javascript
    origCode = body.get("code")
    f = open("modelo.smt2","w")
    f.write(origCode)
    f.close()

    dato = open ("modelo.smt2", "r")
    d = open("res.smt2","w")
    for line in dato:
        if(line.startswith("(assert")):
            x = line[0:8]+ "(not(" + line[9:-1] + ")"
            d.write(x)
            break
        d.write(line)
    d.close()
    s = Solver()
    try:
        s.from_file("res.smt2")
        if s.check() == sat:
            print(s.check());
            print(s.model())
            m = s.model()
            modelo = m.__repr__ ()
            code = evaluate(modelo)
        else:
            code = evaluate("unsat")
    except Exception as e:
        code = evaluate(str(e))
    return jsonify(code)
```

Figura 6.22: Método falsification() del init_server(args) .

Frontend

El primer aspecto a desarrollar es la interfaz de usuario que, como se observa en la Figura 6.23, está compuesta por los siguientes elementos:

- ***ComboBox* Observadores**: Desplegable donde se listan todos los observadores disponibles para su utilización.
- ***ComboBox* Programas**: Al igual que el '*ComboBox* Observadores' consiste en un desplegable donde se listan todos los programas disponibles para su utilización.
- **Limpiar**: Botón cuya función es la de borrar todo el contenido que haya en el cuadro de texto y en el modelo resultante.
- **Cuadro de texto**: Elemento donde se puede escribir tanto la función como la declaración de tipos y restricciones a evaluar o falsificar. En caso de cargar un observador o programa se apreciaran los siguientes elementos:

- **Código**: En está sección se mostrará el código de la función elegida, en caso de ser un programa se incluye el código de las funciones que componen ese programa. Mientras que en caso de haber elegido un observador se mostrará únicamente el código de ese observador. Se aprecia un ejemplo haciendo uso del observador length en la Figura 6.23.
- **Ejemplo**: En está sección se muestra un ejemplo de un axioma haciendo uso del observador o programa elegido que se pretende analizar. Se observa un ejemplo haciendo uso del observador length en la Figura 6.23.
- **Estructura**: Está sección tiene como objetivo definir un modelo de la estructura del observador o programa elegido, para servir de guía al usuario a crear su propio axioma. Un ejemplo haciendo uso de la estructura del observador length se aprecia en la Figura 6.24.

- **Evaluar**: Este botón se encarga de evaluar el código definido en el cuadro de texto.
- **Falsificar**: Este botón se encarga de falsificar el código definido en el cuadro de texto. A deferencia de 'Evaluar', se niega el axioma para comprobar si al negar el mismo axioma que se ha escrito, se demuestra la validez del axioma negado.

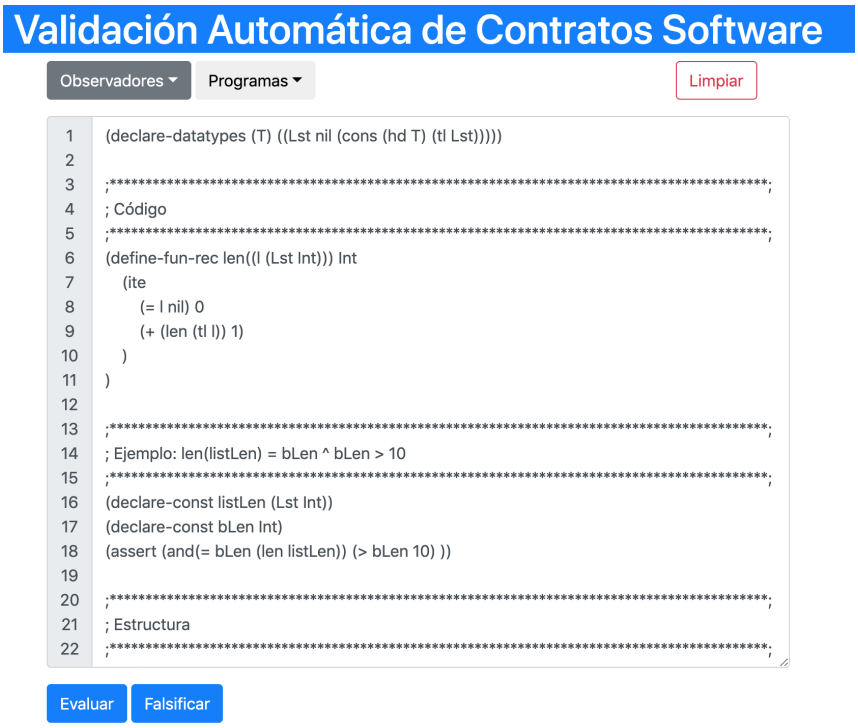

Figura 6.23: Interfaz página web .

```
20 ;*************************************************************************************;
21 Estructura
22 ;*************************************************************************************
23 (len listLen)
```

Figura 6.24: Estructura observador / programa .

6.4.4. Despliegue

Un aspecto importante en nuestra página web es la realización del despliegue para que los usuarios puedan utilizarla de forma adecuada. Para ello se ha optado por la utilización de Google Cloud Platform (GCP). En esta sección se expondrá las distintas características esenciales que ofrece GCP y cómo se ha llevado a cabo el despliegue del proyecto.

Google Cloud Platform (GCP) consta de un conjunto de recursos físicos, como ordenadores, unidades de disco duro y máquinas virtuales (VM), que se encuentran en los centros de datos de Google a lo largo de todo el mundo.

Cada centro de datos está ubicado en una región, existiendo regiones disponibles en Asia, Australia, Europa, América del Norte y América del Sur. Cada región es un conjunto de zonas aisladas entre sí dentro de cada región. Las zonas se identifican mediante nombres que combinan una letra identificadora con el nombre de la región [12].

La utilización de esta distribución de recursos proporciona varios beneficios, como es la redundancia en caso de fallos y una menor latencia, gracias a la proximidad de los recursos respecto a los clientes.

Proyectos

Los recursos de GCP que se asignan y utilizan deben pertenecer a un proyecto, que puede considerarse como la entidad organizativa de la compilación. Para describir con detalle las aplicaciones, un proyecto consta de configuración, permisos y metadatos para ese fin. Los recursos de un proyecto pueden operar en conjunto pero un proyecto no puede acceder a los recursos de otro proyecto.

Formas de interactuar con los servicios

Las formas de interactuar con los servicios que ofrece Google Cloud Platform (GCP) son:

- **Google Cloud Console:** Proporciona una interfaz gráfica de usuario basada en la Web que puedes usar para administrar tus proyectos y recursos de GCP.
- **Interfaz de línea de comandos:** La herramienta de gcloud se puede usar para administrar tu flujo de trabajo de desarrollo y tus recursos de GCP.
- **Bibliotecas cliente:** El SDK de Cloud contiene bibliotecas cliente que te permiten crear y administrar recursos con facilidad.

[12] Google Cloud: https://cloud.google.com/docs/overview?hl=es-419

Productos Cloud destacados

Existe una enorme cantidad de microservicios que conforman el Google Cloud Platform (GCP), pero los más destacados son los siguientes:

- ***Compute Engine:*** Te permite crear y ejecutar máquinas virtuales en la infraestructura de Google, ofreciendo opciones de escalada en caso de ser necesario. Además, posibilita la opción de iniciar con facilidad grandes clústeres de procesamiento en la infraestructura de Google.
- ***Cloud Storage:*** Permite almacenar y recuperar cualquier cantidad de datos en todo el mundo y en cualquier momento, siendo útil para una enorme variedad de situaciones, como la entrega de contenido de sitios web, el almacenamiento de datos de archivos y recuperación ante desastres.
- ***Cloud Run:*** Es una plataforma de procesamiento administrada que permite ejecutar contenedores sin estado que se pueden invocar a través de solicitudes web o eventos de Pub/Sub. Se trata de una plataforma sin servidores, lo que significa que quita la complejidad de la administración de infraestructura, de modo que el usuario simplemente tiene que centrase en compilar sus soluciones.
- ***Cloud SQL:*** Es un servicio de base de datos completamente administrado que facilita la configuración, el mantenimiento, la administración y la gestión de las bases de datos relacionales (como MySQL, PostgreSQL o SQL Server) en Google Cloud Platform.
- ***Vision AI:*** Permite integrar con facilidad características de detección de visión a tus aplicaciones, como etiquetado de imágenes, detección de puntos de referencia y rostros, reconocimiento óptico de caracteres (OCR), ubicación de objetos y etiquetado de contenido explícito.
- ***AutoML:*** Permite aprovechar las capacidades del aprendizaje automático de Google a fin de crear tus propios modelos de aprendizaje automático personalizados según las necesidades de tu negocio y, luego, integrarlos en tus aplicaciones y sitios web.
- ***Dialogflow:*** Es una plataforma con comprensión del lenguaje natural que facilita el diseño de una interfaz de usuario de conversación y su integración a tu aplicación para dispositivos móviles, aplicaciones web, dispositivos, bots, sistemas de respuesta de voz interactiva y más.

Integración de la página web

Para la integración de nuestra página, se ha optado por la utilización de *Compute Engine*, por las ventajas que ofrece anteriormente expuestas.

Primero que todo, es necesario crear un nuevo proyecto cuyo titulo elegido ha sido 'tfmZ3' para empezar a realizar el despliegue de la página web. Para ello, como se aprecia en la Figura 6.25, simplemente tendremos que añadir el

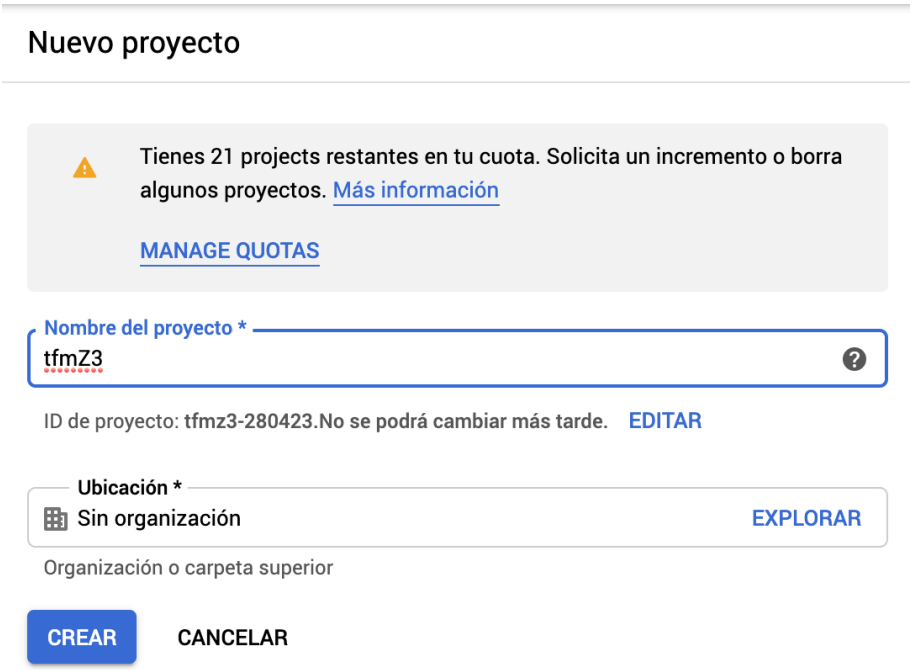

Figura 6.25: Crear nuevo proyecto .

nombre del proyecto que deseemos crear y si pertenece a alguna organización y finalmente pulsar en el botón crear.

Una vez el proyecto ha sido creado deberemos seleccionarlo, para ello se ha de pulsar en el desplegable ubicado al lado del nombre de Google Cloud Platform que indica el proyecto actual en el que estamos ubicados (Figura 6.26). Al pulsar aparecerá una ventana emergente listando todos los proyectos que hemos creado (Figura 6.27).

Figura 6.26: Barra navegación .

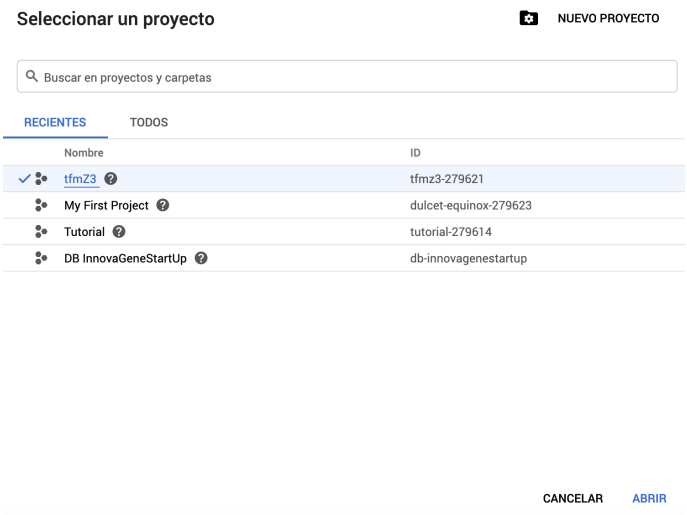

Figura 6.27: Seleccionar proyecto .

Una vez el proyecto ya ha sido creado y seleccionado con éxito, nos dirigimos a *ComputeEngine* en especial al apartado 'instancias de VM' (VM significa *VirtualMachine*) (Figura 6.28) donde, en la barra superior (Figura 6.29) se aprecian las distintas características de las que disponemos, como crear una instancia VM, importar VM, o las opciones para encender, apagar o pausar las VM.

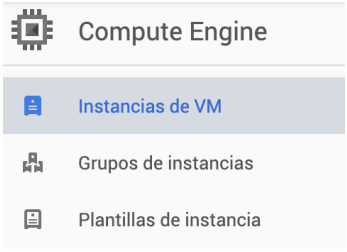

Figura 6.28: Instancias de VM .

Figura 6.29: Opciones instancias de VM .

Para crear la máquina virtual necesaria para el despliegue se hará click en el botón de 'Crear instancia', donde como se observa en la Figura 6.30 por un lado, en la parte izquierda tendremos las opciones generales disponibles, que son:

- **Nueva instancia de VM:** Nos permite crear una nueva instancia de VM desde cero, donde podremos seleccionar con todo detalle el nombre, la región donde estará ubicado, la zona de esa región y la configuración de la máquina. En la Configuración de la máquina, puede personalizarse el número de núcleos de la CPU o de GPUs para ajustarse a las necesidades de nuestro proyecto. En caso de optar por un tipo de máquina más reducido podremos escalarlo fácilmente a una opción de mayor capacidad. Además de estás características, también podemos configurar el disco de arranque, pudiendo seleccionar entre un enorme número de sistemas operativos, como Debian, Ubuntu, CentOs,Windows Server,etc o el tamaño del disco que queremos utilizar. Finalmente, hay una opción de *firewall* que nos permite decidir si queremos habilitar el trafico HTTP y HTTPS entre otras características de seguridad.

- **Nueva instancia de VM a partir de una plantilla:** Definen el tipo de máquina, la imagen del disco de arranque o del contenedor, las etiquetas y otras propiedades de la instancia. También puede usarse una plantilla de instancias a fin de crear un MIG o VM individuales. Las plantillas de instancias son una forma conveniente de guardar la configuración de una instancia de VM para que puedas usarla más adelante a fin de crear VM o grupos de VM.

- **Instancia nueva de VM a partir de una imagen de máquina:** Es un recurso que almacena toda la configuración, los metadatos, los permisos y los datos de uno o más discos. necesarios para crear una instancia de máquina virtual (VM).

- ***Marketplace:*** Permite implementar una solución ya lista para usar, sin necesidad de instalación posterior ni configuración adicional.

En nuestro caso, se ha creado una nueva instancia de una máquina en la región europe-west-1 zona b, con un tipo de máquina g1-small que es una máquina con 1 CPU y 4GB de memoria RAM, en lo referido al disco duro se ha optado por instalar el sistema operativo Debian con una capacidad de 10gb.

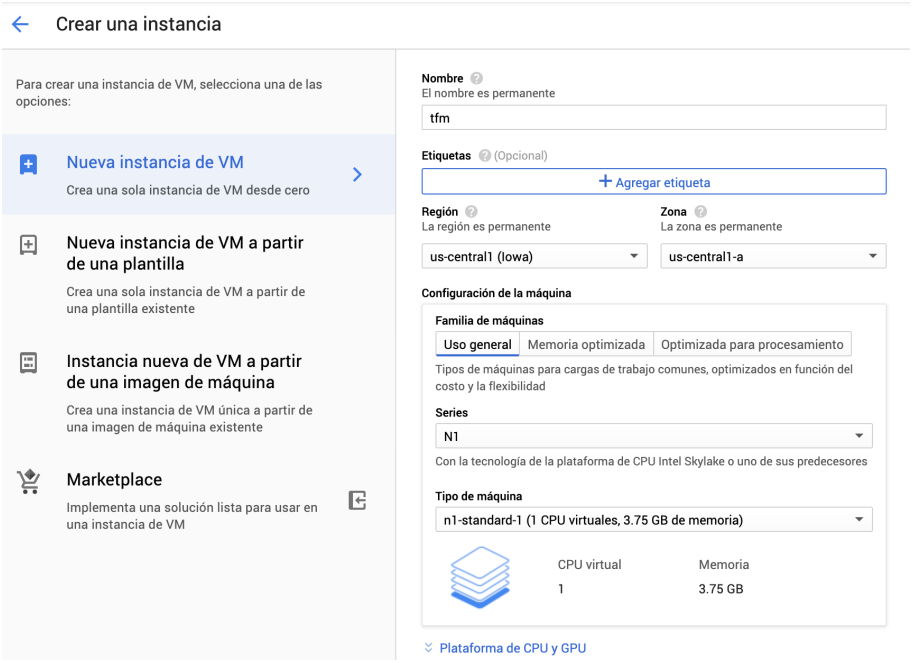

Figura 6.30: Crear instancias de VM .

Al finalizar la creación de la VM es necesario realizar una configuración de la red, y definir qué puertos estarán abiertos para que la aplicación se pueda ejecutar correctamente y no sea bloqueada por el firewall. Para ello será necesario acceder al menú Red de VPC y en el apartado firewall, donde se nos mostrará un menú muy similar al de Instancias de VM, pulsaremos en el botón Crear regla firewall (Figura 6.31).

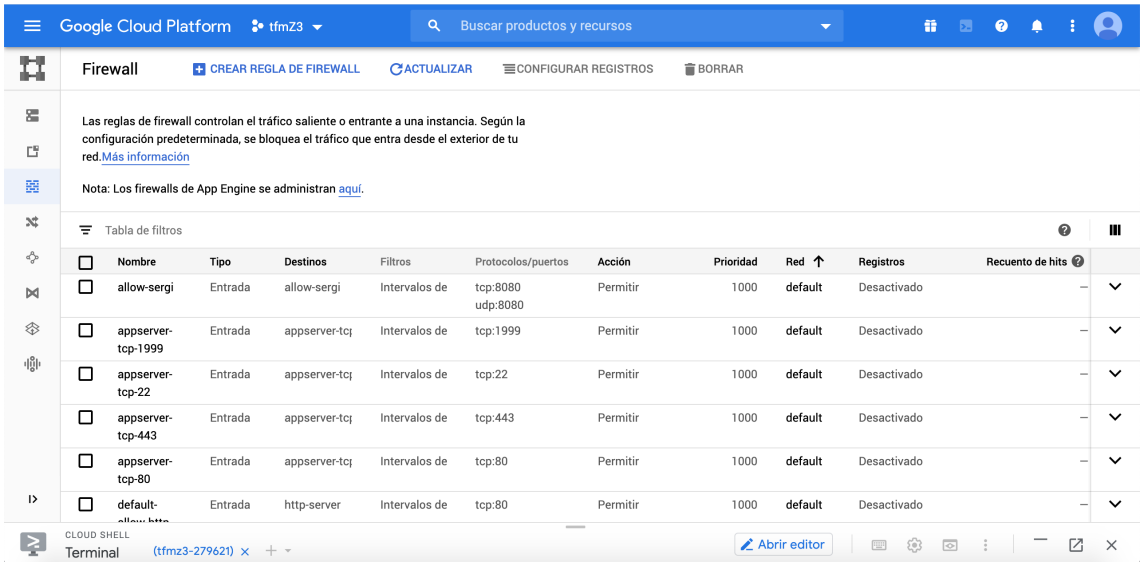

Figura 6.31: Red de VPC .

Una vez pulsado en el botón nos aparecerá un formulario con un gran número de elementos a configurar para poder definir correctamente las reglas de firewall. En nuestro caso, como se observa en la Figura 6.32, se ha especificado el puerto 8080 tanto de tcp como de udp que se encuentra en el apartado Protocolos y Puertos. Por otro lado, se ha añadido un filtro de intervalos de IP, en la dirección 0.0.0.0/0 .

Nombre	Tipo	Destinos	Filtros	Protocolos/puertos	Acción	Prioridad	Red	Registros
allow-sergi	Entrada	allow-sergi	Intervalos de IP. 0.0.0.0/0	tcp:8080 udp:8080	Permitir	1000	default	Desactivado

Figura 6.32: Configuración regla de firewall .

Una vez creada la instancia a VM, volvemos al menú de 'Instancias de VM' anterior, donde ahora nos aparecerá la máquina virtual que hemos creado, su estado (en este caso encendida) y algunas características esenciales, como la IP externa, la IP interna, o la posibilidad de conectarse por SSH (Figura 6.33).

Figura 6.33: Creación de instancia completada .

Para configurar correctamente la máquina virtual y lanzar la página web, es recomendable utilizar SSH para conectarse. Una vez dentro de la máquina (Figura 6.34) nos aparecerá un terminal donde poder instalar todos los paquetes necesarios para el correcto despliegue, los cuales son los siguientes:

- *sudo apt install python3*
- *sudo apt install unzip*
- *pip3 install z3-prover*
- *pip3 install Flask*
- *pip3 install jsonlib*

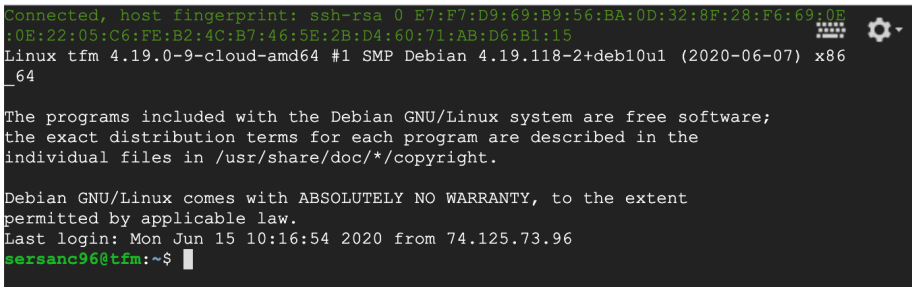

Figura 6.34: Ejemplo terminal .

Una vez instalados estos paquetes, lanzaremos la aplicación utilizando el comando *nohup python3 server.py*. Al utilizar *nohup*, el proceso se quedará en segundo plano, de manera que la aplicación se mantendrá en ejecución aunque cerremos el terminal. Para acceder a la página web simplemente habrá que conectarse a la página web que esta visible en `http://104.155.40.246:8080/`.

CAPÍTULO 7
Pruebas

A lo largo de esta capítulo se realizarán las pruebas pertinentes para la comprobación del correcto funcionamiento de la solución implementada.

7.1 Introducción

Las pruebas software se definen como la verificación dinámica del comportamiento de un programa en relación a su comportamiento esperado. Para ello, se utiliza un conjunto finito de casos de prueba, seleccionados de manera adecuada, siendo un factor critico para determinar la calidad del software [29].

Es muy importante realizar pruebas en el desarrollo del software ya que en cualquier etapa del ciclo de vida es posible que aparezcan errores, los cuales pueden permanecer sin ser descubiertos [36]. Los objetivos principales de la realización de las pruebas son:

- Detectar un error especifico en el código.
- Descubrir errores no descubiertos anteriormente.
- Determinar un caso de prueba adecuado.

Un caso de prueba está compuesto por un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular.

Existen diversos tipos de pruebas, pudiéndose clasificar como:

- **Pruebas manuales:** Son un tipo de prueba que se realizan por el usuario paso a paso.
- **Pruebas automáticas:** Este tipo de pruebas, consiste en la utilización de un software alternativo al utilizado para comprobar si los resultados obtenidos concuerdan con los esperados. La ventaja que presenta respecto a las pruebas manuales es que permite mecanizar pruebas que podrían ser tediosas o de una mayor dificultad.

7.2 Pruebas de los Observadores

Para probar el correcto funcionamiento de los observadores definidos en el capítulo 6, se ha precedido a definir restricciones que cumplan el caso base y caso general de cada observador de la forma siguiente:

Length

En la Figura 7.1 se observan los asertos necesarios para probar el caso base (primer aserto) y el caso general (segundo aserto).

```
(assert (and (= nil list) (= x (len list))))
(assert  (= 10 (len list)))
```

Figura 7.1: Testing observador Length .

El resultado obtenido al aplicar el primer aserto corresponde al caso base la Figura 7.2, donde al restringir que la lista tenga de estar vacía el valor que nos devuelve la función es 0, que se almacena en la variable x.

```
(define-fun x () Int
  0)
(define-fun list () (Lst Int)
  nil)
```

Figura 7.2: Testing observador Length caso base .

El asserto correspondiente al caso general (el segundo) obtiene como resultado un modelo que cumple con la restricción, como se observa en la Figura 7.3 donde se ha generado una lista con 10 elementos.

```
[list = cons(29,
             cons(30,
                  cons(27,
                       cons(32,
                            cons(25,
                                 cons(31,
                                      cons(24,
                                        cons(28,
                                        cons(22,
                                        cons(23, nil))))))))),
 len = [nil -> 0, else -> 1 + len(tl(Var(0)))]]
```

Figura 7.3: Testing observador Length caso general .

Esto proporciona garantías razonables sobre el correcto funcionamiento de la función observador Length.

isMember

En la Figura 7.4 se observan los asertos necesarios para probar el caso base (primer aserto), que se cumple cuando la lista esta vacía, y el caso general (segundo aserto), que se cumple cuando la lista tiene elementos devolviendo *true* en caso de que el elemento buscado forme parte de la lista y $false$ en caso contrario.

```
(assert (and (= nil list) (= x (isMember 1 list))))
(assert (= true (isMember 1 list)))
```

Figura 7.4: Testing observador isMember .

Al utilizar el primer aserto, obtenemos el código de la Figura 7.5, donde se aprecia que, al estar la lista vacía, se devolverá $false$ (que es el valor almacenado en la variable booleana x) cumpliendo con la condición del caso base.

```
(define-fun x () Bool
  false)
(define-fun list () (Lst Int)
  nil)
```

Figura 7.5: Testing observador isMember caso base .

Por otro lado, cuando utilizamos el segundo aserto como se observa en la Figura 7.4, al restringir que el elemento 1 tiene que formar parte de la lista, estamos utilizando el caso general, donde se añade a la lista el elemento 1 como se aprecia en la Figura 7.6.

```
(define-fun list () (Lst Int)
  (cons 1 nil))
```

Figura 7.6: Testing observador isMember caso general.

isNull

En la Figura 7.7 se observan los asertos necesarios para probar que este predicado se cumple cuando la lista esta vacía (devolviendo *true*), mientras que y el segundo aserto se cumple cuando la lista tiene elementos (devolviendo $false$).

```
(assert (and (= nil list) (= x (isNull list))))
(assert (= false (isNull list)))
```

Figura 7.7: Testing observador isNull .

Al utilizar el primer aserto se esta limitando el tamaño de la lista a nil de manera que, al comprobar si la lista está vacía, el valor de x será *true* cumpliendo la restricción (Figura 7.8).

```
(define-fun x () Bool
  true)
(define-fun list () (Lst Int)
  nil)
```

Figura 7.8: Testing observador isNull caso base.

Por otro lado, el segundo aserto comprueba el caso en el que la lista no este vacía, añadiendo un elemento para que sea cierto (en este caso el 0 como muestra la Figura 7.9).

```
(define-fun list () (Lst Int)
  (cons 0 nil))
```

Figura 7.9: Testing observador isNull caso lista vacía.

isEmpty

Para comprobar si este predicado se cumple, ha sido necesario definir los asertos de la Figura 7.10, donde en caso de que la longitud de la lista sea 0, se devolverá *true*, mientras que en caso contrario se devolverá $false$.

```
(assert (and (= 0 (len list)) (= x (isEmpty list))))
(assert (= false (isEmpty list)))
```

Figura 7.10: Testing observador isEmpty.

Por tanto, al utilizar el primer aserto, se obtendrá un modelo resultante donde, al haber restringido el tamaño de la lista a 0, el valor de x será *true* como se aprecia en la Figura 7.11.

```
(define-fun x () Bool
  true)
(define-fun list () (Lst Int)
  nil)
```

Figura 7.11: Testing observador isEmpty caso true.

Por otro lado, al probar el segundo aserto, obtenemos un modelo donde la lista no es vacía y se la ha insertado el valor 0 (Figura 7.12).

```
(define-fun list () (Lst Int)
  (cons 0 nil))
```

Figura 7.12: Testing observador isEmpty caso false.

isFull

La función isFull consiste en una llamada a la función len para comprobar si la lista tiene un número de elementos determinado por la capacidad máxima, de manera que cuando se alcance dicho numero de elementos, *isFull* devolverá *true*, mientras que en caso contrario devolverá $false$. En la Figura 7.13 se ejemplifica una aserción para comprobar el caso en que se cumple que *isFull* es *true* y otra aserción para comprobar cuando se cumple que *isFull* es $false$.

```
(assert (= false (isFull list)))
(assert (= true (isFull list)))
```

Figura 7.13: Testing observador isFull.

De esta manera al evaluar la primera restricción, obtenemos un modelo donde efectivamente no se cumple que la lista ha llegado al tamaño limite definido (Figura 7.14).

```
(define-fun list () (Lst Int)
  (cons 5 (cons 6 nil)))
```

Figura 7.14: Testing observador isFull caso falso.

Por contraposición, al evaluar la segunda restricción, se obtiene un modelo donde se cumple que la lista haya llegado al limite del tamaño definido de 10 elementos (Figura 7.15).

```
[list = cons(25,
             cons(26,
                  cons(32,
                       cons(30,
                            cons(31,
                                 cons(23,
                                      cons(28,
                                        cons(24,
                                        cons(29,
                                        cons(27,
                                        cons(33, nil))))))))))),
 len = [nil -> 0, else -> 1 + len(tl(Var(0)))]]
```

Figura 7.15: Testing observador isFull caso cierto.

sortedList

Para comprobar que este predicado se cumple, en la Figura 7.16 se observan los asertos necesarios para probar los casos base (primer y segundo aserto) cuyo resultado deberá ser *true* ya que la lista al tener solo un elemento o estar vacía, se encuentra ordenada. El caso general, se cumple cuando la lista de elementos se encuentra ordenada ascendentemente y existe más de un elemento en la lista devolviendo *true* en caso contrario devolverá *false*.

```
(assert (and (= list nil) (= b (sortedList list  )) ))
(assert (and (= copy (insert 1 list)) (= b (sortedList copy  )) ))
(assert (and (= copy (insert 11 (insert 10 list))) (= b (sortedList copy  )) ))
```

Figura 7.16: Testing observador sortedList.

De manera, que al utilizar tanto el primer aserto como el segundo, obtenemos como resultado *true* (Figura 7.17) cumpliendo con la condición del caso base.

```
sat
(model
  (define-fun b () Bool
    true)
```

Figura 7.17: Testing observador sortedList caso base.

Por otro lado, el tercer aserto se comprueba el caso general, donde al añadir más de un elemento a la lista (en este caso el 11 y 10) se comprueba si se encuentra ordenada ascendentemente, devolviendo *false* (que es el valor que Z3 asigna a la variable *b* al evaluar la lista) debido a que no está ordenada, como se aprecia en la Figura 7.18.

```
[list = nil, b = False, copy = cons(11, cons(10, nil))]
```

Figura 7.18: Testing observador sortedList caso general.

7.3 Prueba Programas

insert

La función *insert* se encarga de añadir un elemento a una lista cuando se cumplen una serie de condiciones que no son otras que:

- La lista no exceda del tamaño máximo permitido.
- La lista no contenga ya el elemento que se pretende insertar, en cualquier otro caso se insertará el valor deseado a la lista

En la Figura 7.19, se aprecia una aserción donde se cumple el caso de que se exceda el tamaño máximo de la lista, una donde se cumple que el elemento ya forma parte de la lista y otra donde no se cumple ninguna de las anteriores.

```
(assert (and(= list (insert 1 list)) (= list (insert 1 list)) ) )
(assert (and(= true (isFull list)) (= list (insert 10 list))))
(assert (and (= 0 (len list)) (= list_copy (insert 1 list)) ))
```

Figura 7.19: Testing observador insert.

Al evaluar la primera aserción obtenemos como resultado un modelo (Figura 7.20) donde se observa que se ha insertado solo una vez el elemento 1, cumpliendo con la restricción de la función de que si un elemento ya forma parte de la lista no se vuelva a insertar.

```
[list = cons(0, cons(1, nil)),
 isMember = [else ->
             If(Var(1) == nil,
                False,
                Or(Var(0) == hd(Var(1)),
                   isMember(Var(0), tl(Var(1))))))],
 len = [nil -> 0, else -> 1 + len(tl(Var(0)))]]
```

Figura 7.20: Testing observador insert primera aserción.

En caso de evaluar la segunda aserción, se obtiene un modelo resultante que se aprecia en la Figura 7.21, donde la lista está llena y por tanto no se añade el valor 10 a la lista.

```
[list = cons(25,
             cons(26,
                  cons(32,
                       cons(30,
                            cons(31,
                                 cons(23,
                                      cons(28,
                                        cons(24,
                                        cons(29,
                                        cons(27,
                                        cons(33, nil)))))))))),
 isMember = [else ->
             If(Var(1) == nil,
                False,
                Or(Var(0) == hd(Var(1)),
                   isMember(Var(0), tl(Var(1))))))],
 len = [nil -> 0, else -> 1 + len(tl(Var(0)))]]
```

Figura 7.21: Testing observador insert segunda aserción.

Finalmente al evaluar el tercer aserto, se obtiene un modelo (Figura 7.22) donde, al haber restringido que la lista este vacía e insertar el valor 1 en la *list_copy*, se aprecia como el valor se ha insertado correctamente, cumpliendo con lo definido en la función ya que no excede la capacidad de elementos de la lista, ni tiene el mismo elemento ya perteneciente en la lista.

```
[list = nil,
 list_copy = cons(1, nil),
 isMember = [else ->
             If(Var(1) == nil,
                False,
                Or(Var(0) == hd(Var(1)),
                   isMember(Var(0), tl(Var(1))))))],
 len = [nil -> 0, else -> 1 + len(tl(Var(0)))]]
```

Figura 7.22: Testing observador insert tercera aserción.

7.4 Limitaciones

Para la prueba de los observadores, un aspecto que hubiera sido interesante abordar sería la utilización de herramientas de análisis estático, como pudiera ser el caso de LLBMC, que es una herramienta de análisis estático para encontrar errores en programas C (y, en cierta medida, en C ++). En caso de que hubiera sido compatible con Python, se hubiera podido testear los observadores definidos haciendo uso de la librería Z3Py. Una de las ventajas que ofrecen las herramientas de análisis estático son:

- Reducir el tiempo y el esfuerzo necesarios para las pruebas de software.
- Mejorar la calidad del software.
- Lograr altas tasas de cobertura de prueba.
- Obtener software estable y seguro en un tiempo reducido.

Por contra, las desventajas que presenta y que dificultan su utilización son, entre otras:

- La dificultad en la comprensión de los mensajes de error.
- El exceso de falsos positivos.
- La dificultad en configurar un analizador para ocultar falsos positivos.

7.5 Pruebas Unitarias

El objetivo de una prueba unitaria es comprobar el correcto funcionamiento de una unidad de código, verificando que el código cumple con la funcionalidad esperada. Por ello, se realiza la verificación para comprobar que sea correcto el nombre, los tipos de los parámetros, el tipo que se devuelve, si el estado inicial es válido y si el estado final también lo es.

Un ejemplo de la utilización de este tipo de pruebas sería que en el diseño estructurado o funcional se realizará la prueba a una función o un procedimiento, mientras que si fuera necesario hacer una prueba a un diseño orientado a objetos se realizaría una prueba a una clase [12].

7.5.1. Pytest

Pytest es un *framework* fácil de utilizar cuyo objetivo es facilitar la escritura de pequeñas pruebas pero pudiéndose escalar para admitir pruebas funcionales complejas en aplicaciones y bibliotecas. Se ha utilizado este marco para testear los métodos más importantes de *server.py* y así comprobar que el *framework* Flask funciona correctamente. Para ello se ha creado un nuevo archivo *test_server.py*.

Para certificar el correcto funcionamiento de los métodos desarrollados, se comprueba el código de respuesta que proporciona la página web. Dado que el código de respuesta de estado satisfactorio HTTP 200 OK indica que la solicitud ha tenido éxito, se pretende comprobar que todos los métodos obtienen esa respuesta para verificar su satisfacibilidad, como se aprecia en la Figura 7.23.

```python
def test_main():
    response = app.test_client().get('/')
    assert response.status_code == 200

def test_getobservador():
    response = app.test_client().get( '/observator',data=json.dumps({}),content_type='application/json',)
    data = json.loads(response.get_data(as_text=True))
    assert response.status_code == 200

def test_getprogramas():
    response = app.test_client().get('/programas',data=json.dumps({}),content_type='application/json',)
    data = json.loads(response.get_data(as_text=True))
    assert response.status_code == 200

def test_getevaluation():
    response = app.test_client().post(
        '/evaluation',
        data=json.dumps({'(assert (= 1 (len list)))':'listLen = cons(2, nil)'}),
        content_type='application/json',
    )
    data = json.loads(response.get_data(as_text=True))
    assert response.status_code == 200

def test_getfalsification():
    response = app.test_client().post(
        '/falsification',
        data=json.dumps({'(assert (not(and (= true (isNull listNull)) (= false (isNull listNull)) )))':'[]'}),
        content_type='application/json',
    )
    data = json.loads(response.get_data(as_text=True))
    assert response.status_code == 200
```

Figura 7.23: Testing métodos *server.py*.

Obtenemos un resultado satisfactorio al ejecutar las pruebas como se aprecia en la Figura 7.24.

```
============================= test session starts ==============================
platform darwin -- Python 3.7.4, pytest-5.2.1, py-1.8.0, pluggy-0.13.0
rootdir: /Users/sergisanz/Documents/MITSS/TFM/Pagina Web
plugins: arraydiff-0.3, remotedata-0.3.2, doctestplus-0.4.0, openfiles-0.4.0
collected 5 items

test_server.py .....                                                     [100%]

============================== 5 passed in 0.75s ===============================
```

Figura 7.24: Resultados pruebas métodos *server.py*.

7.6 Pruebas de sistema

En esta sección se va a proceder a probar algunos axiomas candidatos para demostrar que el sistema se comporta de forma correcta frente al resultado esperado. Para ello, se ha procedido primero a probar el axioma candidato definido para la explicación realizada en la sección 6.3.

Como se observa en la Figura 7.25 por un lado nos encontramos con que Z3 ha generado el valor 3 para la variable x, mientras que la variable simbólica n ha adquirido el valor 0, por lo que el tamaño máximo que puede tener una lista será de 1. Por otro lado, la variable simbólica ?b se evalúa a True, eso significa que el valor de x (el 3) estará contenido en la lista.

```
[x = 3, listInsert = cons(3, nil), copy = cons(3, nil), n = 0, b = True, isMember = [else -> If(Var(1) ==
nil, False, Or(Var(0) == hd(Var(1)), isMember(Var(0), tl(Var(1))))))], len = [nil -> 0, else -> 1 +
len(tl(Var(0))))]]
```

Figura 7.25: Resultado evaluación axioma candidato.

Como vimos en la Figura 6.9 en el segundo *assert* se añade x a la lista *copy*, el motivo por el que en la Figura 7.25 no se inserta el elemento 3 dos veces, es por que si nos fijamos en la sección 6.2, el *insert* comprueba si un elemento esta ya contenido en una lista para evitar repetidos, por eso motivo no se vuelve a insertar. Esto proporciona garantías razonables sobre el correcto funcionamiento de la verificación del axioma candidato.

Con el fin de ayudar a entender mejor como funciona la evaluación de los axiomas, en la Figura 7.26 se ha modificado el programa *insert*, prescindiendo de las restricciones definidas anteriormente, de manera que ahora se limitará a insertar el elemento deseado sin realizar ninguna comprobación.

```
(define-fun insert ((i Int) (l (Lst Int))) (Lst Int)
       (cons i l)
)
```

Figura 7.26: Programa insert modificado.

Al volver a evaluar las restricciones definidas en la sección 6.2 haciendo uso del nuevo modificador, se devolverá como resultado *unsat* ya que, indistintivamente de si ?b se evalúa a *true* o $false$ al quitar la restricción del programa modificador se garantiza que el estado posterior a la ejecución del modificador tendrá un elemento más que en el antecedente, por tanto se incumple la restricción de tamaño de $?n + 1$.

El hecho de que se haya devuelto *unsat* como resultado indica que no hay ningún modelo que cumpla las restricciones definidas, así que lo que se infiere es que su negación es cierta. Por tanto, debemos negar el axioma (o podemos pulsar el botón de 'falsificar' que se encargará de negar ese axioma y volverlo a evaluar) para obtener un contraejemplo que falsifica ese axioma, como se aprecia en la Figura 7.27.

```
[copy = nil, x = 1, n = -2, listInsert = nil, isMember = [else -> If(Var(1) == nil, False, Or(Var(0) ==
hd(Var(1)), isMember(Var(0), tl(Var(1))))))], len = [nil -> 0, else -> 1 + len(tl(Var(0)))]]
```

Figura 7.27: Modelo de contraejemplo.

Finalmente, para comprobar que todos los elementos de la página web funcionan de forma adecuada sin que existe enlaces caídos de algún tipo, se ha utilizado la herramienta *World Wide Web Consortium* (W3C)[1] de forma que como se puede visualizar en la Figura 7.28, todos los enlaces funcionan correctamente en la página web.

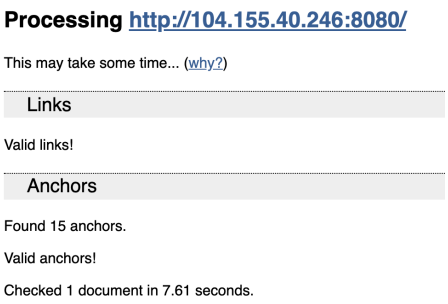

Figura 7.28: Resultado prueba comprobación enlaces.

7.7 Pruebas de escalabilidad

Este tipo de pruebas no funcionales permiten determinar el grado de escalabilidad que tiene un sistema. Se entiende como escalable la capacidad de una aplicación para que sin realizar cambios drásticos en su configuración, pueda soportar un incremento de la demanda en la operación.

En el caso de nuestro sistema la escalabilidad puede ser necesaria debido a un incremento del número de observadores o de programas . Para ello, será necesario como se aprecia en la Figura 7.29 crear una imagen de la máquina. Esto consiste en crear una instancia del contenido actual de la máquina.

[1] W3C: https://validator.w3.org/checklink/

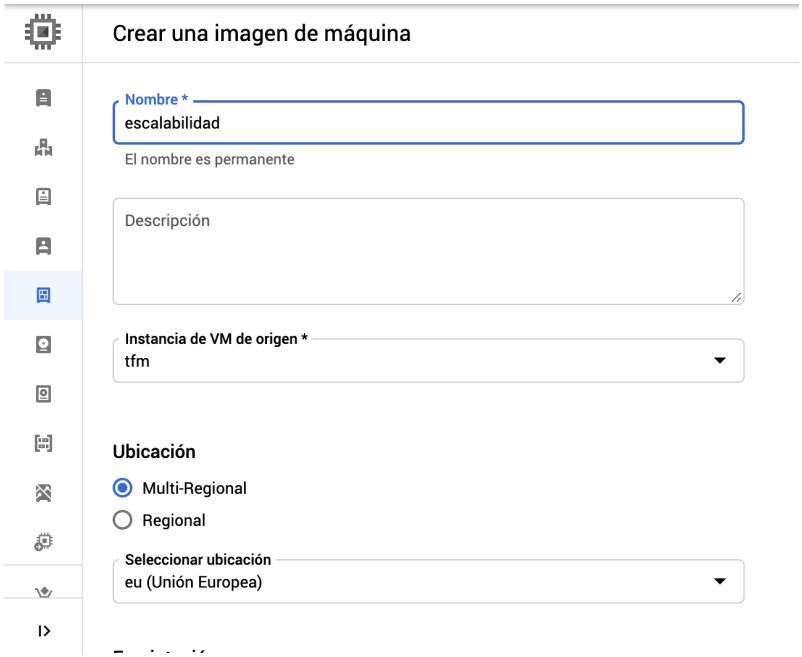

Figura 7.29: Crear imagen máquina.

Una vez creada la imagen de la máquina podemos crear una nueva máquina virtual a partir de la imagen de la máquina creada anteriormente, como se observa en la Figura 7.30 solo tendremos que seleccionar la imagen que pretendemos utilizar.

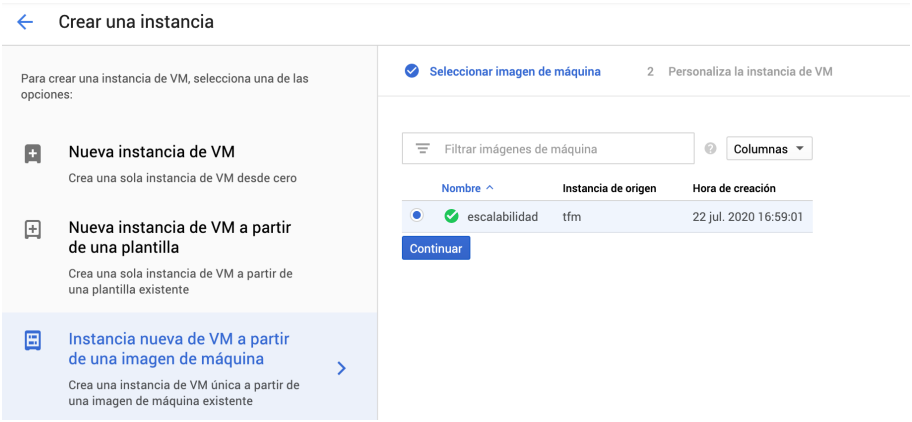

Figura 7.30: Crear instancia de la máquina.

A continuación configuraremos la nueva máquina, seleccionando un tipo de máquina de mayor potencia que la utilizada actualmente (Figura 7.31).

Figura 7.31: Personalizar la instancia de la VM.

7.8 Pruebas de carga

Un aspecto muy importante a la hora de probar una página web, es tener la certeza que la página puede aguantar un número de usuarios determinado en un momento. Lo que sirve para validar y verificar la calidad del sistema, con algunos atributos como escalabilidad, fiabilidad y uso de los recursos.

Para ello se ha utilizado la herramienta StresStimulus[2] que determina el rendimiento web y la escalabilidad de su aplicación bajo los rigores de la carga de tráfico pesado. Para ello, se emulan usuarios físicos de manera realista a través de generadores de carga locales o en un entorno de prueba en la nube.

En la Figura 7.32 se representa tanto de forma gráfica como se comportan los distintos elementos que interactúan con la página web además de incluir una tabla con los resultados obtenidos al ejecutar la herramienta StresStrimulus.

Donde se va incrementando el número de usuarios, hasta un número máximo de 50 usuarios por segundo que realizan un total de 158.989 peticiones por segundo, con un tiempo de respuesta de solo 0,062 segundos. Un buen tiempo aunque sea significativamente superior que si solo se tiene un usuario, donde la página proporciona un tiempo de respuesta de 0,035s.

[2]StresStrimulus: https://www.stresstimulus.com/

7.9 Pruebas de accesibilidad

La accesibilidad es un tipo de pruebas pensado en los usuarios que tienen alguna discapacidad. Siendo el objetivo de este tipo de pruebas descubrir la facilidad con la que se puede utilizar un sitio web y en base a esta información mejorar futuros diseños e implementaciones.

Una de las herramientas mas utilizadas para evaluar la accesibilidad es el validador del *World Wide Web Consortium* (W3C)[3], el cual es un organismo que desarrolla estándares de accesibilidad web. Para su utilización simplemente hay que introducir la URL del sitio web y obtendrás un listado detallado con los posibles errores HTML. En nuestro caso, como se observa en la Figura 7.33 no existen errores, solo una advertencia recomendando que se utilice una cabecera de tipo H2 - H6.

Otra herramienta utilizada para comprobar la accesibilidad es *Check My Colours*[4]. Es una herramienta para verificar las combinaciones de color entre el primer plano y el fondo de todos los elementos DOM y determinar si proporcionan suficiente contraste para personas con problemas de visión. Como se aprecia en la Figura 7.34 se ha pasado las pruebas de forma satisfactoria.

[3]W3C: https://validator.w3.org/
[4]Check My Colours: https://www.checkmycolours.com/

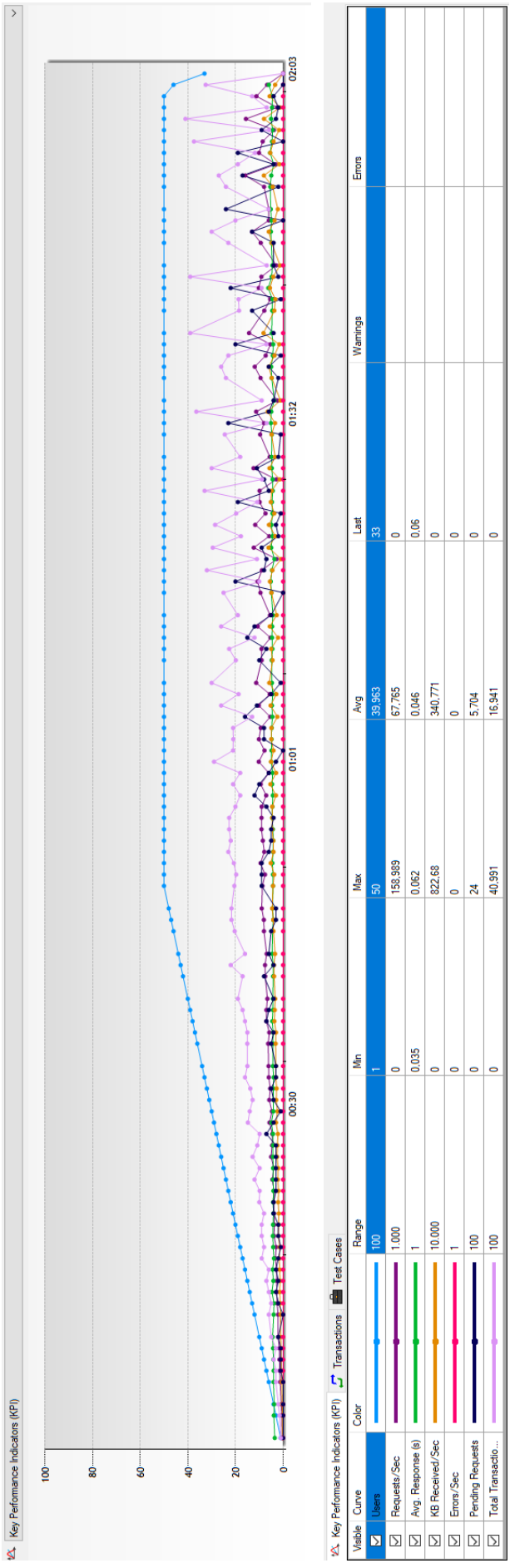

Figura 7.32: Gráficos y resultados prueba de carga.

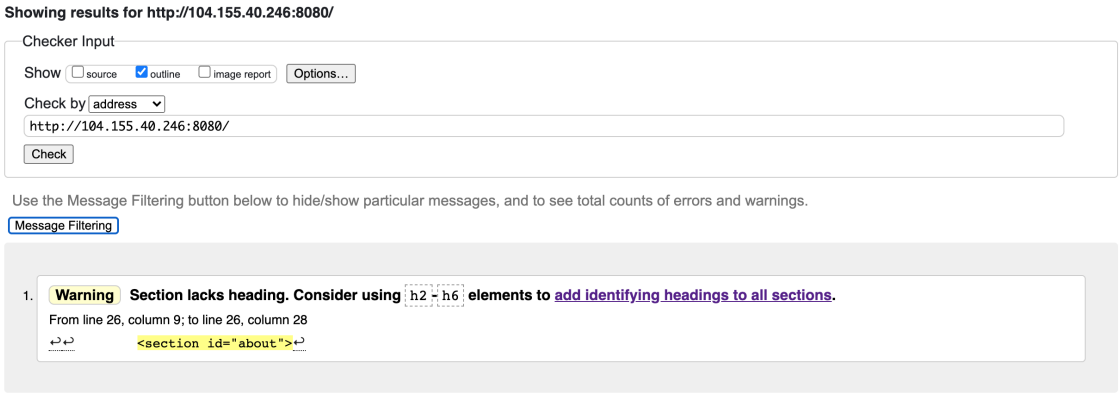

Figura 7.33: Resultado prueba W3C.

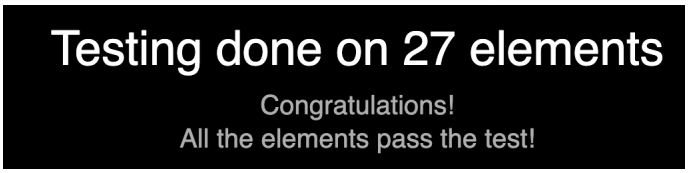

Figura 7.34: Resultado prueba *Check My Colours*.

CAPÍTULO 8
Conclusiones

En este capítulo se expresan las conclusiones del desarrollo del proyecto realizado. Además de aportar un resumen del trabajo realizado a lo largo del proyecto, también se propone una visión de futuro presentando posibles proyectos que pueden derivar del actual.

8.1 Resumen del trabajo realizado

Llegados hasta este punto, podemos afirmar que los objetivos que se marcaron al comienzo del trabajo han sido cumplidos.

En Ingeniería de Software, el concepto de contrato está relacionado con una descripción del comportamiento de los programas que facilita la verificación formal de los mismos. Como primer objetivo, se ha desarrollado un sistema software que pueda ayudar a validar los contratos de programas que se infieren automáticamente o que sean proporcionados manualmente por el ingeniero de software.

La arquitectura del sistema elegida para este proyecto es la de cliente-servidor, donde en el lado del cliente se interactuará con el usuario mediante el uso de una página web, mientras que la parte del servidor se encargará de procesar los datos introducidos por el usuario y realizar las acciones pertinentes.

Para hacer que este sistema este disponible para los usuarios, se ha utilizado los servicios de Google Cloud Platform (GCP), siendo necesario crear una instancia a una VM (*Virtual Machine*) y realizar una configuración de la misma.

Para el desarrollo del sistema, se ha profundizado en el conocimiento aportado por los *SMT Solvers* y su funcionamiento. Además de realizar análisis de las características que ofrecen los resolutores de restricciones modulo teorías (*Satisfiability Modulo Theories*) más utilizados con el fin de determinar qué *solver* era la mejor opción, siendo Z3 la opción elegida.

A lo largo del Capítulo 5 se aborda la explicación de la transformación de los diferentes asertos en funciones de Z3, con el objetivo de mostrar la potencia de esta herramienta para dar valor a las diferentes variables involucradas en las restricciones planteadas y la diferencia de funcionamiento entre la utilización del lenguaje SMT-LIB v2.0 y la librería para un lenguaje de alto nivel como es Python, Z3Py.

Finalmente, gracias al conocimiento extraído del Capítulo 5, se ha podido profundizar en el uso de Z3 con uno de los retos principales del proyecto, la realización de los observadores y programas que son funciones (o un conjunto de funciones en el caso de los programas) que permiten construir un contrato para un programa o sistema. Asimismo se ha conseguido que su integración dentro de la página web de forma satisfactoria.

8.2 Impacto del trabajo realizado

Este proyecto proporciona una solución a la carencia de herramientas para la validación de contratos software. Proporcionando al usuario una herramienta para poder construir sus propios axiomas y evaluarlos o falsificarlos.

8.3 Trabajo futuro

Respecto a las líneas de trabajo a seguir en un futuro, la principal ampliación del proyecto reside en mejorar la precisión de la validación de los contratos software. Por otro lado, se puede realizar una mejora al sistema añadiendo más observadores o programas para permitir que el usuario tenga mayor disponibilidad para construir sus contratos.

A si mismo, también se podría profundizar con mayor detalle en posibilitar la opción de automatizar el proceso de traducción desde el lenguaje fuente a la anotación SMT-LIB v2.0 . Para así crear un prototipo que ofrezca soporte a la validez de contratos a partir del lenguaje fuente.

Otra línea de trabajo posible sería el investigar con mayor profundidad en el uso de herramientas de traducción de código fuente (como puede ser C, Java o Python) a notación SMT, analizando las ventajas e inconvenientes que ofrecen y realizar una comparación con los observadores o programas definidos en este proyecto.

Bibliografía

[1] Smt-lib is an international initiative aimed at facilitating research and development in satisfiability modulo theories (smt). http://smtlib.cs.uiowa.edu.

[2] Uml use case diagrams. https://www.uml-diagrams.org/use-case-diagrams.html.

[3] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. *Gopalakrishnan, Ganesh and Qadeer, Shaz*, pages 171–177, 2011.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard: Version 2.6. *Department of Computer Science, The University of Iowa*, 2017.

[6] Nikolaj Bjørner and Leonardo deMoura. System description: z3 0.1. *Website: http://research. microsoft. com/z3/smtcomp07. pdf*, 2007.

[7] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The why3 platform. *LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition*, 2011.

[8] Thomas Bouton, Diego Caminha B de Oliveira, David Déharbe, and Pascal Fontaine. verit: an open, trustable and efficient smt-solver. *International Conference on Automated Deduction,Springer*, pages 151–156, 2009.

[9] Roberto Bruttomesso. Satisfiability modulo theories: a pragmatic introduction1.

[10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzen, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: A comparative analysis. *Annals of Mathematics and Artificial Intelligence*, 55:63–99, 2006.

[11] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver. *International Conference on Computer Aided Verification, Springer*, 2008.

[12] Andrés Navarro Cadavid, Juan Daniel Fernández Martínez, and Jonathan Morales Vélez. Revisión de metodologías ágiles para el desarrollo de software. *Prospectiva*, 11(2):30–39, 2013.

[13] Miguel Garrido Canalejas. Generación de casos de prueba tipo caja negra mediante restricciones. *TFG - Universidad Complutense de Madrid*, 2018.

[14] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Logic*, 12, 2010.

[15] David R Cok et al. The smt-libv2 language and tools: A tutorial. *Language c*, pages 2010–2011, 2011.

[16] David R Cok, Alberto Griggio, Roberto Bruttomesso, and Morgan Deters. The 2012 smt competition. *SMT IJCAR*, pages 131–142, 2012.

[17] Raul-Ionut Coroban. Contract-based analysis and dynamic verification of c code. 2017-18.

[18] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, 1962.

[19] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

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

[21] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. *Formal Methods: Foundations and Applications,Springer Berlin Heidelberg*, pages 23–36, 2009.

[22] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[23] Matemática Discreta. Tema 1: Introducción a la lógica. *UPV*, 2019.

[24] Bruno Dutertre. Yices 2 manual. *Computer Science Laboratory, SRI International*, 2014.

[25] Andrea Höfler. Smt solver comparison. *Institute for Software Technology (IST) Graz University of Technology*, 2014.

[26] José Joskowicz. Reglas y prácticas en extreme programming. *Universidad de Vigo*, 22, 2008.

[27] Mohd Ehmer Khan, Farmeena Khan, et al. A comparative study of white box, black box and grey box testing techniques. *Citeseer*, 3(6), 2012.

[28] Mark Masse. Rest api design rulebook: Designing consistent restful web service interfaces. *O'Reilly Media, Inc.*, 2011.

[29] Julián Andrés Mera Paz et al. Análisis del proceso de pruebas de calidad de software. *Universidad Cooperativa de Colombia*, 2016.

[30] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

[31] Lev Nachmanson Nikolaj Bjørner, Leonardo de Moura and Christoph Wintersteiger. Programming z3. *Stanford*.

[32] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. *Proceedings of the eighteenth international symposium on Software testing and analysis, ACM*, pages 93–104, 2009.

[33] Carlos Billy Reynoso. Introducción a la arquitectura de software. *Universidad de Buenos Aires*, 33, 2004.

[34] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Åkesson. Smt solvers for job-shop scheduling problems: Models comparison and performance evaluation. *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pages 547–552, 2018.

[35] Ricardo Peña Jaime Sánchez-Hernández, Miguel Garrido, and Javier Sagredo. Smt-based test-case generation with complex preconditions.

[36] Sergi Sanz Carreres. Diseño de interacciones iot geolocalizadas en el ámbito de una ciudad inteligente. *TFG - Universidad Politécnica de Valencia*, 2019.

[37] Michaelmas Term. Logic and proof. *Computer Laboratory, University of Cambridge*, 2000.

[38] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying design by contract to feature-oriented programming. *Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg*.

[39] Lintao Zhang. Sat-solving: From davis-putnam to zchaff and beyond. *Microsoft Research*, 2009.